

Table of Contents

[Overview](#)

[Introduction](#)

[Get Started](#)

[Quick start guide](#)

[Create a storage account](#)

[Blob Storage](#)

[.NET](#)

[Java](#)

[Node.js](#)

[C++](#)

[Python](#)

[PHP](#)

[Ruby](#)

[iOS](#)

[Xamarin](#)

[Queue Storage](#)

[.NET](#)

[Java](#)

[Node.js](#)

[C++](#)

[Python](#)

[PHP](#)

[Ruby](#)

[Table Storage](#)

[.NET](#)

[Java](#)

[Node.js](#)

[C++](#)

[Python](#)

[PHP](#)

[Ruby](#)

[File Storage](#)

[Windows, .NET, PowerShell](#)

[Linux](#)

[Java](#)

[C++](#)

[Python](#)

[How To](#)

[Create a storage account](#)

[Use blobs](#)

[Service overview](#)

[Hot and cool tiers](#)

[Custom domains](#)

[Anonymous access to blobs](#)

[Samples](#)

[Use queues](#)

[Concepts](#)

[Samples](#)

[Use tables](#)

[Overview](#)

[Table design guide](#)

[Samples](#)

[Use files](#)

[Overview](#)

[Troubleshoot Azure Files](#)

[Samples](#)

[Use Virtual Machine Disks](#)

[Premium Storage](#)

[Standard Storage](#)

[Plan and design](#)

[Replication](#)

- [Scalability and performance targets](#)
- [Performance and scalability checklist](#)
- [Concurrency](#)
- [Develop](#)
 - [Samples](#)
 - [Configure connection strings](#)
 - [Use the Storage Emulator](#)
 - [Set and retrieve properties and metadata](#)
- [Manage](#)
 - [PowerShell](#)
 - [Azure CLI 2.0 \(Preview\)](#)
 - [Azure CLI 1.0](#)
 - [Azure Automation](#)
- [Secure](#)
 - [Security guide](#)
 - [Encryption for data at rest](#)
 - [Shared key authentication](#)
 - [Shared access signatures \(SAS\)](#)
 - [Tutorial: Encrypt and decrypt blobs using Azure Key Vault](#)
 - [Client-side encryption](#)
- [Monitor and troubleshoot](#)
 - [Metrics and logging](#)
 - [Troubleshoot disk deletion errors](#)
 - [Troubleshoot File storage](#)
 - [Disaster recovery guidance](#)
- [Transfer Data](#)
 - [Move data to and from Storage](#)
 - [AzCopy command-line utility](#)
 - [Using the Import-Export service](#)
 - [Using the Import-Export Tool](#)
 - [Using the Import-Export Tool \(v1\)](#)
 - [Using the Azure Import-Export Service REST API](#)

Reference

[PowerShell](#)

[Azure CLI](#)

[.NET](#)

[Resource manager](#)

[Data movement](#)

[Blobs, Queues, Tables, and Files](#)

[Java](#)

[Node.js](#)

[Ruby](#)

[Python](#)

[C++](#)

[iOS](#)

[Android](#)

[REST](#)

[Blobs, Queues, Tables, and Files](#)

[Resource provider](#)

[Import/export](#)

Related

[Classic Portal](#)

[Create storage account](#)

[Enable and view metrics](#)

[Monitor, diagnose, and troubleshoot](#)

[Troubleshooting tutorial](#)

Resources

[Pricing](#)

[Azure Storage client tools](#)

[Stack Overflow](#)

[Forum](#)

[Service updates](#)

[Videos](#)

[Azure Storage Explorer](#)

[Storage Explorer \(Preview\)](#)

[Manage blobs with Storage Explorer \(Preview\)](#)

Nuget packages

[Azure Storage Client Library for .NET](#)

[Azure Storage Data Movement Library](#)

[Azure Configuration Manager](#)

Source code

[.NET](#)

[Node.js](#)

[Java](#)

[C++](#)

[PHP](#)

[Ruby](#)

[Python](#)

[iOS](#)

Introduction to Microsoft Azure Storage

1/17/2017 • 20 min to read • [Edit on GitHub](#)

Overview

Azure Storage is the cloud storage solution for modern applications that rely on durability, availability, and scalability to meet the needs of their customers. By reading this article, developers, IT Pros, and business decision makers can learn about:

- What Azure Storage is, and how you can take advantage of it in your cloud, mobile, server, and desktop applications
- What kinds of data you can store with the Azure Storage services: blob (object) data, NoSQL table data, queue messages, and file shares.
- How access to your data in Azure Storage is managed
- How your Azure Storage data is made durable via redundancy and replication
- Where to go next to build your first Azure Storage application

To get up and running with Azure Storage quickly, see [Get started with Azure Storage in five minutes](#).

For details on tools, libraries, and other resources for working with Azure Storage, see [Next Steps](#) below.

What is Azure Storage?

Cloud computing enables new scenarios for applications requiring scalable, durable, and highly available storage for their data – which is exactly why Microsoft developed Azure Storage. In addition to making it possible for developers to build large-scale applications to support new scenarios, Azure Storage also provides the storage foundation for Azure Virtual Machines, a further testament to its robustness.

Azure Storage is massively scalable, so you can store and process hundreds of terabytes of data to support the big data scenarios required by scientific, financial analysis, and media applications. Or you can store the small amounts of data required for a small business website. Wherever your needs fall, you pay only for the data you're storing. Azure Storage currently stores tens of trillions of unique customer objects, and handles millions of requests per second on average.

Azure Storage is elastic, so you can design applications for a large global audience, and scale those applications as needed - both in terms of the amount of data stored and the number of requests made against it. You pay only for what you use, and only when you use it.

Azure Storage uses an auto-partitioning system that automatically load-balances your data based on traffic. This means that as the demands on your application grow, Azure Storage automatically allocates the appropriate resources to meet them.

Azure Storage is accessible from anywhere in the world, from any type of application, whether it's running in the cloud, on the desktop, on an on-premises server, or on a mobile or tablet device. You can use Azure Storage in mobile scenarios where the application stores a subset of data on the device and synchronizes it with a full set of data stored in the cloud.

Azure Storage supports clients using a diverse set of operating systems (including Windows and Linux) and a variety of programming languages (including .NET, Java, Node.js, Python, Ruby, PHP and C++ and mobile programming languages) for convenient development. Azure Storage also exposes data resources via simple REST APIs, which are available to any client capable of sending and receiving data via HTTP/HTTPS.

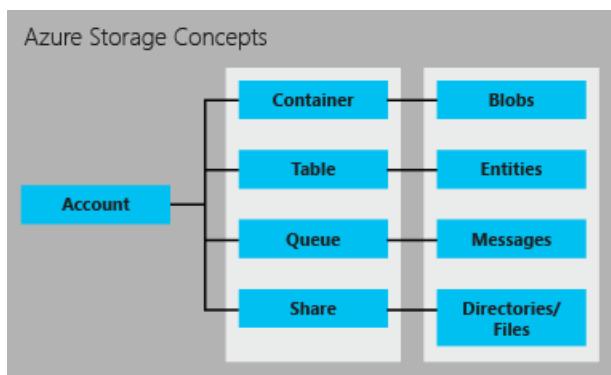
Azure Premium Storage delivers high-performance, low-latency disk support for I/O intensive workloads running on Azure Virtual Machines. With Azure Premium Storage, you can attach multiple persistent data disks to a virtual machine and configure them to meet your performance requirements. Each data disk is backed by an SSD disk in Azure Premium Storage for maximum I/O performance. See [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#) for more details.

Introducing the Azure Storage Services

Azure storage provides the following four services: Blob storage, Table storage, Queue storage, and File storage.

- Blob Storage stores unstructured object data. A blob can be any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as Object storage.
- Table Storage stores structured datasets. Table storage is a NoSQL key-attribute data store, which allows for rapid development and fast access to large quantities of data.
- Queue Storage provides reliable messaging for workflow processing and for communication between components of cloud services.
- File Storage offers shared storage for legacy applications using the standard SMB protocol. Azure virtual machines and cloud services can share file data across application components via mounted shares, and on-premises applications can access file data in a share via the File service REST API.

An Azure storage account is a secure account that gives you access to services in Azure Storage. Your storage account provides the unique namespace for your storage resources. The image below shows the relationships between the Azure storage resources in a storage account:



There are two types of storage accounts:

General-purpose Storage Accounts

A general-purpose storage account gives you access to Azure Storage services such as Tables, Queues, Files, Blobs and Azure virtual machine disks under a single account. This type of storage account has two performance tiers:

- A standard storage performance tier which allows you to store Tables, Queues, Files, Blobs and Azure virtual machine disks.
- A premium storage performance tier which currently only supports Azure virtual machine disks. See [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#) for an in-depth overview of Premium storage.

Blob Storage Accounts

A Blob storage account is a specialized storage account for storing your unstructured data as blobs (objects) in Azure Storage. Blob storage accounts are similar to your existing general-purpose storage accounts and share all the great durability, availability, scalability, and performance features that you use today including 100% API consistency for block blobs and append blobs. For applications requiring only block or append blob storage, we recommend using Blob storage accounts.

NOTE

Blob storage accounts support only block and append blobs, and not page blobs.

Blob storage accounts expose the **Access Tier** attribute which can be specified during account creation and modified later as needed. There are two types of access tiers that can be specified based on your data access pattern:

- A **Hot** access tier which indicates that the objects in the storage account will be more frequently accessed. This allows you to store data at a lower access cost.
- A **Cool** access tier which indicates that the objects in the storage account will be less frequently accessed. This allows you to store data at a lower data storage cost.

If there is a change in the usage pattern of your data, you can also switch between these access tiers at any time. Changing the access tier may result in additional charges. Please see [Pricing and billing for Blob storage accounts](#) for more details.

For more details on Blob storage accounts, see [Azure Blob Storage: Cool and Hot tiers](#).

Before you can create a storage account, you must have an Azure subscription, which is a plan that gives you access to a variety of Azure services. You can get started with Azure with a [free account](#). Once you decide to purchase a subscription plan, you can choose from a variety of [purchase options](#). If you're an [MSDN subscriber](#), you get free monthly credits that you can use with Azure services, including Azure Storage. See [Azure Storage Pricing](#) for information on volume pricing.

To learn how to create a storage account, see [Create a storage account](#) for more details. You can create up to 100 uniquely named storage accounts with a single subscription. See [Azure Storage Scalability and Performance Targets](#) for details about storage account limits.

Storage Service Versions

The Azure Storage services are regularly updated with support for new features. The Azure Storage services REST API reference describes each supported version and its features. We recommend that you use the latest version whenever possible. For information on the latest version of the Azure Storage services, as well as information on previous versions, see [Versioning for the Azure Storage Services](#).

Blob Storage

For users with large amounts of unstructured object data to store in the cloud, Blob storage offers a cost-effective and scalable solution. You can use Blob storage to store content such as:

- Documents
- Social data such as photos, videos, music, and blogs
- Backups of files, computers, databases, and devices
- Images and text for web applications
- Configuration data for cloud applications
- Big data, such as logs and other large datasets

Every blob is organized into a container. Containers also provide a useful way to assign security policies to groups of objects. A storage account can contain any number of containers, and a container can contain any number of blobs, up to the 500 TB capacity limit of the storage account.

Blob storage offers three types of blobs, block blobs, append blobs, and page blobs (disks).

- Block blobs are optimized for streaming and storing cloud objects, and are a good choice for storing documents, media files, backups etc.

- Append blobs are similar to block blobs, but are optimized for append operations. An append blob can be updated only by adding a new block to the end. Append blobs are a good choice for scenarios such as logging, where new data needs to be written only to the end of the blob.
- Page blobs are optimized for representing IaaS disks and supporting random writes, and may be up to 1 TB in size. An Azure virtual machine network attached IaaS disk is a VHD stored as a page blob.

For very large datasets where network constraints make uploading or downloading data to Blob storage over the wire unrealistic, you can ship a hard drive to Microsoft to import or export data directly from the data center. See [Use the Microsoft Azure Import/Export Service to Transfer Data to Blob Storage](#).

Table storage

Modern applications often demand data stores with greater scalability and flexibility than previous generations of software required. Table storage offers highly available, massively scalable storage, so that your application can automatically scale to meet user demand. Table storage is Microsoft's NoSQL key/attribute store – it has a schemaless design, making it different from traditional relational databases. With a schemaless data store, it's easy to adapt your data as the needs of your application evolve. Table storage is easy to use, so developers can create applications quickly. Access to data is fast and cost-effective for all kinds of applications. Table storage is typically significantly lower in cost than traditional SQL for similar volumes of data.

Table storage is a key-attribute store, meaning that every value in a table is stored with a typed property name. The property name can be used for filtering and specifying selection criteria. A collection of properties and their values comprise an entity. Since Table storage is schemaless, two entities in the same table can contain different collections of properties, and those properties can be of different types.

You can use Table storage to store flexible datasets, such as user data for web applications, address books, device information, and any other type of metadata that your service requires. You can store any number of entities in a table, and a storage account may contain any number of tables, up to the capacity limit of the storage account.

Like Blobs and Queues, developers can manage and access Table storage using standard REST protocols, however Table storage also supports a subset of the OData protocol, simplifying advanced querying capabilities and enabling both JSON and AtomPub (XML based) formats.

For today's Internet-based applications, NoSQL databases like Table storage offer a popular alternative to traditional relational databases.

Queue Storage

In designing applications for scale, application components are often decoupled, so that they can scale independently. Queue storage provides a reliable messaging solution for asynchronous communication between application components, whether they are running in the cloud, on the desktop, on an on-premises server, or on a mobile device. Queue storage also supports managing asynchronous tasks and building process workflows.

A storage account can contain any number of queues. A queue can contain any number of messages, up to the capacity limit of the storage account. Individual messages may be up to 64 KB in size.

File Storage

Azure File storage offers cloud-based SMB file shares, so that you can migrate legacy applications that rely on file shares to Azure quickly and without costly rewrites. With Azure File storage, applications running in Azure virtual machines or cloud services can mount a file share in the cloud, just as a desktop application mounts a typical SMB share. Any number of application components can then mount and access the File storage share simultaneously.

Since a File storage share is a standard SMB file share, applications running in Azure can access data in the share via file system I/O APIs. Developers can therefore leverage their existing code and skills to migrate existing

applications. IT Pros can use PowerShell cmdlets to create, mount, and manage File storage shares as part of the administration of Azure applications.

Like the other Azure storage services, File storage exposes a REST API for accessing data in a share. On-premise applications can call the File storage REST API to access data in a file share. This way, an enterprise can choose to migrate some legacy applications to Azure and continue running others from within their own organization. Note that mounting a file share is only possible for applications running in Azure; an on-premises application may only access the file share via the REST API.

Distributed applications can also use File storage to store and share useful application data and development and testing tools. For example, an application may store configuration files and diagnostic data such as logs, metrics, and crash dumps in a File storage share so that they are available to multiple virtual machines or roles. Developers and administrators can store utilities that they need to build or manage an application in a File storage share that is available to all components, rather than installing them on every virtual machine or role instance.

Access to Blob, Table, Queue, and File Resources

By default, only the storage account owner can access resources in the storage account. For the security of your data, every request made against resources in your account must be authenticated. Authentication relies on a Shared Key model. Blobs can also be configured to support anonymous authentication.

Your storage account is assigned two private access keys on creation that are used for authentication. Having two keys ensures that your application remains available when you regularly regenerate the keys as a common security key management practice.

If you do need to allow users controlled access to your storage resources, then you can create a shared access signature. A shared access signature (SAS) is a token that can be appended to a URL that enables delegated access to a storage resource. Anyone who possesses the token can access the resource it points to with the permissions it specifies, for the period of time that it is valid. Beginning with version 2015-04-05, Azure Storage supports two kinds of shared access signatures: service SAS and account SAS.

The service SAS delegates access to a resource in just one of the storage services: the Blob, Queue, Table, or File service.

An account SAS delegates access to resources in one or more of the storage services. You can delegate access to service-level operations that are not available with a service SAS. You can also delegate access to read, write, and delete operations on blob containers, tables, queues, and file shares that are not permitted with a service SAS.

Finally, you can specify that a container and its blobs, or a specific blob, are available for public access. When you indicate that a container or blob is public, anyone can read it anonymously; no authentication is required. Public containers and blobs are useful for exposing resources such as media and documents that are hosted on websites. To decrease network latency for a global audience, you can cache blob data used by websites with the Azure CDN.

See [Using Shared Access Signatures \(SAS\)](#) for more information on shared access signatures. See [Manage anonymous read access to containers and blobs](#) and [Authentication for the Azure Storage Services](#) for more information on secure access to your storage account.

Replication for Durability and High Availability

The data in your Microsoft Azure storage account is always replicated to ensure durability and high availability. Replication copies your data, either within the same data center, or to a second data center, depending on which replication option you choose. Replication protects your data and preserves your application up-time in the event of transient hardware failures. If your data is replicated to a second data center, that also protects your data against a catastrophic failure in the primary location.

Replication ensures that your storage account meets the [Service-Level Agreement \(SLA\) for Storage](#) even in the

face of failures. See the SLA for information about Azure Storage guarantees for durability and availability.

When you create a storage account, you can select one of the following replication options:

- **Locally redundant storage (LRS).** Locally redundant storage maintains three copies of your data. LRS is replicated three times within a single data center in a single region. LRS protects your data from normal hardware failures, but not from the failure of a single data center.

LRS is offered at a discount. For maximum durability, we recommend that you use geo-redundant storage, described below.

- **Zone-redundant storage (ZRS).** Zone-redundant storage maintains three copies of your data. ZRS is replicated three times across two to three facilities, either within a single region or across two regions, providing higher durability than LRS. ZRS ensures that your data is durable within a single region.

ZRS provides a higher level of durability than LRS; however, for maximum durability, we recommend that you use geo-redundant storage, described below.

NOTE

ZRS is currently available only for block blobs, and is only supported for versions 2014-02-14 and later.

Once you have created your storage account and selected ZRS, you cannot convert it to use to any other type of replication, or vice versa.

- **Geo-redundant storage (GRS).** GRS maintains six copies of your data. With GRS, your data is replicated three times within the primary region, and is also replicated three times in a secondary region hundreds of miles away from the primary region, providing the highest level of durability. In the event of a failure at the primary region, Azure Storage will failover to the secondary region. GRS ensures that your data is durable in two separate regions.

For information about primary and secondary pairings by region, see [Azure Regions](#).

- **Read-access geo-redundant storage (RA-GRS).** Read-access geo-redundant storage replicates your data to a secondary geographic location, and also provides read access to your data in the secondary location. Read-access geo-redundant storage allows you to access your data from either the primary or the secondary location, in the event that one location becomes unavailable. Read-access geo-redundant storage is the default option for your storage account by default when you create it.

IMPORTANT

You can change how your data is replicated after your storage account has been created, unless you specified ZRS when you created the account. However, note that you may incur an additional one-time data transfer cost if you switch from LRS to GRS or RA-GRS.

See [Azure Storage replication](#) for additional details about storage replication options.

For pricing information for storage account replication, see [Azure Storage Pricing](#). See [Azure Regions](#) for more information about what services are available in each region.

For architectural details about durability with Azure Storage, see [SOSP Paper - Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency](#).

Transferring Data to and from Azure Storage

You can use the AzCopy command-line utility to copy blob, file, and table data within your storage account or across storage accounts. See [Transfer data with the AzCopy Command-Line Utility](#) for more information.

AzCopy is built on top of the [Azure Data Movement Library](#), which is currently available in preview.

The Azure Import/Export service provides a way to import blob data into or export blob data from your storage account via a hard drive disk mailed to the Azure data center. For more information about the Import/Export service, see [Use the Microsoft Azure Import/Export Service to Transfer Data to Blob Storage](#).

Pricing

You are billed for Azure Storage usage based on your storage account. Storage costs are based on the following factors: region/location, account type, storage capacity, replication scheme, storage transactions, and data egress.

- Region refers to the geographical region in which your account is based.
- Account type refers to whether you are using a general-purpose storage account or a Blob storage account. With a Blob storage account, the access tier also determines the billing model for the account.
- Storage capacity refers to how much of your storage account allotment you are using to store data.
- Replication determines how many copies of your data are maintained at one time, and in what locations.
- Transactions refer to all read and write operations to Azure Storage.
- Data egress refers to data transferred out of an Azure region. When the data in your storage account is accessed by an application that is not running in the same region, you are charged for data egress. (For Azure services, you can take steps to group your data and services in the same data centers to reduce or eliminate data egress charges.)

The [Azure Storage Pricing](#) page provides detailed pricing information based on account type, storage capacity, replication, and transactions. The [Data Transfers Pricing Details](#) provides detailed pricing information for data egress. You can use the [Azure Storage Pricing Calculator](#) to help estimate your costs.

Storage APIs, Libraries, and Tools

Azure Storage resources can be accessed by any language that can make HTTP/HTTPS requests. Additionally, Azure Storage offers programming libraries for several popular languages. These libraries simplify many aspects of working with Azure Storage by handling details such as synchronous and asynchronous invocation, batching of operations, exception management, automatic retries, operational behavior and so forth. Libraries are currently available for the following languages and platforms, with others in the pipeline:

Azure Storage Data Services

- [Storage Services REST API](#)
- [Storage Client Library for .NET, Windows Phone, and Windows Runtime](#)
- [Storage Client Library for C++](#)
- [Storage Client Library for Java/Android](#)
- [Storage Client Library for Node.js](#)
- [Storage Client Library for PHP](#)
- [Storage Client Library for Ruby](#)
- [Storage Client Library for Python](#)
- [Storage Cmdlets for PowerShell 1.0](#)

Azure Storage Management Services

- [Storage Resource Provider REST API Reference](#)
- [Storage Resource Provider Client Library for .NET](#)
- [Storage Resource Provider Cmdlets for PowerShell 1.0](#)
- [Storage Service Management REST API \(Classic\)](#)

Azure Storage Data Movement Services

- [Storage Import/Export Service REST API](#)
- [Storage Data Movement Client Library for .NET](#)

Tools and Utilities

- [Azure Storage Explorer](#)
- [Azure Storage Client Tools](#)
- [Azure SDKs and Tools](#)
- [Azure Storage Emulator](#)
- [Azure PowerShell](#)
- [AzCopy Command-Line Utility](#)

Next Steps

To learn more about Azure Storage, explore these resources:

Documentation

- [Azure Storage Documentation](#)

For Administrators

- [Using Azure PowerShell with Azure Storage](#)
- [Using Azure CLI with Azure Storage](#)

For .NET Developers

- [Get started with Azure Blob storage using .NET](#)
- [Get started with Azure Table storage using .NET](#)
- [Get started with Azure Queue storage using .NET](#)
- [Get started with Azure File storage on Windows](#)

For Java/Android Developers

- [How to use Blob storage from Java](#)
- [How to use Table storage from Java](#)
- [How to use Queue storage from Java](#)
- [How to use File storage from Java](#)

For Node.js Developers

- [How to use Blob storage from Nodejs](#)
- [How to use Table storage from Nodejs](#)
- [How to use Queue storage from Nodejs](#)

For PHP Developers

- [How to use Blob storage from PHP](#)
- [How to use Table storage from PHP](#)
- [How to use Queue storage from PHP](#)

For Ruby Developers

- [How to use Blob storage from Ruby](#)
- [How to use Table storage from Ruby](#)
- [How to use Queue storage from Ruby](#)

For Python Developers

- [How to use Blob storage from Python](#)
- [How to use Table storage from Python](#)

- [How to use Queue storage from Python](#)
- [How to use File storage from Python](#)

Next steps

- [Create a storage account](#)
- [Get started with Azure Storage in five minutes](#)

Get started with Azure Storage in five minutes

1/17/2017 • 4 min to read • [Edit on GitHub](#)

Overview

It's easy to get started developing with Azure Storage. This tutorial shows you how to get an Azure Storage application up and running quickly. You'll use the Quick Start templates included with the Azure SDK for .NET. These Quick Starts contain ready-to-run code that demonstrates some basic programming scenarios with Azure Storage.

To learn more about Azure Storage before diving into the code, see [Next Steps](#).

Prerequisites

You'll need the following prerequisites before you start:

1. To compile and build the application, you'll need a version of [Visual Studio](#) installed on your computer.
2. Install the latest version [Azure SDK for .NET](#). The SDK includes the Azure QuickStart sample projects, the Azure storage emulator, and the [Azure Storage Client Library for .NET](#).
3. Make sure that you have [.NET Framework 4.5](#) installed on your computer, as it is required by the Azure QuickStart sample projects that we'll be using in this tutorial.

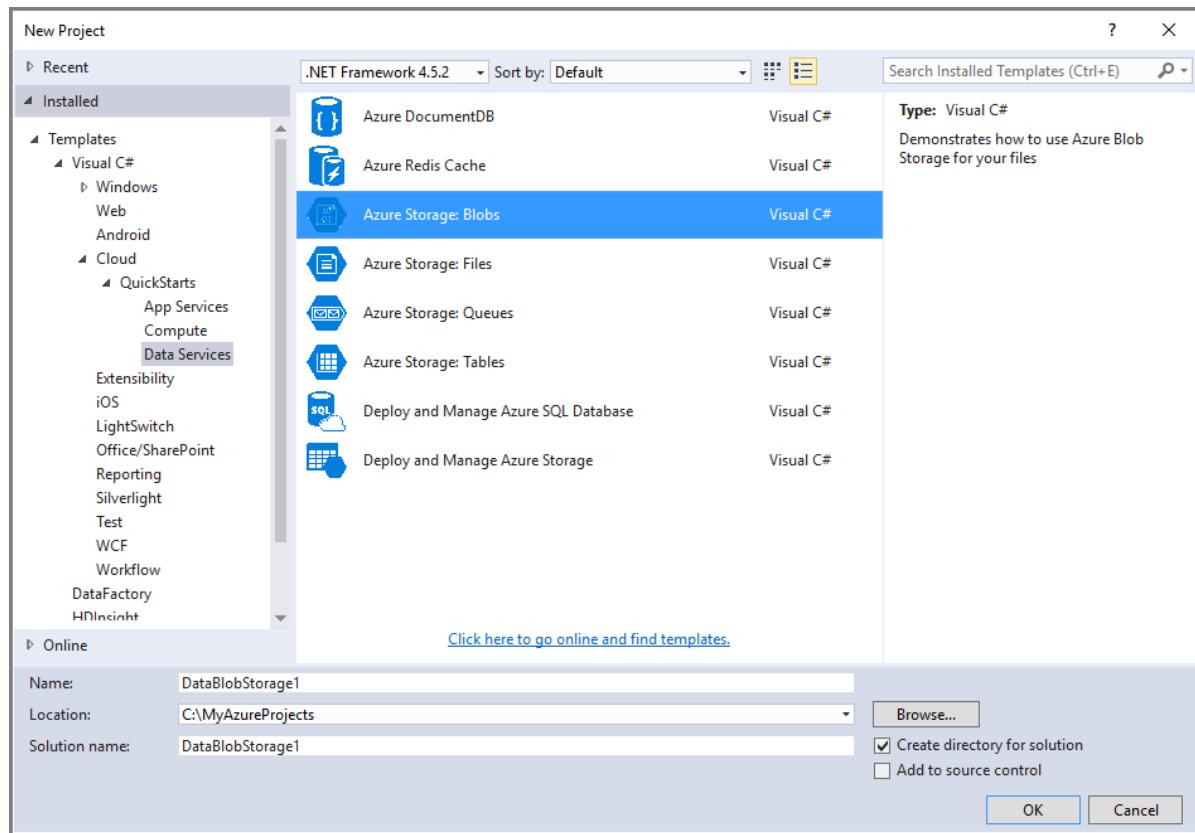
If you are not sure which version of .NET Framework is installed in your computer, see [How to: Determine Which .NET Framework Versions Are Installed](#). Or, press the **Start** button or the Windows key, type **Control Panel**. Then, click **Programs > Programs and Features**, and determine whether the .NET Framework 4.5 is listed among the installed programs.

4. You'll need an Azure subscription and an Azure storage account.
 - To get an Azure subscription, see [Free Trial](#), [Purchase Options](#), and [Member Offers](#) (for members of MSDN, Microsoft Partner Network, and BizSpark, and other Microsoft programs).
 - To create a storage account in Azure, see [How to create a storage account](#).

Run your first Azure Storage application against Azure Storage in the cloud

Once you have an account, you can create a simple Azure Storage application using one of the Azure Quick Starts sample projects in Visual Studio. This tutorial focuses on the sample projects for Azure Storage: **Azure Storage: Blobs**, **Azure Storage: Files**, **Azure Storage: Queues**, and **Azure Storage: Tables**:

1. Start Visual Studio.
2. From the **File** menu, click **New Project**.
3. In the **New Project** dialog box, click **Installed > Templates > Visual C# > Cloud > QuickStarts > Data Services**.
 - a. Choose one of the following templates: **Azure Storage: Blobs**, **Azure Storage: Files**, **Azure Storage: Queues**, or **Azure Storage: Tables**.
 - b. Make sure that **.NET Framework 4.5** is selected as the target framework.
 - c. Specify a name for your project and create the new Visual Studio solution, as shown:



You may want to review the source code before running the application. To review the code, select **Solution Explorer** on the **View** menu in Visual Studio. Then, double-click the Program.cs file.

Next, run the sample application:

1. In Visual Studio, select **Solution Explorer** on the **View** menu. Open the App.config file and comment out the connection string for the Azure storage emulator:

```
<!--<add key="StorageConnectionString" value = "UseDevelopmentStorage=true;" /-->
```

2. Uncomment the connection string for the Azure Storage Service and provide the storage account name and access key in the App.config file:

```
<add key="StorageConnectionString" value="DefaultEndpointsProtocol=https;AccountName=[AccountName];AccountKey=[AccountKey]"
```

To retrieve your storage account access key, see [Manage your storage access keys](#).

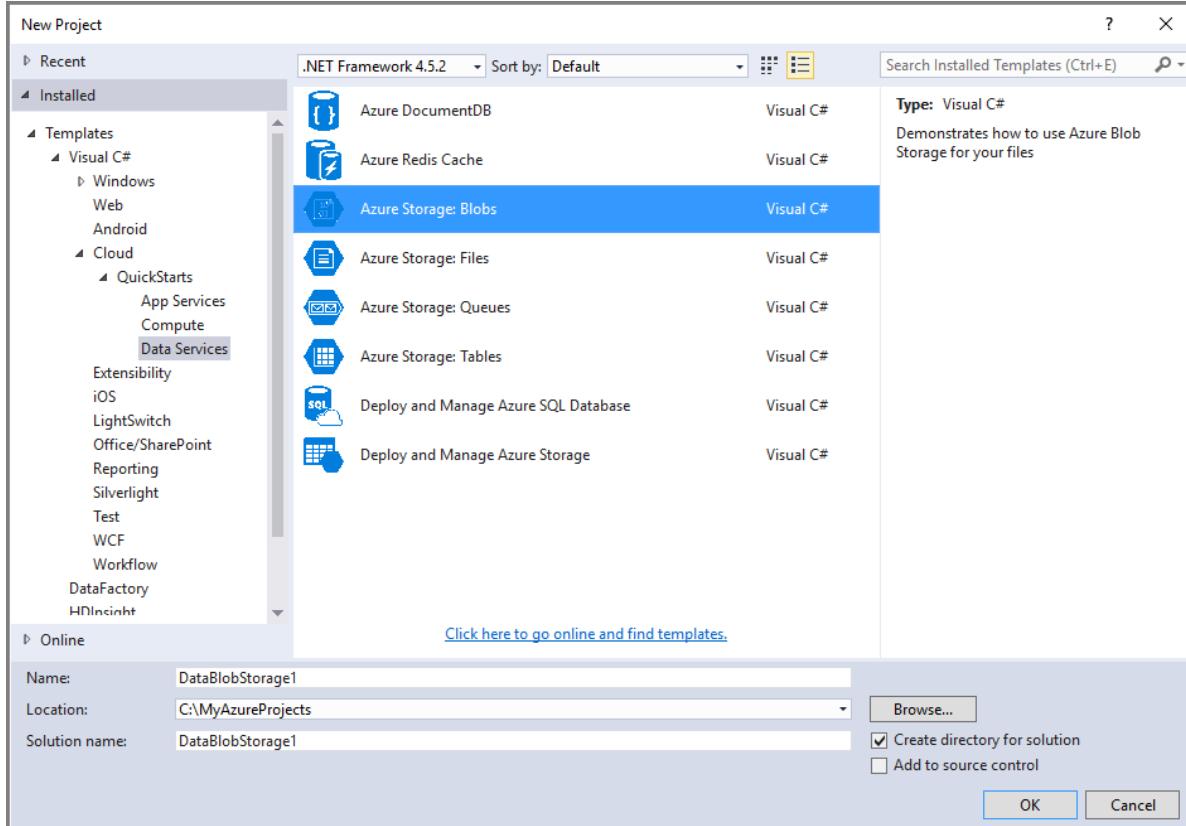
3. After you provide the storage account name and access key in the App.config file, on the **File** menu, click **Save All** to save all the project files.
4. On the **Build** menu, click **Build Solution**.
5. On the **Debug** menu, Press **F11** to run the solution step by step or press **F5** to run the solution.

Run your first Azure Storage application locally against the Azure Storage Emulator

The [Azure Storage Emulator](#) provides a local environment that emulates the Azure Blob, Queue, and Table services for development purposes. You can use the storage emulator to test your storage application locally, without creating an Azure subscription or storage account, and without incurring any cost.

To try it, let's create a simple Azure Storage application using one of the Azure Quick Starts sample projects in Visual Studio. This tutorial focuses on the **Azure Blob Storage**, **Azure Table Storage**, and **Azure Queue Storage** sample projects:

1. Start Visual Studio.
2. From the **File** menu, click **New Project**.
3. In the **New Project** dialog box, click **Installed > Templates > Visual C# > Cloud > QuickStarts > Data Services**. a. Choose one of the following templates: **Azure Storage: Blobs**, **Azure Storage: Files**, **Azure Storage: Queues**, or **Azure Storage: Tables**. b. Make sure that **.NET Framework 4.5** is selected as the target framework. c. Specify a name for your project and create the new Visual Studio solution, as shown:



4. In Visual Studio, select **Solution Explorer** on the **View** menu. Open the App.config file and comment out the connection string for your Azure storage account if you have already added one. Then uncomment the connection string for the Azure storage emulator:

```
<add key="StorageConnectionString" value = "UseDevelopmentStorage=true;"/>
```

You may want to review the source code before running the application. To review the code, select **Solution Explorer** on the **View** menu in Visual Studio. Then, double-click the Program.cs file.

Next, run the sample application in the Azure Storage Emulator:

1. Press the **Start** button or the Windows key, search for *Microsoft Azure Storage emulator*, and start the application. When the emulator starts, you'll see an icon and a notification in the Windows Task View area.
2. In Visual Studio, click **Build Solution** on the **Build** menu.
3. On the **Debug** menu, press **F11** to run the solution step by step, or press **F5** to run the solution from start to finish.

Next Steps

See these resources to learn more about Azure Storage:

- [Introduction to Microsoft Azure Storage](#)
- [Get started with Azure Storage Explorer](#)
- [Get started with Azure Blob storage using .NET](#)
- [Get started with Azure Table storage using .NET](#)

- [Get started with Azure Queue Storage using .NET](#)
- [Get started with Azure File storage on Windows](#)
- [Transfer data with the AzCopy Command-Line Utility](#)
- [Azure Storage Documentation](#)
- [Microsoft Azure Storage Client Library for .NET](#)
- [Azure Storage Services REST API](#)

About Azure storage accounts

1/17/2017 • 11 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

An Azure storage account provides a unique namespace to store and access your Azure Storage data objects. All objects in a storage account are billed together as a group. By default, the data in your account is available only to you, the account owner.

There are two types of storage accounts:

General-purpose Storage Accounts

A general-purpose storage account gives you access to Azure Storage services such as Tables, Queues, Files, Blobs and Azure virtual machine disks under a single account. This type of storage account has two performance tiers:

- A standard storage performance tier which allows you to store Tables, Queues, Files, Blobs and Azure virtual machine disks.
- A premium storage performance tier which currently only supports Azure virtual machine disks. See [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#) for an in-depth overview of Premium storage.

Blob Storage Accounts

A Blob storage account is a specialized storage account for storing your unstructured data as blobs (objects) in Azure Storage. Blob storage accounts are similar to your existing general-purpose storage accounts and share all the great durability, availability, scalability, and performance features that you use today including 100% API consistency for block blobs and append blobs. For applications requiring only block or append blob storage, we recommend using Blob storage accounts.

NOTE

Blob storage accounts support only block and append blobs, and not page blobs.

Blob storage accounts expose the **Access Tier** attribute which can be specified during account creation and modified later as needed. There are two types of access tiers that can be specified based on your data access pattern:

- A **Hot** access tier which indicates that the objects in the storage account will be more frequently accessed. This allows you to store data at a lower access cost.
- A **Cool** access tier which indicates that the objects in the storage account will be less frequently accessed. This allows you to store data at a lower data storage cost.

If there is a change in the usage pattern of your data, you can also switch between these access tiers at any time. Changing the access tier may result in additional charges. Please see [Pricing and billing for Blob storage accounts](#)

for more details.

For more details on Blob storage accounts, see [Azure Blob Storage: Cool and Hot tiers](#).

Before you can create a storage account, you must have an Azure subscription, which is a plan that gives you access to a variety of Azure services. You can get started with Azure with a [free account](#). Once you decide to purchase a subscription plan, you can choose from a variety of [purchase options](#). If you're an [MSDN subscriber](#), you get free monthly credits that you can use with Azure services, including Azure Storage. See [Azure Storage Pricing](#) for information on volume pricing.

To learn how to create a storage account, see [Create a storage account](#) for more details. You can create up to 100 uniquely named storage accounts with a single subscription. See [Azure Storage Scalability and Performance Targets](#) for details about storage account limits.

Storage account billing

You are billed for Azure Storage usage based on your storage account. Storage costs are based on the following factors: region/location, account type, storage capacity, replication scheme, storage transactions, and data egress.

- Region refers to the geographical region in which your account is based.
- Account type refers to whether you are using a general-purpose storage account or a Blob storage account. With a Blob storage account, the access tier also determines the billing model for the account.
- Storage capacity refers to how much of your storage account allotment you are using to store data.
- Replication determines how many copies of your data are maintained at one time, and in what locations.
- Transactions refer to all read and write operations to Azure Storage.
- Data egress refers to data transferred out of an Azure region. When the data in your storage account is accessed by an application that is not running in the same region, you are charged for data egress. (For Azure services, you can take steps to group your data and services in the same data centers to reduce or eliminate data egress charges.)

The [Azure Storage Pricing](#) page provides detailed pricing information based on account type, storage capacity, replication, and transactions. The [Data Transfers Pricing Details](#) provides detailed pricing information for data egress. You can use the [Azure Storage Pricing Calculator](#) to help estimate your costs.

NOTE

When you create an Azure virtual machine, a storage account is created for you automatically in the deployment location if you do not already have a storage account in that location. So it's not necessary to follow the steps below to create a storage account for your virtual machine disks. The storage account name will be based on the virtual machine name. See the [Azure Virtual Machines documentation](#) for more details.

Storage account endpoints

Every object that you store in Azure Storage has a unique URL address. The storage account name forms the subdomain of that address. The combination of subdomain and domain name, which is specific to each service, forms an *endpoint* for your storage account.

For example, if your storage account is named *mystorageaccount*, then the default endpoints for your storage account are:

- Blob service: `http://mystorageaccount.blob.core.windows.net`
- Table service: `http://mystorageaccount.table.core.windows.net`
- Queue service: `http://mystorageaccount.queue.core.windows.net`
- File service: `http://mystorageaccount.file.core.windows.net`

NOTE

A Blob storage account only exposes the Blob service endpoint.

The URL for accessing an object in a storage account is built by appending the object's location in the storage account to the endpoint. For example, a blob address might have this format:

`http://mystorageaccount.blob.core.windows.net/mycontainer/myblob`.

You can also configure a custom domain name to use with your storage account. For classic storage accounts, see [Configure a custom domain Name for your Blob Storage Endpoint](#) for details. For Resource Manager storage accounts, this capability has not been added to the [Azure portal](#) yet, but you can configure it with PowerShell. For more information, see the [Set-AzureRmStorageAccount](#) cmdlet.

Create a storage account

1. Sign in to the [Azure portal](#).
2. On the Hub menu, select **New -> Storage -> Storage account**.
3. Enter a name for your storage account. See [Storage account endpoints](#) for details about how the storage account name will be used to address your objects in Azure Storage.

NOTE

Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only.

Your storage account name must be unique within Azure. The Azure portal will indicate if the storage account name you select is already in use.

4. Specify the deployment model to be used: **Resource Manager** or **Classic**. **Resource Manager** is the recommended deployment model. For more information, see [Understanding Resource Manager deployment and classic deployment](#).

NOTE

Blob storage accounts can only be created using the Resource Manager deployment model.

5. Select the type of storage account: **General purpose** or **Blob storage**. **General purpose** is the default.

If **General purpose** was selected, then specify the performance tier: **Standard** or **Premium**. The default is **Standard**. For more details on standard and premium storage accounts, see [Introduction to Microsoft Azure Storage](#) and [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#).

If **Blob Storage** was selected, then specify the access tier: **Hot** or **Cool**. The default is **Hot**. See [Azure Blob Storage: Cool and Hot tiers](#) for more details.

6. Select the replication option for the storage account: **LRS**, **GRS**, **RA-GRS**, or **ZRS**. The default is **RA-GRS**. For more details on Azure Storage replication options, see [Azure Storage replication](#).
7. Select the subscription in which you want to create the new storage account.
8. Specify a new resource group or select an existing resource group. For more information on resource groups, see [Azure Resource Manager overview](#).
9. Select the geographic location for your storage account. See [Azure Regions](#) for more information about what services are available in which region.
10. Click **Create** to create the storage account.

Manage your storage account

Change your account configuration

After you create your storage account, you can modify its configuration, such as changing the replication option used for the account or changing the access tier for a Blob storage account. In the [Azure portal](#), navigate to your storage account, click **All settings** and then click **Configuration** to view and/or change the account configuration.

NOTE

Depending on the performance tier you chose when creating the storage account, some replication options may not be available.

Changing the replication option will change your pricing. For more details, see [Azure Storage Pricing](#) page.

For Blob storage accounts, changing the access tier may incur charges for the change in addition to changing your pricing. Please see the [Blob storage accounts - Pricing and Billing](#) for more details.

Manage your storage access keys

When you create a storage account, Azure generates two 512-bit storage access keys, which are used for authentication when the storage account is accessed. By providing two storage access keys, Azure enables you to regenerate the keys with no interruption to your storage service or access to that service.

NOTE

We recommend that you avoid sharing your storage access keys with anyone else. To permit access to storage resources without giving out your access keys, you can use a *shared access signature*. A shared access signature provides access to a resource in your account for an interval that you define and with the permissions that you specify. See [Using Shared Access Signatures \(SAS\)](#) for more information.

View and copy storage access keys

In the [Azure portal](#), navigate to your storage account, click **All settings** and then click **Access keys** to view, copy, and regenerate your account access keys. The **Access Keys** blade also includes pre-configured connection strings using your primary and secondary keys that you can copy to use in your applications.

Regenerate storage access keys

We recommend that you change the access keys to your storage account periodically to help keep your storage connections secure. Two access keys are assigned so that you can maintain connections to the storage account by using one access key while you regenerate the other access key.

WARNING

Regenerating your access keys can affect services in Azure as well as your own applications that are dependent on the storage account. All clients that use the access key to access the storage account must be updated to use the new key.

Media services - If you have media services that are dependent on your storage account, you must re-sync the access keys with your media service after you regenerate the keys.

Applications - If you have web applications or cloud services that use the storage account, you will lose the connections if you regenerate keys, unless you roll your keys.

Storage Explorers - If you are using any [storage explorer applications](#), you will probably need to update the storage key used by those applications.

Here is the process for rotating your storage access keys:

1. Update the connection strings in your application code to reference the secondary access key of the storage account.
2. Regenerate the primary access key for your storage account. On the **Access Keys** blade, click **Regenerate Key1**, and then click **Yes** to confirm that you want to generate a new key.
3. Update the connection strings in your code to reference the new primary access key.
4. Regenerate the secondary access key in the same manner.

Delete a storage account

To remove a storage account that you are no longer using, navigate to the storage account in the [Azure portal](#), and click **Delete**. Deleting a storage account deletes the entire account, including all data in the account.

WARNING

It's not possible to restore a deleted storage account or retrieve any of the content that it contained before deletion. Be sure to back up anything you want to save before you delete the account. This also holds true for any resources in the account—once you delete a blob, table, queue, or file, it is permanently deleted.

To delete a storage account that is associated with an Azure virtual machine, you must first ensure that any virtual machine disks have been deleted. If you do not first delete your virtual machine disks, then when you attempt to delete your storage account, you will see an error message similar to:

```
Failed to delete storage account <vm-storage-account-name>. Unable to delete storage account <vm-storage-account-name>: 'Storage account <vm-storage-account-name> has some active image(s) and/or disk(s). Ensure these image(s) and/or disk(s) are removed before deleting this storage account.'
```

If the storage account uses the Classic deployment model, you can remove the virtual machine disk by following these steps in the [Azure portal](#):

1. Navigate to the [classic Azure portal](#).
2. Navigate to the Virtual Machines tab.
3. Click the Disks tab.
4. Select your data disk, then click Delete Disk.
5. To delete disk images, navigate to the Images tab and delete any images that are stored in the account.

For more information, see the [Azure Virtual Machine documentation](#).

Next steps

- [Azure Blob Storage: Cool and Hot tiers](#)
- [Azure Storage replication](#)
- [Configure Azure Storage Connection Strings](#)
- [Transfer data with the AzCopy Command-Line Utility](#)
- Visit the [Azure Storage Team Blog](#).

Get started with Azure Blob storage using .NET

1/17/2017 • 20 min to read • [Edit on GitHub](#)

TIP

Manage Azure Blob Storage resources with Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to [manage Azure Blob Storage resources](#). Using Microsoft Azure Storage Explorer, you can visually create, read, update, and delete blob containers and blobs, as well as manage access to your blobs containers and blobs.

Overview

Azure Blob storage is a service that stores unstructured data in the cloud as objects/blobs. Blob storage can store any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as object storage.

About this tutorial

This tutorial shows how to write .NET code for some common scenarios using Azure Blob storage. Scenarios covered include uploading, listing, downloading, and deleting blobs.

Prerequisites:

- [Microsoft Visual Studio](#)
- [Azure Storage Client Library for .NET](#)
- [Azure Configuration Manager for .NET](#)
- An [Azure storage account](#)

NOTE

We recommend that you use the latest version of the Azure Storage Client Library for .NET to complete this tutorial. The latest version of the library is 7.x, available for download on [Nuget](#). The source for the client library is available on [GitHub](#).

If you are using the storage emulator, note that version 7.x of the client library requires at least version 4.3 of the storage emulator

More samples

For additional examples using Blob storage, see [Getting Started with Azure Blob Storage in .NET](#). You can download the sample application and run it, or browse the code on GitHub.

What is Blob Storage?

Azure Blob storage is a service for storing large amounts of unstructured object data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately.

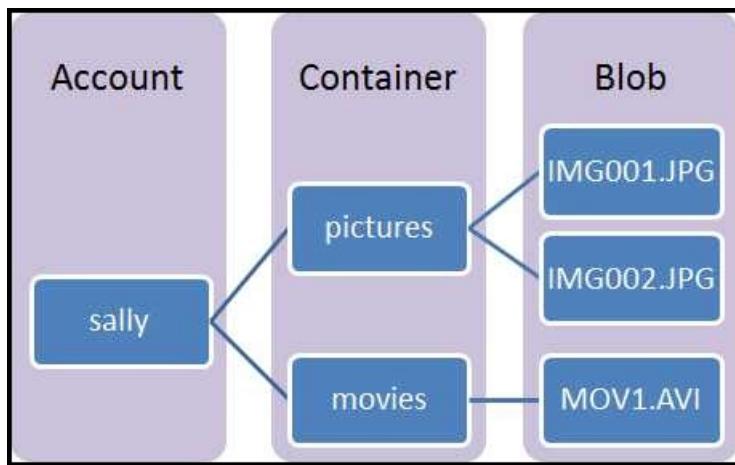
Common uses of Blob storage include:

- 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

Blob service concepts

The Blob service contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. This storage account can be a **General-purpose storage account** or a **Blob storage account** which is specialized for storing objects/blobs. For more information about storage accounts, see [Azure storage account](#).
- **Container:** A container provides a grouping of a set of blobs. All blobs must be in a container. An account can contain an unlimited number of containers. A container can store an unlimited number of blobs. Note that the container name must be lowercase.
- **Blob:** A file of any type and size. Azure Storage offers three types of blobs: block blobs, page blobs, and append blobs.

Block blobs are ideal for storing text or binary files, such as documents and media files. *Append blobs* are similar to block blobs in that they are made up of blocks, but they are optimized for append operations, so they are useful for logging scenarios. A single block blob can contain up to 50,000 blocks of up to 100 MB each, for a total size of slightly more than 4.75 TB (100 MB X 50,000). A single append blob can contain up to 50,000 blocks of up to 4 MB each, for a total size of slightly more than 195 GB (4 MB X 50,000).

Page blobs can be up to 1 TB in size, and are more efficient for frequent read/write operations. Azure Virtual Machines use page blobs as OS and data disks.

For details about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

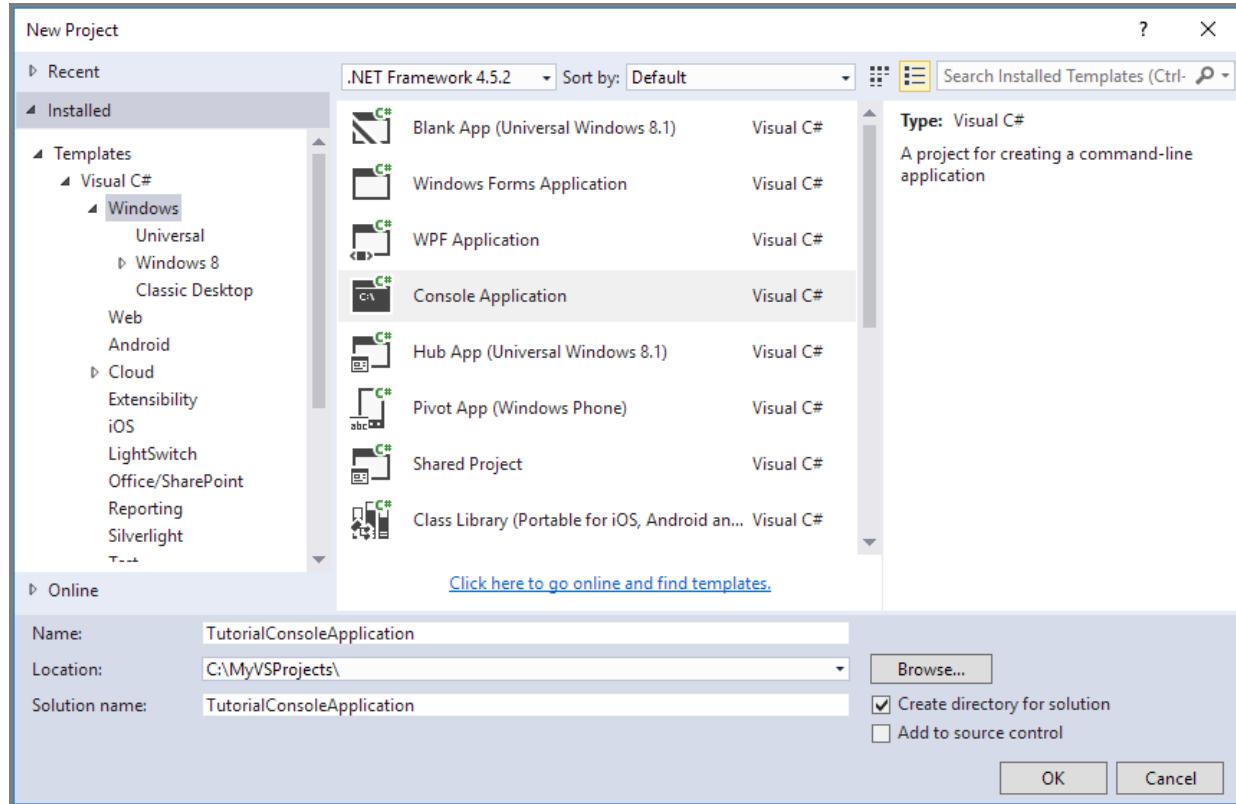
If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Set up your development environment

Next, set up your development environment in Visual Studio so that you are ready to try the code examples provided in this guide.

Create a Windows console application project

In Visual Studio, create a new Windows console application, as shown:



All of the code examples in this tutorial can be added to the **Main()** method in `program.cs` in your console application.

Note that you can use the Azure Storage Client Library from any type of .NET application, including an Azure cloud service, an Azure web app, a desktop application, or a mobile application. In this guide, we use a console application for simplicity.

Use NuGet to install the required packages

There are two packages that you'll need to install to your project to complete this tutorial:

- [Microsoft Azure Storage Client Library for .NET](#): This package provides programmatic access to data resources in your storage account.
- [Microsoft Azure Configuration Manager library for .NET](#): This package provides a class for parsing a connection string from a configuration file, regardless of where your application is running.

You can use NuGet to obtain both packages. Follow these steps:

1. Right-click your project in **Solution Explorer** and choose **Manage NuGet Packages**.
2. Search online for "WindowsAzure.Storage" and click **Install** to install the Storage Client Library and its dependencies.
3. Search online for "ConfigurationManager" and click **Install** to install the Azure Configuration Manager.

NOTE

The Storage Client Library package is also included in the [Azure SDK for .NET](#). However, we recommend that you also install the Storage Client Library from NuGet to ensure that you always have the latest version of the client library.

The ODataLib dependencies in the Storage Client Library for .NET are resolved through the ODataLib (version 5.0.2 and greater) packages available through NuGet, and not through WCF Data Services. The ODataLib libraries can be downloaded directly or referenced by your code project through NuGet. The specific ODataLib packages used by the Storage Client Library are [OData](#), [Edm](#), and [Spatial](#). While these libraries are used by the Azure Table storage classes, they are required dependencies for programming with the Storage Client Library.

Determine your target environment

You have two environment options for running the examples in this guide:

- You can run your code against an Azure Storage account in the cloud.
 - You can run your code against the Azure storage emulator. The storage emulator is a local environment that emulates an Azure Storage account in the cloud. The emulator is a free option for testing and debugging your code while your application is under development. The emulator uses a well-known account and key.
- For more details, see [Use the Azure Storage Emulator for Development and Testing](#)

If you are targeting a storage account in the cloud, copy the primary access key for your storage account from the Azure Portal. For more information, see [View and copy storage access keys](#).

NOTE

You can target the storage emulator to avoid incurring any costs associated with Azure Storage. However, if you do choose to target an Azure storage account in the cloud, costs for performing this tutorial will be negligible.

Configure your storage connection string

The Azure Storage Client Library for .NET supports using a storage connection string to configure endpoints and credentials for accessing storage services. The best way to maintain your storage connection string is in a configuration file.

For more information about connection strings, see [Configure a Connection String to Azure Storage](#).

NOTE

Your storage account key is similar to the root password for your storage account. Always be careful to protect your storage account key. Avoid distributing it to other users, hard-coding it, or saving it in a plain-text file that is accessible to others. Regenerate your key using the Azure Portal if you believe it may have been compromised.

To configure your connection string, open the `app.config` file from Solution Explorer in Visual Studio. Add the contents of the `<appSettings>` element shown below. Replace `account-name` with the name of your storage account, and `account-key` with your account access key:

```
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
  </startup>
  <appSettings>
    <add key="StorageConnectionString" value="DefaultEndpointsProtocol=https;AccountName=account-
name;AccountKey=account-key" />
  </appSettings>
</configuration>
```

For example, your configuration setting will be similar to:

```
<add key="StorageConnectionString"
    value="DefaultEndpointsProtocol=https;AccountName=storagesample;AccountKey=nYV0gln6fT7mvY+rxu2iWAEyzPKITGkhM8
    8J8HUoyofvK7C6fHcZc2kRZp6cKgYRUM74lHI84L50Iau1+9hPjB==" />
```

To target the storage emulator, you can use a shortcut that maps to the well-known account name and key. In that case, your connection string setting will be:

```
<add key="StorageConnectionString" value="UseDevelopmentStorage=true;" />
```

Add namespace declarations

Add the following **using** statements to the top of the `program.cs` file:

```
using Microsoft.Azure; // Namespace for CloudConfigurationManager
using Microsoft.WindowsAzure.Storage; // Namespace for CloudStorageAccount
using Microsoft.WindowsAzure.Storage.Blob; // Namespace for Blob storage types
```

Parse the connection string

The [Microsoft Azure Configuration Manager Library for .NET](#) provides a class for parsing a connection string from a configuration file. The [CloudConfigurationManager](#) class parses configuration settings regardless of whether the client application is running on the desktop, on a mobile device, in an Azure virtual machine, or in an Azure cloud service.

To reference the CloudConfigurationManager package, add the following `using` directive:

```
using Microsoft.Azure; //Namespace for CloudConfigurationManager
```

Here's an example that shows how to retrieve a connection string from a configuration file:

```
// Parse the connection string and return a reference to the storage account.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));
```

Using the Azure Configuration Manager is optional. You can also use an API like the .NET Framework's [ConfigurationManager](#) class.

Create the Blob service client

The [CloudBlobClient](#) class enables you to retrieve containers and blobs stored in Blob storage. Here's one way to create the service client:

```
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
```

Now you are ready to write code that reads data from and writes data to Blob storage.

Create a container

Every blob in Azure storage must reside in a container. The container forms part of the blob name. For example, `mycontainer` is the name of the container in these sample blob URLs:

```
https://storagesample.blob.core.windows.net/mycontainer/blob1.txt
https://storagesample.blob.core.windows.net/mycontainer/photos/myphoto.jpg
```

A container name must be a valid DNS name, conforming to the following naming rules:

1. Container names must start with a letter or number, and can contain only letters, numbers, and the dash (-) character.
2. Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive dashes are not permitted in container names.
3. All letters in a container name must be lowercase.
4. Container names must be from 3 through 63 characters long.

IMPORTANT

Note that the name of a container must always be lowercase. If you include an upper-case letter in a container name, or otherwise violate the container naming rules, you may receive a 400 error (Bad Request).

This example shows how to create a container if it does not already exist:

```
// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the blob client.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Retrieve a reference to a container.
CloudBlobContainer container = blobClient.GetContainerReference("mycontainer");

// Create the container if it doesn't already exist.
container.CreateIfNotExists();
```

By default, the new container is private, meaning that you must specify your storage access key to download blobs from this container. If you want to make the files within the container available to everyone, you can set the container to be public using the following code:

```
container.SetPermissions(
    new BlobContainerPermissions { PublicAccess = BlobContainerPublicAccessType.Blob });
```

Anyone on the Internet can see blobs in a public container, but you can modify or delete them only if you have the appropriate account access key or a shared access signature.

Upload a blob into a container

Azure Blob Storage supports block blobs and page blobs. In the majority of cases, block blob is the recommended type to use.

To upload a file to a block blob, get a container reference and use it to get a block blob reference. Once you have a blob reference, you can upload any stream of data to it by calling the **UploadFromStream** method. This operation will create the blob if it didn't previously exist, or overwrite it if it does exist.

The following example shows how to upload a blob into a container and assumes that the container was already created.

```

// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the blob client.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Retrieve reference to a previously created container.
CloudBlobContainer container = blobClient.GetContainerReference("mycontainer");

// Retrieve reference to a blob named "myblob".
CloudBlockBlob blockBlob = container.GetBlockBlobReference("myblob");

// Create or overwrite the "myblob" blob with contents from a local file.
using (var fileStream = System.IO.File.OpenRead(@"path\myfile"))
{
    blockBlob.UploadFromStream(fileStream);
}

```

List the blobs in a container

To list the blobs in a container, first get a container reference. You can then use the container's **ListBlobs** method to retrieve the blobs and/or directories within it. To access the rich set of properties and methods for a returned **IListBlobItem**, you must cast it to a **CloudBlockBlob**, **CloudPageBlob**, or **CloudBlobDirectory** object. If the type is unknown, you can use a type check to determine which to cast it to. The following code demonstrates how to retrieve and output the URI of each item in the *photos* container:

```

// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the blob client.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Retrieve reference to a previously created container.
CloudBlobContainer container = blobClient.GetContainerReference("photos");

// Loop over items within the container and output the length and URI.
foreach (IListBlobItem item in container.ListBlobs(null, false))
{
    if (item.GetType() == typeof(CloudBlockBlob))
    {
        CloudBlockBlob blob = (CloudBlockBlob)item;

        Console.WriteLine("Block blob of length {0}: {1}", blob.Properties.Length, blob.Uri);
    }
    else if (item.GetType() == typeof(CloudPageBlob))
    {
        CloudPageBlob pageBlob = (CloudPageBlob)item;

        Console.WriteLine("Page blob of length {0}: {1}", pageBlob.Properties.Length, pageBlob.Uri);
    }
    else if (item.GetType() == typeof(CloudBlobDirectory))
    {
        CloudBlobDirectory directory = (CloudBlobDirectory)item;

        Console.WriteLine("Directory: {0}", directory.Uri);
    }
}

```

As shown above, you can name blobs with path information in their names. This creates a virtual directory

structure that you can organize and traverse as you would a traditional file system. Note that the directory structure is virtual only - the only resources available in Blob storage are containers and blobs. However, the storage client library offers a **CloudBlobDirectory** object to refer to a virtual directory and simplify the process of working with blobs that are organized in this way.

For example, consider the following set of block blobs in a container named *photos*:

```
photo1.jpg
2010/architecture/description.txt
2010/architecture/photo3.jpg
2010/architecture/photo4.jpg
2011/architecture/photo5.jpg
2011/architecture/photo6.jpg
2011/architecture/description.txt
2011/photo7.jpg
```

When you call **ListBlobs** on the *photos* container (as in the above sample), a hierarchical listing is returned. It contains both **CloudBlobDirectory** and **CloudBlockBlob** objects, representing the directories and blobs in the container, respectively. The resulting output looks like:

```
Directory: https://<accountname>.blob.core.windows.net/photos/2010/
Directory: https://<accountname>.blob.core.windows.net/photos/2011/
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

Optionally, you can set the **UseFlatBlobListing** parameter of the **ListBlobs** method to **true**. In this case, every blob in the container is returned as a **CloudBlockBlob** object. The call to **ListBlobs** to return a flat listing looks like this:

```
// Loop over items within the container and output the length and URI.
foreach (IListBlobItem item in container.ListBlobs(null, true))
{
    ...
}
```

and the results look like this:

```
Block blob of length 4: https://<accountname>.blob.core.windows.net/photos/2010/architecture/description.txt
Block blob of length 314618: https://<accountname>.blob.core.windows.net/photos/2010/architecture/photo3.jpg
Block blob of length 522713: https://<accountname>.blob.core.windows.net/photos/2010/architecture/photo4.jpg
Block blob of length 4: https://<accountname>.blob.core.windows.net/photos/2011/architecture/description.txt
Block blob of length 419048: https://<accountname>.blob.core.windows.net/photos/2011/architecture/photo5.jpg
Block blob of length 506388: https://<accountname>.blob.core.windows.net/photos/2011/architecture/photo6.jpg
Block blob of length 399751: https://<accountname>.blob.core.windows.net/photos/2011/photo7.jpg
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

Download blobs

To download blobs, first retrieve a blob reference and then call the **DownloadToStream** method. The following example uses the **DownloadToStream** method to transfer the blob contents to a stream object that you can then persist to a local file.

```

// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the blob client.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Retrieve reference to a previously created container.
CloudBlobContainer container = blobClient.GetContainerReference("mycontainer");

// Retrieve reference to a blob named "photo1.jpg".
CloudBlockBlob blockBlob = container.GetBlockBlobReference("photo1.jpg");

// Save blob contents to a file.
using (var fileStream = System.IO.File.OpenWrite(@"path\myfile"))
{
    blockBlob.DownloadToStream(fileStream);
}

```

You can also use the **DownloadToStream** method to download the contents of a blob as a text string.

```

// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the blob client.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Retrieve reference to a previously created container.
CloudBlobContainer container = blobClient.GetContainerReference("mycontainer");

// Retrieve reference to a blob named "myblob.txt"
CloudBlockBlob blockBlob2 = container.GetBlockBlobReference("myblob.txt");

string text;
using (var memoryStream = new MemoryStream())
{
    blockBlob2.DownloadToStream(memoryStream);
    text = System.Text.Encoding.UTF8.GetString(memoryStream.ToArray());
}

```

Delete blobs

To delete a blob, first get a blob reference and then call the **Delete** method on it.

```

// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the blob client.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Retrieve reference to a previously created container.
CloudBlobContainer container = blobClient.GetContainerReference("mycontainer");

// Retrieve reference to a blob named "myblob.txt".
CloudBlockBlob blockBlob = container.GetBlockBlobReference("myblob.txt");

// Delete the blob.
blockBlob.Delete();

```

List blobs in pages asynchronously

If you are listing a large number of blobs, or you want to control the number of results you return in one listing operation, you can list blobs in pages of results. This example shows how to return results in pages asynchronously, so that execution is not blocked while waiting to return a large set of results.

This example shows a flat blob listing, but you can also perform a hierarchical listing, by setting the `useFlatBlobListing` parameter of the **ListBlobsSegmentedAsync** method to *false*.

Because the sample method calls an asynchronous method, it must be prefaced with the `async` keyword, and it must return a **Task** object. The `await` keyword specified for the **ListBlobsSegmentedAsync** method suspends execution of the sample method until the listing task completes.

```
async public static Task ListBlobsSegmentedInFlatListing(CloudBlobContainer container)
{
    //List blobs to the console window, with paging.
    Console.WriteLine("List blobs in pages:");

    int i = 0;
    BlobContinuationToken continuationToken = null;
    BlobResultSegment resultSegment = null;

    //Call ListBlobsSegmentedAsync and enumerate the result segment returned, while the continuation token is non-null.
    //When the continuation token is null, the last page has been returned and execution can exit the loop.
    do
    {
        //This overload allows control of the page size. You can return all remaining results by passing null for the maxResults parameter,
        //or by calling a different overload.
        resultSegment = await container.ListBlobsSegmentedAsync("", true, BlobListingDetails.All, 10,
continuationToken, null, null);
        if (resultSegment.Results.Count<IListBlobItem>() > 0) { Console.WriteLine("Page {0}:", ++i); }
        foreach (var blobItem in resultSegment.Results)
        {
            Console.WriteLine("\t{0}", blobItem.StorageUri.PrimaryUri);
        }
        Console.WriteLine();

        //Get the continuation token.
        continuationToken = resultSegment.ContinuationToken;
    }
    while (continuationToken != null);
}
```

Writing to an append blob

An append blob is a new type of blob, introduced with version 5.x of the Azure storage client library for .NET. An append blob is optimized for append operations, such as logging. Like a block blob, an append blob is comprised of blocks, but when you add a new block to an append blob, it is always appended to the end of the blob. You cannot update or delete an existing block in an append blob. The block IDs for an append blob are not exposed as they are for a block blob.

Each block in an append blob can be a different size, up to a maximum of 4 MB, and an append blob can include a maximum of 50,000 blocks. The maximum size of an append blob is therefore slightly more than 195 GB (4 MB X 50,000 blocks).

The example below creates a new append blob and appends some data to it, simulating a simple logging operation.

```

//Parse the connection string for the storage account.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    Microsoft.Azure.CloudConfigurationManager.GetSetting("StorageConnectionString"));

//Create service client for credentialled access to the Blob service.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

//Get a reference to a container.
CloudBlobContainer container = blobClient.GetContainerReference("my-append-blobs");

//Create the container if it does not already exist.
container.CreateIfNotExists();

//Get a reference to an append blob.
CloudAppendBlob appendBlob = container.GetAppendBlobReference("append-blob.log");

//Create the append blob. Note that if the blob already exists, the CreateOrReplace() method will overwrite
it.
//You can check whether the blob exists to avoid overwriting it by using CloudAppendBlob.Exists().
appendBlob.CreateOrReplace();

int numBlocks = 10;

//Generate an array of random bytes.
Random rnd = new Random();
byte[] bytes = new byte[numBlocks];
rnd.NextBytes(bytes);

//Simulate a logging operation by writing text data and byte data to the end of the append blob.
for (int i = 0; i < numBlocks; i++)
{
    appendBlob.AppendText(String.Format("Timestamp: {0:u} \tLog Entry: {1}{2}",
        DateTime.UtcNow, bytes[i], Environment.NewLine));
}

//Read the append blob to the console window.
Console.WriteLine(appendBlob.DownloadText());

```

See [Understanding Block Blobs, Page Blobs, and Append Blobs](#) for more information about the differences between the three types of blobs.

Managing security for blobs

By default, Azure Storage keeps your data secure by limiting access to the account owner, who is in possession of the account access keys. When you need to share blob data in your storage account, it is important to do so without compromising the security of your account access keys. Additionally, you can encrypt blob data to ensure that it is secure going over the wire and in Azure Storage.

IMPORTANT

Your storage account key is similar to the root password for your storage account. Always be careful to protect your account key. Avoid distributing it to other users, hard-coding it, or saving it in a plain-text file that is accessible to others. Regenerate your account key using the Azure Portal if you believe it may have been compromised. To learn how to regenerate your account key, see [How to create, manage, or delete a storage account in the Azure Portal](#).

Controlling access to blob data

By default, the blob data in your storage account is accessible only to storage account owner. Authenticating requests against Blob storage requires the account access key by default. However, you may wish to make certain blob data available to other users. You have two options:

- **Anonymous access:** You can make a container or its blobs publicly available for anonymous access. See

[Manage anonymous read access to containers and blobs](#) for more information.

- **Shared access signatures:** You can provide clients with a shared access signature (SAS), which provides delegated access to a resource in your storage account, with permissions that you specify and over an interval that you specify. See [Using Shared Access Signatures \(SAS\)](#) for more information.

Encrypting blob data

Azure Storage supports encrypting blob data both at the client and on the server:

- **Client-side encryption:** The Storage Client Library for .NET supports encrypting data within client applications before uploading to Azure Storage, and decrypting data while downloading to the client. The library also supports integration with Azure Key Vault for storage account key management. See [Client-Side Encryption with .NET for Microsoft Azure Storage](#) for more information. Also see [Tutorial: Encrypt and decrypt blobs in Microsoft Azure Storage using Azure Key Vault](#).
- **Server-side encryption:** Azure Storage now supports server-side encryption. See [Azure Storage Service Encryption for Data at Rest \(Preview\)](#).

Next steps

Now that you've learned the basics of Blob storage, follow these links to learn more.

Microsoft Azure Storage Explorer

- [Microsoft Azure Storage Explorer \(MASE\)](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Blob storage samples

- [Getting Started with Azure Blob Storage in .NET](#)

Blob storage reference

- [Storage Client Library for .NET reference](#)
- [REST API reference](#)

Conceptual guides

- [Transfer data with the AzCopy command-line utility](#)
- [Get started with File storage for .NET](#)
- [How to use Azure blob storage with the WebJobs SDK](#)

How to use Blob storage from Java

1/17/2017 • 11 min to read • [Edit on GitHub](#)

TIP

Manage Azure Blob Storage resources with Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to [manage Azure Blob Storage resources](#). Using Microsoft Azure Storage Explorer, you can visually create, read, update, and delete blob containers and blobs, as well as manage access to your blobs containers and blobs.

Overview

Azure Blob storage is a service that stores unstructured data in the cloud as objects/blobs. Blob storage can store any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as object storage.

This article will show you how to perform common scenarios using the Microsoft Azure Blob storage. The samples are written in Java and use the [Azure Storage SDK for Java](#). The scenarios covered include **uploading**, **listing**, **downloading**, and **deleting** blobs. For more information on blobs, see the [Next Steps](#) section.

NOTE

An SDK is available for developers who are using Azure Storage on Android devices. For more information, see the [Azure Storage SDK for Android](#).

What is Blob Storage?

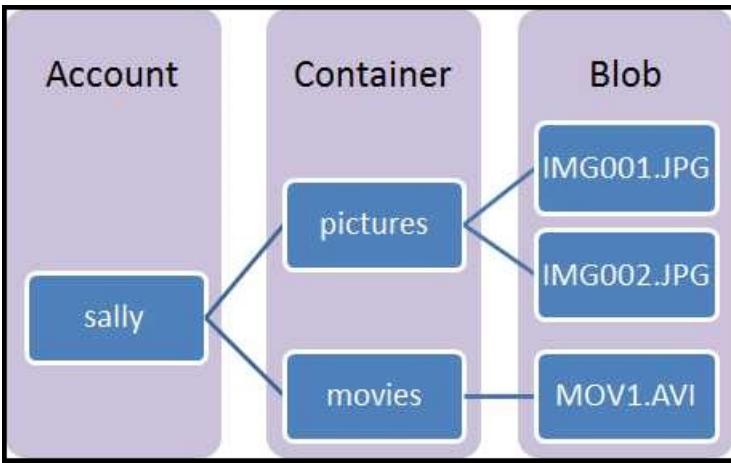
Azure Blob storage is a service for storing large amounts of unstructured object data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately.

Common uses of Blob storage include:

- 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

Blob service concepts

The Blob service contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. This storage account can be a **General-purpose storage account** or a **Blob storage account** which is specialized for storing objects/blobs. For more information about storage accounts, see [Azure storage account](#).
- **Container:** A container provides a grouping of a set of blobs. All blobs must be in a container. An account can contain an unlimited number of containers. A container can store an unlimited number of blobs. Note that the container name must be lowercase.
- **Blob:** A file of any type and size. Azure Storage offers three types of blobs: block blobs, page blobs, and append blobs.

Block blobs are ideal for storing text or binary files, such as documents and media files. *Append blobs* are similar to block blobs in that they are made up of blocks, but they are optimized for append operations, so they are useful for logging scenarios. A single block blob can contain up to 50,000 blocks of up to 100 MB each, for a total size of slightly more than 4.75 TB (100 MB X 50,000). A single append blob can contain up to 50,000 blocks of up to 4 MB each, for a total size of slightly more than 195 GB (4 MB X 50,000).

Page blobs can be up to 1 TB in size, and are more efficient for frequent read/write operations. Azure Virtual Machines use page blobs as OS and data disks.

For details about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a Java application

In this article, you will use storage features which can be run within a Java application locally, or in code running within a web role or worker role in Azure.

To do so, you will need to install the Java Development Kit (JDK) and create an Azure Storage account in your Azure subscription. Once you have done so, you will need to verify that your development system meets the minimum requirements and dependencies which are listed in the [Azure Storage SDK for Java](#) repository on GitHub. If your system meets those requirements, you can follow the instructions for downloading and installing

the Azure Storage Libraries for Java on your system from that repository. Once you have completed those tasks, you will be able to create a Java application which uses the examples in this article.

Configure your application to access Blob storage

Add the following import statements to the top of the Java file where you want to use the Azure Storage APIs to access blobs.

```
// Include the following imports to use blob APIs.  
import com.microsoft.azure.storage.*;  
import com.microsoft.azure.storage.blob.*;
```

Set up an Azure Storage connection string

An Azure Storage client uses a storage connection string to store endpoints and credentials for accessing data management services. When running in a client application, you must provide the storage connection string in the following format, using the name of your storage account and the Primary access key for the storage account listed in the [Azure portal](#) for the *AccountName* and *AccountKey* values. The following example shows how you can declare a static field to hold the connection string.

```
// Define the connection-string with your values  
public static final String storageConnectionString =  
    "DefaultEndpointsProtocol=http;" +  
    "AccountName=your_storage_account;" +  
    "AccountKey=your_storage_account_key";
```

In an application running within a role in Microsoft Azure, this string can be stored in the service configuration file, *ServiceConfiguration.cscfg*, and can be accessed with a call to the **RoleEnvironment.getConfigurationSettings** method. The following example gets the connection string from a **Setting** element named *StorageConnectionString* in the service configuration file.

```
// Retrieve storage account from connection-string.  
String storageConnectionString =  
    RoleEnvironment.getConfigurationSettings().get("StorageConnectionString");
```

The following samples assume that you have used one of these two methods to get the storage connection string.

Create a container

A **CloudBlobClient** object lets you get reference objects for containers and blobs. The following code creates a **CloudBlobClient** object.

NOTE

There are additional ways to create **CloudStorageAccount** objects; for more information, see **CloudStorageAccount** in the [Azure Storage Client SDK Reference](#).

Every blob in Azure storage must reside in a container. The container forms part of the blob name. For example, `mycontainer` is the name of the container in these sample blob URLs:

```
https://storagesample.blob.core.windows.net/mycontainer/blob1.txt  
https://storagesample.blob.core.windows.net/mycontainer/photos/myphoto.jpg
```

A container name must be a valid DNS name, conforming to the following naming rules:

1. Container names must start with a letter or number, and can contain only letters, numbers, and the dash (-) character.
2. Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive dashes are not permitted in container names.
3. All letters in a container name must be lowercase.
4. Container names must be from 3 through 63 characters long.

IMPORTANT

Note that the name of a container must always be lowercase. If you include an upper-case letter in a container name, or otherwise violate the container naming rules, you may receive a 400 error (Bad Request).

Use the **CloudBlobClient** object to get a reference to the container you want to use. You can create the container if it doesn't exist with the **createIfNotExists** method, which will otherwise return the existing container. By default, the new container is private, so you must specify your storage access key (as you did earlier) to download blobs from this container.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount = CloudStorageAccount.parse(storageConnectionString);

    // Create the blob client.
    CloudBlobClient blobClient = storageAccount.createCloudBlobClient();

    // Get a reference to a container.
    // The container name must be lower case
    CloudBlobContainer container = blobClient.getContainerReference("mycontainer");

    // Create the container if it does not exist.
    container.createIfNotExists();
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Optional: Configure a container for public access

A container's permissions are configured for private access by default, but you can easily configure a container's permissions to allow public, read-only access for all users on the Internet:

```
// Create a permissions object.
BlobContainerPermissions containerPermissions = new BlobContainerPermissions();

// Include public access in the permissions object.
containerPermissions.setPublicAccess(BlobContainerPublicAccessType.CONTAINER);

// Set the permissions on the container.
container.uploadPermissions(containerPermissions);
```

Upload a blob into a container

To upload a file to a blob, get a container reference and use it to get a blob reference. Once you have a blob reference, you can upload any stream by calling `upload` on the blob reference. This operation will create the blob if it doesn't exist, or overwrite it if it does. The following code sample shows this, and assumes that the container has already been created.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount = CloudStorageAccount.parse(storageConnectionString);

    // Create the blob client.
    CloudBlobClient blobClient = storageAccount.createCloudBlobClient();

    // Retrieve reference to a previously created container.
    CloudBlobContainer container = blobClient.getContainerReference("mycontainer");

    // Define the path to a local file.
    final String filePath = "C:\\\\myimages\\\\myimage.jpg";

    // Create or overwrite the "myimage.jpg" blob with contents from a local file.
    CloudBlockBlob blob = container.getBlockBlobReference("myimage.jpg");
    File source = new File(filePath);
    blob.upload(new FileInputStream(source), source.length());
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

List the blobs in a container

To list the blobs in a container, first get a container reference like you did to upload a blob. You can use the container's **listBlobs** method with a **for** loop. The following code outputs the Uri of each blob in a container to the console.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount = CloudStorageAccount.parse(storageConnectionString);

    // Create the blob client.
    CloudBlobClient blobClient = storageAccount.createCloudBlobClient();

    // Retrieve reference to a previously created container.
    CloudBlobContainer container = blobClient.getContainerReference("mycontainer");

    // Loop over blobs within the container and output the URI to each of them.
    for (ListBlobItem blobItem : container.listBlobs()) {
        System.out.println(blobItem.getUri());
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

Note that you can name blobs with path information in their names. This creates a virtual directory structure that you can organize and traverse as you would a traditional file system. Note that the directory structure is virtual only - the only resources available in Blob storage are containers and blobs. However, the client library offers a **CloudBlobDirectory** object to refer to a virtual directory and simplify the process of working with blobs that are organized in this way.

For example, you could have a container named "photos", in which you might upload blobs named "rootphoto1", "2010/photo1", "2010/photo2", and "2011/photo1". This would create the virtual directories "2010" and "2011" within the "photos" container. When you call **listBlobs** on the "photos" container, the collection returned will

contain **CloudBlobDirectory** and **CloudBlob** objects representing the directories and blobs contained at the top level. In this case, directories "2010" and "2011", as well as photo "rootphoto1" would be returned. You can use the **instanceof** operator to distinguish these objects.

Optionally, you can pass in parameters to the **listBlobs** method with the **useFlatBlobListing** parameter set to true. This will result in every blob being returned, regardless of directory. For more information, see [CloudBlobContainer.listBlobs](#) in the [Azure Storage Client SDK Reference](#).

Download a blob

To download blobs, follow the same steps as you did for uploading a blob in order to get a blob reference. In the uploading example, you called upload on the blob object. In the following example, call download to transfer the blob contents to a stream object such as a **FileOutputStream** that you can use to persist the blob to a local file.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount = CloudStorageAccount.parse(storageConnectionString);

    // Create the blob client.
    CloudBlobClient blobClient = storageAccount.createCloudBlobClient();

    // Retrieve reference to a previously created container.
    CloudBlobContainer container = blobClient.getContainerReference("mycontainer");

    // Loop through each blob item in the container.
    for (ListBlobItem blobItem : container.listBlobs()) {
        // If the item is a blob, not a virtual directory.
        if (blobItem instanceof CloudBlob) {
            // Download the item and save it to a file with the same name.
            CloudBlob blob = (CloudBlob) blobItem;
            blob.download(new FileOutputStream("C:\\\\mydownloads\\\\" + blob.getName()));
        }
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Delete a blob

To delete a blob, get a blob reference, and call **deleteIfExists**.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount = CloudStorageAccount.parse(storageConnectionString);

    // Create the blob client.
    CloudBlobClient blobClient = storageAccount.createCloudBlobClient();

    // Retrieve reference to a previously created container.
    CloudBlobContainer container = blobClient.getContainerReference("mycontainer");

    // Retrieve reference to a blob named "myimage.jpg".
    CloudBlockBlob blob = container.getBlockBlobReference("myimage.jpg");

    // Delete the blob.
    blob.deleteIfExists();
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

Delete a blob container

Finally, to delete a blob container, get a blob container reference, and call **deleteIfExists**.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount = CloudStorageAccount.parse(storageConnectionString);

    // Create the blob client.
    CloudBlobClient blobClient = storageAccount.createCloudBlobClient();

    // Retrieve reference to a previously created container.
    CloudBlobContainer container = blobClient.getContainerReference("mycontainer");

    // Delete the blob container.
    container.deleteIfExists();
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

Next steps

Now that you've learned the basics of Blob storage, follow these links to learn about more complex storage tasks.

- [Azure Storage SDK for Java](#)
- [Azure Storage Client SDK Reference](#)
- [Azure Storage REST API](#)
- [Azure Storage Team Blog](#)

For more information, see also the [Java Developer Center](#).

How to use Blob storage from Node.js

1/17/2017 • 15 min to read • [Edit on GitHub](#)

TIP

Manage Azure Blob Storage resources with Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to [manage Azure Blob Storage resources](#). Using Microsoft Azure Storage Explorer, you can visually create, read, update, and delete blob containers and blobs, as well as manage access to your blobs containers and blobs.

Overview

This article shows you how to perform common scenarios using Blob storage. The samples are written via the Node.js API. The scenarios covered include how to upload, list, download, and delete blobs.

What is Blob Storage?

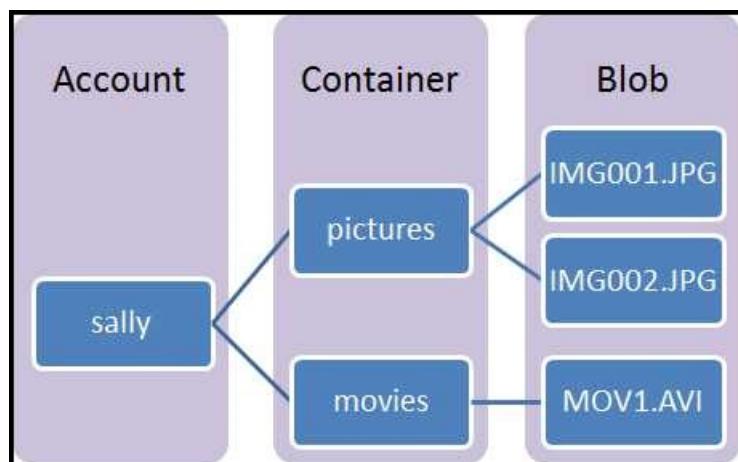
Azure Blob storage is a service for storing large amounts of unstructured object data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately.

Common uses of Blob storage include:

- 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

Blob service concepts

The Blob service contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. This storage account can be a **General-purpose storage account** or a **Blob storage account** which is specialized for storing objects/blobs. For more information about storage accounts, see [Azure storage account](#).

- **Container:** A container provides a grouping of a set of blobs. All blobs must be in a container. An account can contain an unlimited number of containers. A container can store an unlimited number of blobs. Note that the container name must be lowercase.
- **Blob:** A file of any type and size. Azure Storage offers three types of blobs: block blobs, page blobs, and append blobs.

Block blobs are ideal for storing text or binary files, such as documents and media files. *Append blobs* are similar to block blobs in that they are made up of blocks, but they are optimized for append operations, so they are useful for logging scenarios. A single block blob can contain up to 50,000 blocks of up to 100 MB each, for a total size of slightly more than 4.75 TB (100 MB X 50,000). A single append blob can contain up to 50,000 blocks of up to 4 MB each, for a total size of slightly more than 195 GB (4 MB X 50,000).

Page blobs can be up to 1 TB in size, and are more efficient for frequent read/write operations. Azure Virtual Machines use page blobs as OS and data disks.

For details about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a Node.js application

For instructions on how to create a Node.js application, see [Create a Node.js web app in Azure App Service](#), [Build and deploy a Node.js application to an Azure Cloud Service](#) -- using Windows PowerShell, or [Build and deploy a Node.js web app to Azure using Web Matrix](#).

Configure your application to access storage

To use Azure storage, you need the Azure Storage SDK for Node.js, which includes a set of convenience libraries that communicate with the storage REST services.

Use Node Package Manager (NPM) to obtain the package

1. Use a command-line interface such as **PowerShell** (Windows), **Terminal** (Mac), or **Bash** (Unix), to navigate to the folder where you created your sample application.
2. Type **npm install azure-storage** in the command window. Output from the command is similar to the following code example.

```
azure-storage@0.5.0 node_modules\azure-storage +-- extend@1.2.1 +-- xmlbuilder@0.4.3 +--  
mime@1.2.11 +-- node-uuid@1.4.3 +-- validator@3.22.2 +-- underscore@1.4.4 +-- readable-  
stream@1.0.33 (string_decoder@0.10.31, isarray@0.0.1, inherits@2.0.1, core-util-is@1.0.1) +--  
xml2js@0.2.7 (sax@0.5.2) +-- request@2.57.0 (caseless@0.10.0, aws-sign2@0.5.0, forever-agent@0.6.1,  
stringstream@0.0.4, oauth-sign@0.8.0, tunnel-agent@0.4.1, isstream@0.1.2, json-stringify-safe@5.0.1,  
bl@0.9.4, combined-stream@1.0.5, qs@3.1.0, mime-types@2.0.14, form-data@0.2.0, http-  
signature@0.11.0, tough-cookie@2.0.0, hawk@2.3.1, har-validator@1.8.0)
```

3. You can manually run the **ls** command to verify that a **node_modules** folder was created. Inside that

folder, find the **azure-storage** package, which contains the libraries that you need to access storage.

Import the package

Using Notepad or another text editor, add the following to the top of the **server.js** file of the application where you intend to use storage:

```
var azure = require('azure-storage');
```

Set up an Azure Storage connection

The Azure module will read the environment variables `AZURE_STORAGE_ACCOUNT` and `AZURE_STORAGE_ACCESS_KEY`, or `AZURE_STORAGE_CONNECTION_STRING`, for information required to connect to your Azure storage account. If these environment variables are not set, you must specify the account information when calling **createBlobService**.

For an example of setting the environment variables in the [Azure portal](#) for an Azure web app, see [Node.js web app using the Azure Table Service](#).

Create a container

The **BlobService** object lets you work with containers and blobs. The following code creates a **BlobService** object. Add the following near the top of **server.js**:

```
var blobSvc = azure.createBlobService();
```

NOTE

You can access a blob anonymously by using **createBlobServiceAnonymous** and providing the host address. For example, use `var blobSvc = azure.createBlobServiceAnonymous('https://myblob.blob.core.windows.net/');`.

Every blob in Azure storage must reside in a container. The container forms part of the blob name. For example, `mycontainer` is the name of the container in these sample blob URLs:

```
https://storagesample.blob.core.windows.net/mycontainer/blob1.txt  
https://storagesample.blob.core.windows.net/mycontainer/photos/myphoto.jpg
```

A container name must be a valid DNS name, conforming to the following naming rules:

1. Container names must start with a letter or number, and can contain only letters, numbers, and the dash (-) character.
2. Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive dashes are not permitted in container names.
3. All letters in a container name must be lowercase.
4. Container names must be from 3 through 63 characters long.

IMPORTANT

Note that the name of a container must always be lowercase. If you include an upper-case letter in a container name, or otherwise violate the container naming rules, you may receive a 400 error (Bad Request).

To create a new container, use **createContainerIfNotExists**. The following code example creates a new container named 'mycontainer':

```
blobSvc.createContainerIfNotExists('mycontainer', function(error, result, response){
  if(!error){
    // Container exists and is private
  }
});
```

If the container is newly created, `result.created` is true. If the container already exists, `result.created` is false. `response` contains information about the operation, including the ETag information for the container.

Container security

By default, new containers are private and cannot be accessed anonymously. To make the container public so that you can access it anonymously, you can set the container's access level to **blob** or **container**.

- **blob** - allows anonymous read access to blob content and metadata within this container, but not to container metadata such as listing all blobs within a container
- **container** - allows anonymous read access to blob content and metadata as well as container metadata

The following code example demonstrates setting the access level to **blob**:

```
blobSvc.createContainerIfNotExists('mycontainer', {publicAccessLevel : 'blob'}, function(error, result,
response){
  if(!error){
    // Container exists and allows
    // anonymous read access to blob
    // content and metadata within this container
  }
});
```

Alternatively, you can modify the access level of a container by using **setContainerAcl** to specify the access level. The following code example changes the access level to container:

```
blobSvc.setContainerAcl('mycontainer', null /* signedIdentifiers */, {publicAccessLevel : 'container'} /* publicAccessLevel*/, function(error, result, response){
  if(!error){
    // Container access level set to 'container'
  }
});
```

The result contains information about the operation, including the current **ETag** for the container.

Filters

You can apply optional filtering operations to operations performed using **BlobService**. Filtering operations can include logging, automatically retrying, etc. Filters are objects that implement a method with the signature:

```
function handle (requestOptions, next)
```

After doing its preprocessing on the request options, the method needs to call "next", passing a callback with the following signature:

```
function (returnObject, finalCallback, next)
```

In this callback, and after processing the `returnObject` (the response from the request to the server), the callback needs to either invoke `next` if it exists to continue processing other filters or simply invoke `finalCallback` to end the service invocation.

Two filters that implement retry logic are included with the Azure SDK for Node.js, **ExponentialRetryPolicyFilter** and **LinearRetryPolicyFilter**. The following creates a **BlobService** object that

uses the **ExponentialRetryPolicyFilter**:

```
var retryOperations = new azure.ExponentialRetryPolicyFilter();
var blobSvc = azure.createBlobService().withFilter(retryOperations);
```

Upload a blob into a container

There are three types of blobs: block blobs, page blobs and append blobs. Block blobs allow you to more efficiently upload large data. Append blobs are optimized for append operations. Page blobs are optimized for read/write operations. For more information, see [Understanding Block Blobs, Append Blobs, and Page Blobs](#).

Block blobs

To upload data to a block blob, use the following:

- **createBlockBlobFromLocalFile** - creates a new block blob and uploads the contents of a file
- **createBlockBlobFromStream** - creates a new block blob and uploads the contents of a stream
- **createBlockBlobFromText** - creates a new block blob and uploads the contents of a string
- **createWriteStreamToBlockBlob** - provides a write stream to a block blob

The following code example uploads the contents of the **test.txt** file into **myblob**.

```
blobSvc.createBlockBlobFromLocalFile('mycontainer', 'myblob', 'test.txt', function(error, result, response){
  if(!error){
    // file uploaded
  }
});
```

The **result** returned by these methods contains information on the operation, such as the **ETag** of the blob.

Append blobs

To upload data to a new append blob, use the following:

- **createAppendBlobFromLocalFile** - creates a new append blob and uploads the contents of a file
- **createAppendBlobFromStream** - creates a new append blob and uploads the contents of a stream
- **createAppendBlobFromText** - creates a new append blob and uploads the contents of a string
- **createWriteStreamToNewAppendBlob** - creates a new append blob and then provides a stream to write to it

The following code example uploads the contents of the **test.txt** file into **myappendblob**.

```
blobSvc.createAppendBlobFromLocalFile('mycontainer', 'myappendblob', 'test.txt', function(error, result,
response){
  if(!error){
    // file uploaded
  }
});
```

To append a block to an existing append blob, use the following:

- **appendFromLocalFile** - append the contents of a file to an existing append blob
- **appendFromStream** - append the contents of a stream to an existing append blob
- **appendFromText** - append the contents of a string to an existing append blob
- **appendBlockFromStream** - append the contents of a stream to an existing append blob
- **appendBlockFromText** - append the contents of a string to an existing append blob

NOTE

appendFromXXX APIs will do some client-side validation to fail fast to avoid unnecessary server calls. appendBlockFromXXX won't.

The following code example uploads the contents of the **test.txt** file into **myappendblob**.

```
blobSvc.appendFromText('mycontainer', 'myappendblob', 'text to be appended', function(error, result, response){  
    if(!error){  
        // text appended  
    }  
});
```

Page blobs

To upload data to a page blob, use the following:

- **createPageBlob** - creates a new page blob of a specific length
- **createPageBlobFromLocalFile** - creates a new page blob and uploads the contents of a file
- **createPageBlobFromStream** - creates a new page blob and uploads the contents of a stream
- **createWriteStreamToExistingPageBlob** - provides a write stream to an existing page blob
- **createWriteStreamToNewPageBlob** - creates a new page blob and then provides a stream to write to it

The following code example uploads the contents of the **test.txt** file into **mypageblob**.

```
blobSvc.createPageBlobFromLocalFile('mycontainer', 'mypageblob', 'test.txt', function(error, result, response){  
    if(!error){  
        // file uploaded  
    }  
});
```

NOTE

Page blobs consist of 512-byte 'pages'. You will receive an error when uploading data with a size that is not a multiple of 512.

List the blobs in a container

To list the blobs in a container, use the **listBlobsSegmented** method. If you'd like to return blobs with a specific prefix, use **listBlobsSegmentedWithPrefix**.

```
blobSvc.listBlobsSegmented('mycontainer', null, function(error, result, response){  
    if(!error){  
        // result.entries contains the entries  
        // If not all blobs were returned, result.continuationToken has the continuation token.  
    }  
});
```

The `result` contains an `entries` collection, which is an array of objects that describe each blob. If all blobs cannot be returned, the `result` also provides a `continuationToken`, which you may use as the second parameter to retrieve additional entries.

Download blobs

To download data from a blob, use the following:

- **getBlobToLocalFile** - writes the blob contents to file
- **getBlobToStream** - writes the blob contents to a stream
- **getBlobToText** - writes the blob contents to a string
- **createReadStream** - provides a stream to read from the blob

The following code example demonstrates using **getBlobToStream** to download the contents of the **myblob** blob and store it to the **output.txt** file by using a stream:

```
var fs = require('fs');
blobSvc.getBlobToStream('mycontainer', 'myblob', fs.createWriteStream('output.txt'), function(error, result, response){
  if(!error){
    // blob retrieved
  }
});
```

The **result** contains information about the blob, including **ETag** information.

Delete a blob

Finally, to delete a blob, call **deleteBlob**. The following code example deletes the blob named **myblob**.

```
blobSvc.deleteBlob(containerName, 'myblob', function(error, response){
  if(!error){
    // Blob has been deleted
  }
});
```

Concurrent access

To support concurrent access to a blob from multiple clients or multiple process instances, you can use **ETags** or **leases**.

- **Etag** - provides a way to detect that the blob or container has been modified by another process
- **Lease** - provides a way to obtain exclusive, renewable, write or delete access to a blob for a period of time

ETag

Use ETags if you need to allow multiple clients or instances to write to the block Blob or page Blob simultaneously. The ETag allows you to determine if the container or blob was modified since you initially read or created it, which allows you to avoid overwriting changes committed by another client or process.

You can set ETag conditions by using the optional **options.accessConditions** parameter. The following code example only uploads the **test.txt** file if the blob already exists and has the ETag value contained by **etagToMatch**.

```
blobSvc.createBlockBlobFromLocalFile('mycontainer', 'myblob', 'test.txt', { accessConditions: { EtagMatch: etagToMatch} }, function(error, result, response){
  if(!error){
    // file uploaded
  }
});
```

When you're using ETags, the general pattern is:

1. Obtain the ETag as the result of a create, list, or get operation.
2. Perform an action, checking that the ETag value has not been modified.

If the value was modified, this indicates that another client or instance modified the blob or container since you

obtained the ETag value.

Lease

You can acquire a new lease by using the **acquireLease** method, specifying the blob or container that you wish to obtain a lease on. For example, the following code acquires a lease on **myblob**.

```
blobSvc.acquireLease('mycontainer', 'myblob', function(error, result, response){
  if(!error) {
    console.log('leaseId: ' + result.id);
  }
});
```

Subsequent operations on **myblob** must provide the `options.leaseId` parameter. The lease ID is returned as `result.id` from **acquireLease**.

NOTE

By default, the lease duration is infinite. You can specify a non-infinite duration (between 15 and 60 seconds) by providing the `options.leaseDuration` parameter.

To remove a lease, use **releaseLease**. To break a lease, but prevent others from obtaining a new lease until the original duration has expired, use **breakLease**.

Work with shared access signatures

Shared access signatures (SAS) are a secure way to provide granular access to blobs and containers without providing your storage account name or keys. Shared access signatures are often used to provide limited access to your data, such as allowing a mobile app to access blobs.

NOTE

While you can also allow anonymous access to blobs, shared access signatures allow you to provide more controlled access, as you must generate the SAS.

A trusted application such as a cloud-based service generates shared access signatures using the **generateSharedAccessSignature** of the **BlobService**, and provides it to an untrusted or semi-trusted application such as a mobile app. Shared access signatures are generated using a policy, which describes the start and end dates during which the shared access signatures are valid, as well as the access level granted to the shared access signatures holder.

The following code example generates a new shared access policy that allows the shared access signatures holder to perform read operations on the **myblob** blob, and expires 100 minutes after the time it is created.

```

var startDate = new Date();
var expiryDate = new Date(startDate);
expiryDate.setMinutes(startDate.getMinutes() + 100);
startDate.setMinutes(startDate.getMinutes() - 100);

var sharedAccessPolicy = {
    AccessPolicy: {
        Permissions: azure.BlobUtilities.SharedAccessPermissions.READ,
        Start: startDate,
        Expiry: expiryDate
    },
};

var blobSAS = blobSvc.generateSharedAccessSignature('mycontainer', 'myblob', sharedAccessPolicy);
var host = blobSvc.host;

```

Note that the host information must be provided also, as it is required when the shared access signatures holder attempts to access the container.

The client application then uses shared access signatures with **BlobServiceWithSAS** to perform operations against the blob. The following gets information about **myblob**.

```

var sharedBlobSvc = azure.createBlobServiceWithSas(host, blobSAS);
sharedBlobSvc.getBlobProperties('mycontainer', 'myblob', function (error, result, response) {
    if(!error) {
        // retrieved info
    }
});

```

Since the shared access signatures were generated with read-only access, if an attempt is made to modify the blob, an error will be returned.

Access control lists

You can also use an access control list (ACL) to set the access policy for SAS. This is useful if you wish to allow multiple clients to access a container but provide different access policies for each client.

An ACL is implemented using an array of access policies, with an ID associated with each policy. The following code example defines two policies, one for 'user1' and one for 'user2':

```

var sharedAccessPolicy = {
    user1: {
        Permissions: azure.BlobUtilities.SharedAccessPermissions.READ,
        Start: startDate,
        Expiry: expiryDate
    },
    user2: {
        Permissions: azure.BlobUtilities.SharedAccessPermissions.WRITE,
        Start: startDate,
        Expiry: expiryDate
    }
};

```

The following code example gets the current ACL for **mycontainer**, and then adds the new policies using **setBlobAcl**. This approach allows:

```
var extend = require('extend');
blobSvc.getBlobAcl('mycontainer', function(error, result, response) {
  if(!error){
    var newSignedIdentifiers = extend(true, result.signedIdentifiers, sharedAccessPolicy);
    blobSvc.setBlobAcl('mycontainer', newSignedIdentifiers, function(error, result, response){
      if(!error){
        // ACL set
      }
    });
  }
});
```

Once the ACL is set, you can then create shared access signatures based on the ID for a policy. The following code example creates new shared access signatures for 'user2':

```
blobSAS = blobSvc.generateSharedAccessSignature('mycontainer', { Id: 'user2' });
```

Next steps

For more information, see the following resources.

- [Azure Storage SDK for Node API Reference](#)
- [Azure Storage Team Blog](#)
- [Azure Storage SDK for Node](#) repository on GitHub
- [Node.js Developer Center](#)
- [Transfer data with the AzCopy command-line utility](#)

How to use Blob Storage from C++

1/17/2017 • 11 min to read • [Edit on GitHub](#)

TIP

Manage Azure Blob Storage resources with Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to [manage Azure Blob Storage resources](#). Using Microsoft Azure Storage Explorer, you can visually create, read, update, and delete blob containers and blobs, as well as manage access to your blobs containers and blobs.

Overview

Azure Blob storage is a service that stores unstructured data in the cloud as objects/blobs. Blob storage can store any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as object storage.

This guide will demonstrate how to perform common scenarios using the Azure Blob storage service. The samples are written in C++ and use the [Azure Storage Client Library for C++](#). The scenarios covered include **uploading, listing, downloading, and deleting** blobs.

NOTE

This guide targets the Azure Storage Client Library for C++ version 1.0.0 and above. The recommended version is Storage Client Library 2.2.0, which is available via [NuGet](#) or [GitHub](#).

What is Blob Storage?

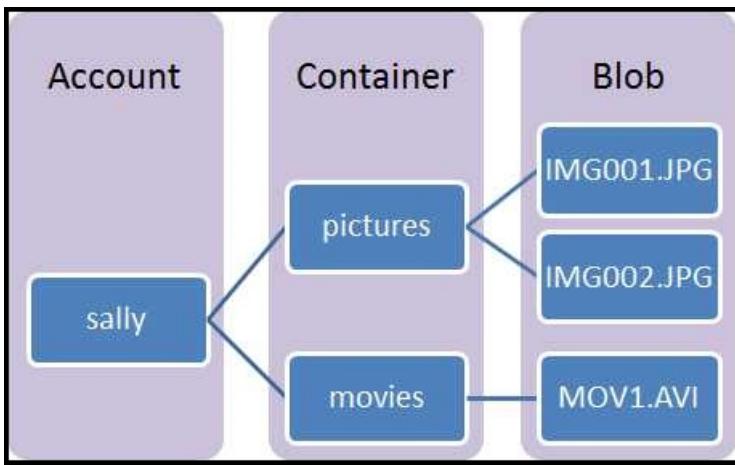
Azure Blob storage is a service for storing large amounts of unstructured object data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately.

Common uses of Blob storage include:

- 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

Blob service concepts

The Blob service contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. This storage account can be a **General-purpose storage account** or a **Blob storage account** which is specialized for storing objects/blobs. For more information about storage accounts, see [Azure storage account](#).
- **Container:** A container provides a grouping of a set of blobs. All blobs must be in a container. An account can contain an unlimited number of containers. A container can store an unlimited number of blobs. Note that the container name must be lowercase.
- **Blob:** A file of any type and size. Azure Storage offers three types of blobs: block blobs, page blobs, and append blobs.

Block blobs are ideal for storing text or binary files, such as documents and media files. *Append blobs* are similar to block blobs in that they are made up of blocks, but they are optimized for append operations, so they are useful for logging scenarios. A single block blob can contain up to 50,000 blocks of up to 100 MB each, for a total size of slightly more than 4.75 TB (100 MB X 50,000). A single append blob can contain up to 50,000 blocks of up to 4 MB each, for a total size of slightly more than 195 GB (4 MB X 50,000).

Page blobs can be up to 1 TB in size, and are more efficient for frequent read/write operations. Azure Virtual Machines use page blobs as OS and data disks.

For details about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a C++ application

In this guide, you will use storage features which can be run within a C++ application.

To do so, you will need to install the Azure Storage Client Library for C++ and create an Azure storage account in your Azure subscription.

To install the Azure Storage Client Library for C++, you can use the following methods:

- **Linux:** Follow the instructions given in the [Azure Storage Client Library for C++ README](#) page.

- **Windows:** In Visual Studio, click **Tools > NuGet Package Manager > Package Manager Console**. Type the following command into the [NuGet Package Manager console](#) and press **ENTER**.

```
Install-Package wastorage
```

Configure your application to access Blob Storage

Add the following include statements to the top of the C++ file where you want to use the Azure storage APIs to access blobs:

```
#include "was/storage_account.h"
#include "was/blob.h"
```

Setup an Azure storage connection string

An Azure storage client uses a storage connection string to store endpoints and credentials for accessing data management services. When running in a client application, you must provide the storage connection string in the following format, using the name of your storage account and the storage access key for the storage account listed in the [Azure Portal](#) for the *AccountName* and *AccountKey* values. For information on storage accounts and access keys, see [About Azure Storage Accounts](#). This example shows how you can declare a static field to hold the connection string:

```
// Define the connection-string with your values.
const utility::string_t
storage_connection_string(U("DefaultEndpointsProtocol=https;AccountName=your_storage_account;AccountKey=your_storage_account_key"));
```

To test your application in your local Windows computer, you can use the Microsoft Azure [storage emulator](#) that is installed with the [Azure SDK](#). The storage emulator is a utility that simulates the Blob, Queue, and Table services available in Azure on your local development machine. The following example shows how you can declare a static field to hold the connection string to your local storage emulator:

```
// Define the connection-string with Azure Storage Emulator.
const utility::string_t storage_connection_string(U("UseDevelopmentStorage=true;"));
```

To start the Azure storage emulator, Select the **Start** button or press the **Windows** key. Begin typing **Azure Storage Emulator**, and select **Microsoft Azure Storage Emulator** from the list of applications.

The following samples assume that you have used one of these two methods to get the storage connection string.

Retrieve your connection string

You can use the **cloud_storage_account** class to represent your Storage Account information. To retrieve your storage account information from the storage connection string, you can use the **parse** method.

```
// Retrieve storage account from connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);
```

Next, get a reference to a **cloud_blob_client** class as it allows you to retrieve objects that represent containers and blobs stored within the Blob Storage Service. The following code creates a **cloud_blob_client** object using the storage account object we retrieved above:

```
// Create the blob client.  
azure::storage::cloud_blob_client blob_client = storage_account.create_cloud_blob_client();
```

How to: Create a container

Every blob in Azure storage must reside in a container. The container forms part of the blob name. For example, `mycontainer` is the name of the container in these sample blob URLs:

```
https://storagesample.blob.core.windows.net/mycontainer/blob1.txt  
https://storagesample.blob.core.windows.net/mycontainer/photos/myphoto.jpg
```

A container name must be a valid DNS name, conforming to the following naming rules:

1. Container names must start with a letter or number, and can contain only letters, numbers, and the dash (-) character.
2. Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive dashes are not permitted in container names.
3. All letters in a container name must be lowercase.
4. Container names must be from 3 through 63 characters long.

IMPORTANT

Note that the name of a container must always be lowercase. If you include an upper-case letter in a container name, or otherwise violate the container naming rules, you may receive a 400 error (Bad Request).

This example shows how to create a container if it does not already exist:

```
try  
{  
    // Retrieve storage account from connection string.  
    azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
    // Create the blob client.  
    azure::storage::cloud_blob_client blob_client = storage_account.create_cloud_blob_client();  
  
    // Retrieve a reference to a container.  
    azure::storage::cloud_blob_container container = blob_client.get_container_reference(U("my-sample-  
    container"));  
  
    // Create the container if it doesn't already exist.  
    container.create_if_not_exists();  
}  
catch (const std::exception& e)  
{  
    std::wcout << U("Error: ") << e.what() << std::endl;  
}
```

By default, the new container is private and you must specify your storage access key to download blobs from this container. If you want to make the files (blobs) within the container available to everyone, you can set the container to be public using the following code:

```
// Make the blob container publicly accessible.  
azure::storage::blob_container_permissions permissions;  
permissions.set_public_access(azure::storage::blob_container_public_access_type::blob);  
container.upload_permissions(permissions);
```

Anyone on the Internet can see blobs in a public container, but you can modify or delete them only if you have the appropriate access key.

How to: Upload a blob into a container

Azure Blob Storage supports block blobs and page blobs. In the majority of cases, block blob is the recommended type to use.

To upload a file to a block blob, get a container reference and use it to get a block blob reference. Once you have a blob reference, you can upload any stream of data to it by calling the **upload_from_stream** method. This operation will create the blob if it didn't previously exist, or overwrite it if it does exist. The following example shows how to upload a blob into a container and assumes that the container was already created.

```
// Retrieve storage account from connection string.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the blob client.  
azure::storage::cloud_blob_client blob_client = storage_account.create_cloud_blob_client();  
  
// Retrieve a reference to a previously created container.  
azure::storage::cloud_blob_container container = blob_client.get_container_reference(U("my-sample-container"));  
  
// Retrieve reference to a blob named "my-blob-1".  
azure::storage::cloud_block_blob blockBlob = container.get_block_blob_reference(U("my-blob-1"));  
  
// Create or overwrite the "my-blob-1" blob with contents from a local file.  
concurrency::streams::istream input_stream =  
    concurrency::streams::file_stream<uint8_t>::open_istream(U("DataFile.txt")).get();  
blockBlob.upload_from_stream(input_stream);  
input_stream.close().wait();  
  
// Create or overwrite the "my-blob-2" and "my-blob-3" blobs with contents from text.  
// Retrieve a reference to a blob named "my-blob-2".  
azure::storage::cloud_block_blob blob2 = container.get_block_blob_reference(U("my-blob-2"));  
blob2.upload_text(U("more text"));  
  
// Retrieve a reference to a blob named "my-blob-3".  
azure::storage::cloud_block_blob blob3 = container.get_block_blob_reference(U("my-directory/my-sub-  
directory/my-blob-3"));  
blob3.upload_text(U("other text"));
```

Alternatively, you can use the **upload_from_file** method to upload a file to a block blob.

How to: List the blobs in a container

To list the blobs in a container, first get a container reference. You can then use the container's **list_blobs** method to retrieve the blobs and/or directories within it. To access the rich set of properties and methods for a returned **list_blob_item**, you must call the **list_blob_item.as_blob** method to get a **cloud_blob** object, or the **list_blob.as_directory** method to get a **cloud_blob_directory** object. The following code demonstrates how to retrieve and output the URI of each item in the **my-sample-container** container:

```

// Retrieve storage account from connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the blob client.
azure::storage::cloud_blob_client blob_client = storage_account.create_cloud_blob_client();

// Retrieve a reference to a previously created container.
azure::storage::cloud_blob_container container = blob_client.get_container_reference(U("my-sample-container"));

// Output URI of each item.
azure::storage::list_blob_item_iterator end_of_results;
for (auto it = container.list_blobs(); it != end_of_results; ++it)
{
    if (it->is_blob())
    {
        std::wcout << U("Blob: ") << it->as_blob().uri().primary_uri().to_string() << std::endl;
    }
    else
    {
        std::wcout << U("Directory: ") << it->as_directory().uri().primary_uri().to_string() << std::endl;
    }
}

```

For more details on listing operations, see [List Azure Storage Resources in C++](#).

How to: Download blobs

To download blobs, first retrieve a blob reference and then call the **download_to_stream** method. The following example uses the **download_to_stream** method to transfer the blob contents to a stream object that you can then persist to a local file.

```

// Retrieve storage account from connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the blob client.
azure::storage::cloud_blob_client blob_client = storage_account.create_cloud_blob_client();

// Retrieve a reference to a previously created container.
azure::storage::cloud_blob_container container = blob_client.get_container_reference(U("my-sample-container"));

// Retrieve reference to a blob named "my-blob-1".
azure::storage::cloud_block_blob blockBlob = container.get_block_blob_reference(U("my-blob-1"));

// Save blob contents to a file.
concurrency::streams::container_buffer<std::vector<uint8_t>> buffer;
concurrency::streams::ostream output_stream(buffer);
blockBlob.download_to_stream(output_stream);

std::ofstream outfile("DownloadBlobFile.txt", std::ofstream::binary);
std::vector<unsigned char>& data = buffer.collection();

outfile.write((char *)&data[0], buffer.size());
outfile.close();

```

Alternatively, you can use the **download_to_file** method to download the contents of a blob to a file. In addition, you can also use the **download_text** method to download the contents of a blob as a text string.

```
// Retrieve storage account from connection string.  
azure::storage::cloud_storage_account storage_account =  
azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the blob client.  
azure::storage::cloud_blob_client blob_client = storage_account.create_cloud_blob_client();  
  
// Retrieve a reference to a previously created container.  
azure::storage::cloud_blob_container container = blob_client.get_container_reference(U("my-sample-container"));  
  
// Retrieve reference to a blob named "my-blob-2".  
azure::storage::cloud_block_blob text_blob = container.get_block_blob_reference(U("my-blob-2"));  
  
// Download the contents of a blob as a text string.  
utility::string_t text = text_blob.download_text();
```

How to: Delete blobs

To delete a blob, first get a blob reference and then call the **delete_blob** method on it.

```
// Retrieve storage account from connection string.  
azure::storage::cloud_storage_account storage_account =  
azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the blob client.  
azure::storage::cloud_blob_client blob_client = storage_account.create_cloud_blob_client();  
  
// Retrieve a reference to a previously created container.  
azure::storage::cloud_blob_container container = blob_client.get_container_reference(U("my-sample-container"));  
  
// Retrieve reference to a blob named "my-blob-1".  
azure::storage::cloud_block_blob blockBlob = container.get_block_blob_reference(U("my-blob-1"));  
  
// Delete the blob.  
blockBlob.delete_blob();
```

Next steps

Now that you've learned the basics of blob storage, follow these links to learn more about Azure Storage.

- [How to use Queue Storage from C++](#)
- [How to use Table Storage from C++](#)
- [List Azure Storage Resources in C++](#)
- [Storage Client Library for C++ Reference](#)
- [Azure Storage Documentation](#)
- [Transfer data with the AzCopy command-line utility](#)

How to use Azure Blob storage from Python

1/17/2017 • 7 min to read • [Edit on GitHub](#)

TIP

Manage Azure Blob Storage resources with Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to [manage Azure Blob Storage resources](#). Using Microsoft Azure Storage Explorer, you can visually create, read, update, and delete blob containers and blobs, as well as manage access to your blobs containers and blobs.

Overview

Azure Blob storage is a service that stores unstructured data in the cloud as objects/blobs. Blob storage can store any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as object storage.

This article will show you how to perform common scenarios using Blob storage. The samples are written in Python and use the [Microsoft Azure Storage SDK for Python](#). The scenarios covered include uploading, listing, downloading, and deleting blobs.

What is Blob Storage?

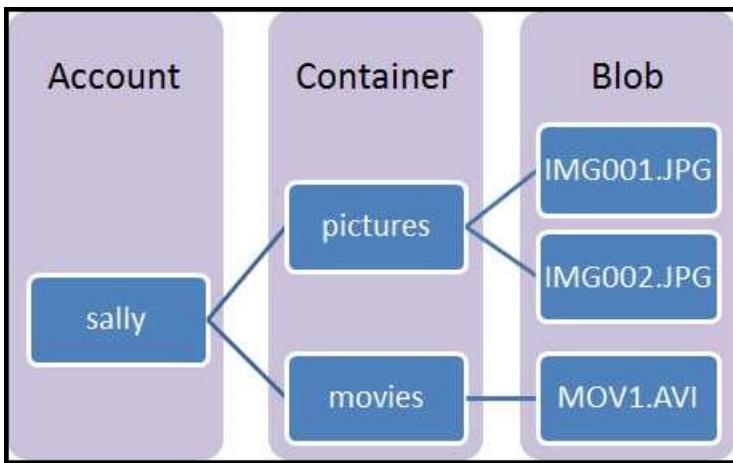
Azure Blob storage is a service for storing large amounts of unstructured object data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately.

Common uses of Blob storage include:

- 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

Blob service concepts

The Blob service contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. This storage account can be a **General-purpose storage account** or a **Blob storage account** which is specialized for storing objects/blobs. For more information about storage accounts, see [Azure storage account](#).
- **Container:** A container provides a grouping of a set of blobs. All blobs must be in a container. An account can contain an unlimited number of containers. A container can store an unlimited number of blobs. Note that the container name must be lowercase.
- **Blob:** A file of any type and size. Azure Storage offers three types of blobs: block blobs, page blobs, and append blobs.

Block blobs are ideal for storing text or binary files, such as documents and media files. *Append blobs* are similar to block blobs in that they are made up of blocks, but they are optimized for append operations, so they are useful for logging scenarios. A single block blob can contain up to 50,000 blocks of up to 100 MB each, for a total size of slightly more than 4.75 TB (100 MB X 50,000). A single append blob can contain up to 50,000 blocks of up to 4 MB each, for a total size of slightly more than 195 GB (4 MB X 50,000).

Page blobs can be up to 1 TB in size, and are more efficient for frequent read/write operations. Azure Virtual Machines use page blobs as OS and data disks.

For details about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a container

Based on the type of blob you would like to use, create a **BlockBlobService**, **AppendBlobService**, or **PageBlobService** object. The following code uses a **BlockBlobService** object. Add the following near the top of any Python file in which you wish to programmatically access Azure Block Blob Storage.

```
from azure.storage.blob import BlockBlobService
```

The following code creates a **BlockBlobService** object using the storage account name and account key. Replace

'myaccount' and 'mykey' with your account name and key.

```
block_blob_service = BlockBlobService(account_name='myaccount', account_key='mykey')
```

Every blob in Azure storage must reside in a container. The container forms part of the blob name. For example, `mycontainer` is the name of the container in these sample blob URLs:

```
https://storagesample.blob.core.windows.net/mycontainer/blob1.txt  
https://storagesample.blob.core.windows.net/mycontainer/photos/myphoto.jpg
```

A container name must be a valid DNS name, conforming to the following naming rules:

1. Container names must start with a letter or number, and can contain only letters, numbers, and the dash (-) character.
2. Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive dashes are not permitted in container names.
3. All letters in a container name must be lowercase.
4. Container names must be from 3 through 63 characters long.

IMPORTANT

Note that the name of a container must always be lowercase. If you include an upper-case letter in a container name, or otherwise violate the container naming rules, you may receive a 400 error (Bad Request).

In the following code example, you can use a **BlockBlobService** object to create the container if it doesn't exist.

```
block_blob_service.create_container('mycontainer')
```

By default, the new container is private, so you must specify your storage access key (as you did earlier) to download blobs from this container. If you want to make the blobs within the container available to everyone, you can create the container and pass the public access level using the following code.

```
from azure.storage.blob import PublicAccess  
block_blob_service.create_container('mycontainer', public_access=PublicAccess.Container)
```

Alternatively, you can modify a container after you have created it using the following code.

```
block_blob_service.set_container_acl('mycontainer', public_access=PublicAccess.Container)
```

After this change, anyone on the Internet can see blobs in a public container, but only you can modify or delete them.

Upload a blob into a container

To create a block blob and upload data, use the **create_blob_from_path**, **create_blob_from_stream**, **create_blob_from_bytes** or **create_blob_from_text** methods. They are high-level methods that perform the necessary chunking when the size of the data exceeds 64 MB.

create_blob_from_path uploads the contents of a file from the specified path, and **create_blob_from_stream** uploads the contents from an already opened file/stream. **create_blob_from_bytes** uploads an array of bytes, and **create_blob_from_text** uploads the specified text value using the specified encoding (defaults to UTF-8).

The following example uploads the contents of the **sunset.png** file into the **myblob** blob.

```
from azure.storage.blob import ContentSettings
block_blob_service.create_blob_from_path(
    'mycontainer',
    'myblockblob',
    'sunset.png',
    content_settings=ContentSettings(content_type='image/png')
)
```

List the blobs in a container

To list the blobs in a container, use the **list_blobs** method. This method returns a generator. The following code outputs the **name** of each blob in a container to the console.

```
generator = block_blob_service.list_blobs('mycontainer')
for blob in generator:
    print(blob.name)
```

Download blobs

To download data from a blob, use **get_blob_to_path**, **get_blob_to_stream**, **get_blob_to_bytes**, or **get_blob_to_text**. They are high-level methods that perform the necessary chunking when the size of the data exceeds 64 MB.

The following example demonstrates using **get_blob_to_path** to download the contents of the **myblob** blob and store it to the **out-sunset.png** file.

```
block_blob_service.get_blob_to_path('mycontainer', 'myblockblob', 'out-sunset.png')
```

Delete a blob

Finally, to delete a blob, call **delete_blob**.

```
block_blob_service.delete_blob('mycontainer', 'myblockblob')
```

Writing to an append blob

An append blob is optimized for append operations, such as logging. Like a block blob, an append blob is comprised of blocks, but when you add a new block to an append blob, it is always appended to the end of the blob. You cannot update or delete an existing block in an append blob. The block IDs for an append blob are not exposed as they are for a block blob.

Each block in an append blob can be a different size, up to a maximum of 4 MB, and an append blob can include a maximum of 50,000 blocks. The maximum size of an append blob is therefore slightly more than 195 GB (4 MB X 50,000 blocks).

The example below creates a new append blob and appends some data to it, simulating a simple logging operation.

```
from azure.storage.blob import AppendBlobService
append_blob_service = AppendBlobService(account_name='myaccount', account_key='mykey')

# The same containers can hold all types of blobs
append_blob_service.create_container('mycontainer')

# Append blobs must be created before they are appended to
append_blob_service.create_blob('mycontainer', 'myappendblob')
append_blob_service.append_blob_from_text('mycontainer', 'myappendblob', u'Hello, world!')

append_blob = append_blob_service.get_blob_to_text('mycontainer', 'myappendblob')
```

Next steps

Now that you've learned the basics of Blob storage, follow these links to learn more.

- [Python Developer Center](#)
- [Azure Storage Services REST API](#)
- [Azure Storage Team Blog](#)
- [Microsoft Azure Storage SDK for Python](#)

How to use blob storage from PHP

1/17/2017 • 10 min to read • [Edit on GitHub](#)

TIP

Manage Azure Blob Storage resources with Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to [manage Azure Blob Storage resources](#). Using Microsoft Azure Storage Explorer, you can visually create, read, update, and delete blob containers and blobs, as well as manage access to your blobs containers and blobs.

Overview

Azure Blob storage is a service that stores unstructured data in the cloud as objects/blobs. Blob storage can store any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as object storage.

This guide shows you how to perform common scenarios using the Azure blob service. The samples are written in PHP and use the [Azure SDK for PHP](#). The scenarios covered include **uploading**, **listing**, **downloading**, and **deleting** blobs. For more information on blobs, see the [Next steps](#) section.

What is Blob Storage?

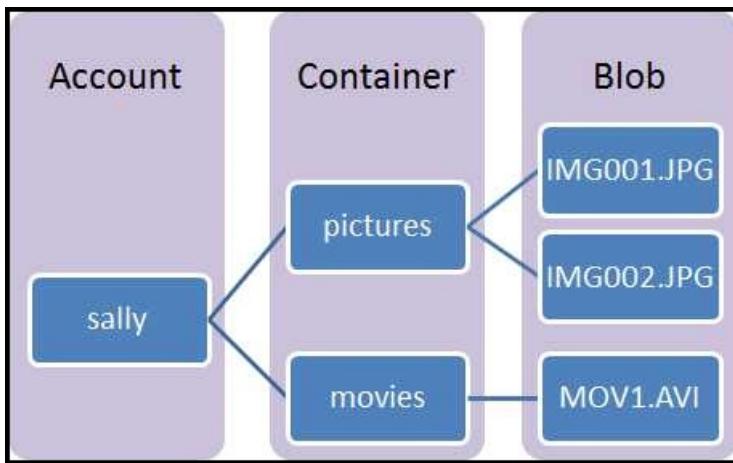
Azure Blob storage is a service for storing large amounts of unstructured object data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately.

Common uses of Blob storage include:

- 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

Blob service concepts

The Blob service contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. This storage account can be a **General-purpose storage account** or a **Blob storage account** which is specialized for storing objects/blobs. For more information about storage accounts, see [Azure storage account](#).
- **Container:** A container provides a grouping of a set of blobs. All blobs must be in a container. An account can contain an unlimited number of containers. A container can store an unlimited number of blobs. Note that the container name must be lowercase.
- **Blob:** A file of any type and size. Azure Storage offers three types of blobs: block blobs, page blobs, and append blobs.

Block blobs are ideal for storing text or binary files, such as documents and media files. *Append blobs* are similar to block blobs in that they are made up of blocks, but they are optimized for append operations, so they are useful for logging scenarios. A single block blob can contain up to 50,000 blocks of up to 100 MB each, for a total size of slightly more than 4.75 TB (100 MB X 50,000). A single append blob can contain up to 50,000 blocks of up to 4 MB each, for a total size of slightly more than 195 GB (4 MB X 50,000).

Page blobs can be up to 1 TB in size, and are more efficient for frequent read/write operations. Azure Virtual Machines use page blobs as OS and data disks.

For details about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a PHP application

The only requirement for creating a PHP application that accesses the Azure blob service is the referencing of classes in the Azure SDK for PHP from within your code. You can use any development tools to create your application, including Notepad.

In this guide, you use service features, which can be called within a PHP application locally or in code running within an Azure web role, worker role, or website.

Get the Azure Client Libraries

Install via Composer

1. [Install Git](#). Note that on Windows, you must also add the Git executable to your PATH environment variable.
2. Create a file named **composer.json** in the root of your project and add the following code to it:

```
{  
    "require": {  
        "microsoft/windowsazure": "^0.4"  
    }  
}
```

3. Download [composer.phar](#) in your project root.
4. Open a command prompt and execute the following command in your project root

```
php composer.phar install
```

Configure your application to access the blob service

To use the Azure blob service APIs, you need to:

1. Reference the autoloader file using the `require_once` statement, and
2. Reference any classes you might use.

The following example shows how to include the autoloader file and reference the **ServicesBuilder** class.

NOTE

The examples in this article assume you have installed the PHP Client Libraries for Azure via Composer. If you installed the libraries manually, you need to reference the `WindowsAzure.php` autoloader file.

```
require_once 'vendor/autoload.php';  
use WindowsAzure\Common\ServicesBuilder;
```

In the examples below, the `require_once` statement will be shown always, but only the classes necessary for the example to execute are referenced.

Set up an Azure storage connection

To instantiate an Azure blob service client, you must first have a valid connection string. The format for the blob service connection string is:

For accessing a live service:

```
DefaultEndpointsProtocol=[http|https];AccountName=[yourAccount];AccountKey=[yourKey]
```

For accessing the storage emulator:

```
UseDevelopmentStorage=true
```

To create any Azure service client, you need to use the **ServicesBuilder** class. You can:

- Pass the connection string directly to it or

- Use the **CloudConfigurationManager (CCM)** to check multiple external sources for the connection string:
 - By default, it comes with support for one external source - environmental variables.
 - You can add new sources by extending the **ConnectionStringSource** class.

For the examples outlined here, the connection string will be passed directly.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;

$blobRestProxy = ServicesBuilder::getInstance()->createBlobService($connectionString);
```

Create a container

Every blob in Azure storage must reside in a container. The container forms part of the blob name. For example, `mycontainer` is the name of the container in these sample blob URLs:

```
https://storagesample.blob.core.windows.net/mycontainer/blob1.txt
https://storagesample.blob.core.windows.net/mycontainer/photos/myphoto.jpg
```

A container name must be a valid DNS name, conforming to the following naming rules:

1. Container names must start with a letter or number, and can contain only letters, numbers, and the dash (-) character.
2. Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive dashes are not permitted in container names.
3. All letters in a container name must be lowercase.
4. Container names must be from 3 through 63 characters long.

IMPORTANT

Note that the name of a container must always be lowercase. If you include an upper-case letter in a container name, or otherwise violate the container naming rules, you may receive a 400 error (Bad Request).

A **BlobRestProxy** object lets you create a blob container with the **createContainer** method. When creating a container, you can set options on the container, but doing so is not required. (The example below shows how to set the container access control list (ACL) and container metadata.)

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Blob\Models\CreateContainerOptions;
use MicrosoftAzure\Storage\Blob\Models\PublicAccessType;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create blob REST proxy.
$blobRestProxy = ServicesBuilder::getInstance()->createBlobService($connectionString);

// OPTIONAL: Set public access policy and metadata.
// Create container options object.
$createContainerOptions = new CreateContainerOptions();

// Set public access policy. Possible values are
// PublicAccessType::CONTAINER_AND_BLOBS and PublicAccessType::BLOBS_ONLY.
// CONTAINER_AND_BLOBS:
// Specifies full public read access for container and blob data.
// proxys can enumerate blobs within the container via anonymous
// request, but cannot enumerate containers within the storage account.
//
// BLOBS_ONLY:
// Specifies public read access for blobs. Blob data within this
// container can be read via anonymous request, but container data is not
// available. proxys cannot enumerate blobs within the container via
// anonymous request.
// If this value is not specified in the request, container data is
// private to the account owner.
$createContainerOptions->setPublicAccess(PublicAccessType::CONTAINER_AND_BLOBS);

// Set container metadata.
$createContainerOptions->addMetaData("key1", "value1");
$createContainerOptions->addMetaData("key2", "value2");

try {
    // Create container.
    $blobRestProxy->createContainer("mycontainer", $createContainerOptions);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179439.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Calling **setPublicAccess(PublicAccessType::CONTAINER_AND_BLOBS)** makes the container and blob data accessible via anonymous requests. Calling **setPublicAccess(PublicAccessType::BLOBS_ONLY)** makes only blob data accessible via anonymous requests. For more information about container ACLs, see [Set container ACL \(REST API\)](#).

For more information about Blob service error codes, see [Blob Service Error Codes](#).

Upload a blob into a container

To upload a file as a blob, use the **BlobRestProxy->createBlockBlob** method. This operation creates the blob if it doesn't exist, or overwrites it if it does. The code example below assumes that the container has already been created and uses **fopen** to open the file as a stream.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create blob REST proxy.
$blobRestProxy = ServicesBuilder::getInstance()->createBlobService($connectionString);

$content = fopen("c:\myfile.txt", "r");
$blob_name = "myblob";

try {
    //Upload blob
    $blobRestProxy->createBlockBlob("mycontainer", $blob_name, $content);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179439.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Note that the previous sample uploads a blob as a stream. However, a blob can also be uploaded as a string using, for example, the [file_get_contents](#) function. To do this using the previous sample, change

```
$content = fopen("c:\myfile.txt", "r"); to $content = file_get_contents("c:\myfile.txt");
```

List the blobs in a container

To list the blobs in a container, use the **BlobRestProxy->listBlobs** method with a **foreach** loop to loop through the result. The following code displays the name of each blob as output in a container and displays its URI to the browser.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create blob REST proxy.
$blobRestProxy = ServicesBuilder::getInstance()->createBlobService($connectionString);

try {
    // List blobs.
    $blob_list = $blobRestProxy->listBlobs("mycontainer");
    $blobs = $blob_list->getBlobs();

    foreach($blobs as $blob)
    {
        echo $blob->getName()." : ".$blob->getUrl()."<br />";
    }
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179439.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Download a blob

To download a blob, call the **BlobRestProxy->getBlob** method, then call the **getContentStream** method on the resulting **GetBlobResult** object.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create blob REST proxy.
$blobRestProxy = ServicesBuilder::getInstance()->createBlobService($connectionString);

try {
    // Get blob.
    $blob = $blobRestProxy->getBlob("mycontainer", "myblob");
    fpassthru($blob->getContentStream());
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179439.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

Note that the example above gets a blob as a stream resource (the default behavior). However, you can use the [stream_get_contents](#) function to convert the returned stream to a string.

Delete a blob

To delete a blob, pass the container name and blob name to **BlobRestProxy->deleteBlob**.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create blob REST proxy.
$blobRestProxy = ServicesBuilder::getInstance()->createBlobService($connectionString);

try {
    // Delete blob.
    $blobRestProxy->deleteBlob("mycontainer", "myblob");
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179439.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

Delete a blob container

Finally, to delete a blob container, pass the container name to **BlobRestProxy->deleteContainer**.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create blob REST proxy.
$blobRestProxy = ServicesBuilder::getInstance()->createBlobService($connectionString);

try {
    // Delete container.
    $blobRestProxy->deleteContainer("mycontainer");
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179439.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

Next steps

Now that you've learned the basics of the Azure blob service, follow these links to learn about more complex storage tasks.

- Visit the [Azure Storage team blog](#)
- See the [PHP block blob example](#).
- See the [PHP page blob example](#).
- [Transfer data with the AzCopy Command-Line Utility](#)

For more information, see also the [PHP Developer Center](#).

How to use Blob storage from Ruby

1/17/2017 • 7 min to read • [Edit on GitHub](#)

TIP

Manage Azure Blob Storage resources with Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to [manage Azure Blob Storage resources](#). Using Microsoft Azure Storage Explorer, you can visually create, read, update, and delete blob containers and blobs, as well as manage access to your blobs containers and blobs.

Overview

Azure Blob storage is a service that stores unstructured data in the cloud as objects/blobs. Blob storage can store any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as object storage.

This guide will show you how to perform common scenarios using Blob storage. The samples are written using the Ruby API. The scenarios covered include **uploading, listing, downloading, and deleting** blobs.

What is Blob Storage?

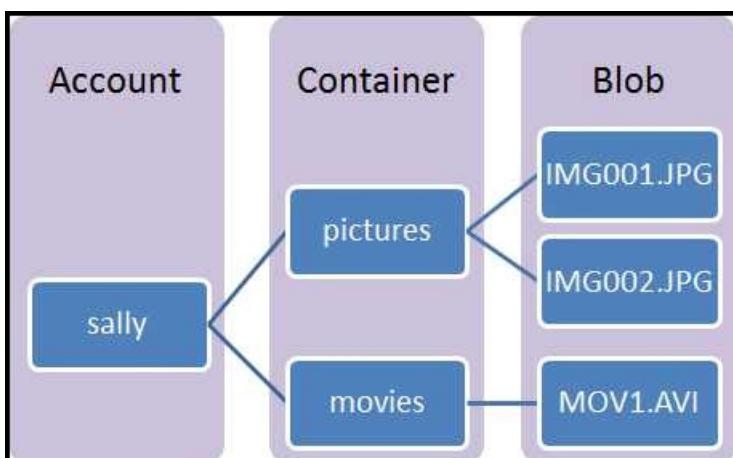
Azure Blob storage is a service for storing large amounts of unstructured object data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately.

Common uses of Blob storage include:

- 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

Blob service concepts

The Blob service contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. This storage account can be a **General-purpose storage account** or a **Blob storage account** which is specialized for storing objects/blobs. For more information about storage accounts, see [Azure storage account](#).
- **Container:** A container provides a grouping of a set of blobs. All blobs must be in a container. An account can contain an unlimited number of containers. A container can store an unlimited number of blobs. Note that the container name must be lowercase.
- **Blob:** A file of any type and size. Azure Storage offers three types of blobs: block blobs, page blobs, and append blobs.

Block blobs are ideal for storing text or binary files, such as documents and media files. *Append blobs* are similar to block blobs in that they are made up of blocks, but they are optimized for append operations, so they are useful for logging scenarios. A single block blob can contain up to 50,000 blocks of up to 100 MB each, for a total size of slightly more than 4.75 TB (100 MB X 50,000). A single append blob can contain up to 50,000 blocks of up to 4 MB each, for a total size of slightly more than 195 GB (4 MB X 50,000).

Page blobs can be up to 1 TB in size, and are more efficient for frequent read/write operations. Azure Virtual Machines use page blobs as OS and data disks.

For details about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a Ruby application

Create a Ruby application. For instructions, see [Ruby on Rails Web application on an Azure VM](#)

Configure your application to access Storage

To use Azure Storage, you need to download and use the Ruby azure package, which includes a set of convenience libraries that communicate with the storage REST services.

Use RubyGems to obtain the package

1. Use a command-line interface such as **PowerShell** (Windows), **Terminal** (Mac), or **Bash** (Unix).
2. Type "gem install azure" in the command window to install the gem and dependencies.

Import the package

Using your favorite text editor, add the following to the top of the Ruby file where you intend to use storage:

```
require "azure"
```

Set up an Azure Storage Connection

The azure module will read the environment variables **AZURE_STORAGE_ACCOUNT** and

AZURE_STORAGE_ACCESS_KEY for information required to connect to your Azure storage account. If these environment variables are not set, you must specify the account information before using **Azure::Blob::BlobService** with the following code:

```
Azure.config.storage_account_name = "<your azure storage account>"  
Azure.config.storage_access_key = "<your azure storage access key>"
```

To obtain these values from a classic or Resource Manager storage account in the Azure portal:

1. Log in to the [Azure portal](#).
2. Navigate to the storage account you want to use.
3. In the Settings blade on the right, click **Access Keys**.
4. In the Access keys blade that appears, you'll see the access key 1 and access key 2. You can use either of these.
5. Click the copy icon to copy the key to the clipboard.

To obtain these values from a classic storage account in the classic Azure portal:

1. Log in to the [classic Azure portal](#).
2. Navigate to the storage account you want to use.
3. Click **MANAGE ACCESS KEYS** at the bottom of the navigation pane.
4. In the pop-up dialog, you'll see the storage account name, primary access key and secondary access key. For access key, you can use either the primary one or the secondary one.
5. Click the copy icon to copy the key to the clipboard.

Create a container

Every blob in Azure storage must reside in a container. The container forms part of the blob name. For example, `mycontainer` is the name of the container in these sample blob URLs:

```
https://storagesample.blob.core.windows.net/mycontainer/blob1.txt  
https://storagesample.blob.core.windows.net/mycontainer/photos/myphoto.jpg
```

A container name must be a valid DNS name, conforming to the following naming rules:

1. Container names must start with a letter or number, and can contain only letters, numbers, and the dash (-) character.
2. Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive dashes are not permitted in container names.
3. All letters in a container name must be lowercase.
4. Container names must be from 3 through 63 characters long.

IMPORTANT

Note that the name of a container must always be lowercase. If you include an upper-case letter in a container name, or otherwise violate the container naming rules, you may receive a 400 error (Bad Request).

The **Azure::Blob::BlobService** object lets you work with containers and blobs. To create a container, use the **create_container()** method.

The following code example creates a container or prints the error if there is any.

```
azure_blob_service = Azure::Blob::BlobService.new
begin
  container = azure_blob_service.create_container("test-container")
rescue
  puts $!
end
```

If you want to make the files in the container public, you can set the container's permissions.

You can just modify the **create_container()** call to pass the **:public_access_level** option:

```
container = azure_blob_service.create_container("test-container",
  :public_access_level => "<public access level>")
```

Valid values for the **:public_access_level** option are:

- **blob:** Specifies public read access for blobs. Blob data within this container can be read via anonymous request, but container data is not available. Clients cannot enumerate blobs within the container via anonymous request.
- **container:** Specifies full public read access for container and blob data. Clients can enumerate blobs within the container via anonymous request, but cannot enumerate containers within the storage account.

Alternatively, you can modify the public access level of a container by using **set_container_acl()** method to specify the public access level.

The following code example changes the public access level to **container**:

```
azure_blob_service.set_container_acl('test-container', "container")
```

Upload a blob into a container

To upload content to a blob, use the **create_block_blob()** method to create the blob, use a file or string as the content of the blob.

The following code uploads the file **test.png** as a new blob named "image-blob" in the container.

```
content = File.open("test.png", "rb") { |file| file.read }
blob = azure_blob_service.create_block_blob(container.name,
  "image-blob", content)
puts blob.name
```

List the blobs in a container

To list the containers, use **list_containers()** method. To list the blobs within a container, use **list_blobs()** method.

This outputs the urls of all the blobs in all the containers for the account.

```
containers = azure_blob_service.list_containers()
containers.each do |container|
  blobs = azure_blob_service.list_blobs(container.name)
  blobs.each do |blob|
    puts blob.name
  end
end
```

Download blobs

To download blobs, use the **get_blob()** method to retrieve the contents.

The following code example demonstrates using **get_blob()** to download the contents of "image-blob" and write it to a local file.

```
blob, content = azure_blob_service.get_blob(container.name,"image-blob")
File.open("download.png","wb") {|f| f.write(content)}
```

Delete a Blob

Finally, to delete a blob, use the **delete_blob()** method. The following code example demonstrates how to delete a blob.

```
azure_blob_service.delete_blob(container.name, "image-blob")
```

Next steps

To learn about more complex storage tasks, follow these links:

- [Azure Storage Team Blog](#)
- [Azure SDK for Ruby](#) repository on GitHub
- [Transfer data with the AzCopy Command-Line Utility](#)

How to use Blob storage from iOS

1/17/2017 • 14 min to read • [Edit on GitHub](#)

TIP

Manage Azure Blob Storage resources with Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to [manage Azure Blob Storage resources](#).

Using Microsoft Azure Storage Explorer, you can visually create, read, update, and delete blob containers and blobs, as well as manage access to your blobs containers and blobs.

Overview

This article will show you how to perform common scenarios using Microsoft Azure Blob storage. The samples are written in Objective-C and use the [Azure Storage Client Library for iOS](#). The scenarios covered include **uploading**, **listing**, **downloading**, and **deleting** blobs. For more information on blobs, see the [Next Steps](#) section. You can also download the [sample app](#) to quickly see the use of Azure Storage in an iOS application.

What is Blob Storage?

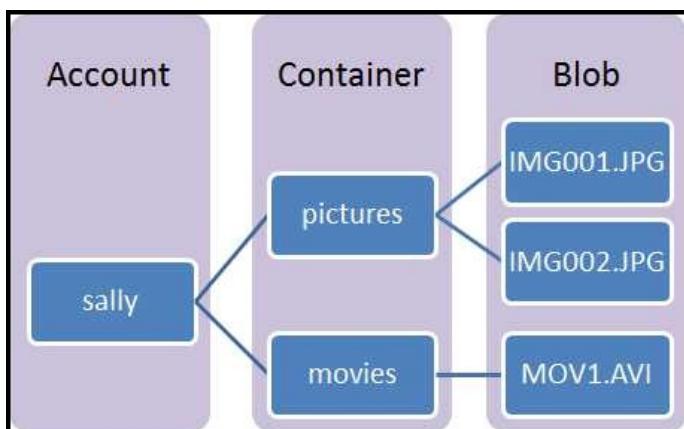
Azure Blob storage is a service for storing large amounts of unstructured object data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately.

Common uses of Blob storage include:

- 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

Blob service concepts

The Blob service contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. This storage account can be a **General-purpose storage account** or a **Blob storage account** which is specialized for storing objects/blobs. For more information about storage accounts, see [Azure storage account](#).
- **Container:** A container provides a grouping of a set of blobs. All blobs must be in a container. An account can contain an unlimited number of containers. A container can store an unlimited number of blobs. Note that the container name

must be lowercase.

- **Blob:** A file of any type and size. Azure Storage offers three types of blobs: block blobs, page blobs, and append blobs.

Block blobs are ideal for storing text or binary files, such as documents and media files. *Append blobs* are similar to block blobs in that they are made up of blocks, but they are optimized for append operations, so they are useful for logging scenarios. A single block blob can contain up to 50,000 blocks of up to 100 MB each, for a total size of slightly more than 4.75 TB (100 MB X 50,000). A single append blob can contain up to 50,000 blocks of up to 4 MB each, for a total size of slightly more than 195 GB (4 MB X 50,000).

Page blobs can be up to 1 TB in size, and are more efficient for frequent read/write operations. Azure Virtual Machines use page blobs as OS and data disks.

For details about naming containers and blobs, see [Naming and Referencing Containers, Blobs, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Import the Azure Storage iOS library into your application

You can import the Azure Storage iOS library into your application either by using the [Azure Storage CocoaPod](#) or by importing the **Framework** file.

CocoaPod

1. If you haven't done so already, [Install CocoaPods](#) on your computer by opening a terminal window and running the following command

```
sudo gem install cocoapods
```

2. Next, in the project directory (the directory containing your .xcodeproj file), create a new file called *Podfile*(no file extension). Add the following to *Podfile* and save.

```
pod 'AZSClient'
```

3. In the terminal window, navigate to the project directory and run the following command

```
pod install
```

4. If your .xcodeproj is open in Xcode, close it. In your project directory open the newly created project file which will have the .xcworkspace extension. This is the file you'll work from for now on.

Framework

In order to use the Azure Storage iOS library, you will first need to build the framework file.

1. First, download or clone the [azure-storage-ios repo](#).
2. Go into *azure-storage-ios -> Lib -> Azure Storage Client Library*, and open AZSClient.xcodeproj in Xcode.
3. At the top-left of Xcode, change the active scheme from "Azure Storage Client Library" to "Framework".
4. Build the project ($\text{Shift}+\text{B}$). This will create an AZSClient.framework file on your Desktop.

You can then import the framework file into your application by doing the following:

1. Create a new project or open up your existing project in Xcode.

2. Click on your project in the left-hand navigation and click the *General* tab at the top of the project editor.
3. Under the *Linked Frameworks and Libraries* section, click the Add button (+).
4. Click *Add Other...*. Navigate to and add the `AZSClient.framework` file you just created.
5. Under the *Linked Frameworks and Libraries* section, click the Add button (+) again.
6. In the list of libraries already provided, search for `libxml2.2.dylib` and add it to your project.
7. Click the *Build Settings* tab at the top of the project editor.
8. Under the *Search Paths* section, double-click *Framework Search Paths* and add the path to your `AZSClient.framework` file.

Import Statement

You will need to include the following import statement in the file where you want to invoke the Azure Storage API.

```
// Include the following import statement to use blob APIs.
#import <AZSClient/AZSClient.h>
```

Configure your application to access Azure Storage

There are two ways to authenticate your application to access Storage services:

- Shared Key: Use Shared Key for testing purposes only
- Shared Access Signature (SAS): Use SAS for production applications

Shared Key

Shared Key authentication means that your application will use your account name and account key to access Storage services. For the purposes of quickly showing how to use this library, we will be using Shared Key authentication in this getting started.

WARNING

Only use Shared Key authentication for testing purposes! Your account name and account key, which give full read/write access to the associated Storage account, will be distributed to every person that downloads your app. This is **not** a good practice as you risk having your key compromised by untrusted clients.

When using Shared Key authentication, you will create a [connection string](#). The connection string is comprised of:

- The **DefaultEndpointsProtocol** - you can choose HTTP or HTTPS. However, using HTTPS is highly recommended.
- The **Account Name** - the name of your storage account
- The **Account Key** - On the [Azure Portal](#), navigate to your storage account and click the **Keys** icon to find this information.
- (Optional) **EndpointSuffix** - This is used for storage services in regions with different endpoint suffixes, such as Azure China or Azure Governance.

Here is an example of connection string using Shared Key authentication:

```
"DefaultEndpointsProtocol=https;AccountName=your_account_name_here;AccountKey=your_account_key_here"
```

Shared Access Signatures (SAS)

For a mobile application, the recommended method for authenticating a request by a client against the Azure Storage service is by using a Shared Access Signature (SAS). SAS allows you to grant a client access to a resource for a specified period of time, with a specified set of permissions. As the storage account owner, you'll need to generate a SAS for your mobile clients to consume. To generate the SAS, you'll probably want to write a separate service that generates the SAS to be distributed to your clients. For testing purposes, you can use the [Microsoft Azure Storage Explorer](#) or the [Azure Portal](#) to generate a SAS. When you create the SAS, you can specify the time interval over which the SAS is valid, and the permissions that the SAS grants to the client.

The following example shows how to use the Microsoft Azure Storage Explorer to generate a SAS.

1. If you haven't already, [Install the Microsoft Azure Storage Explorer](#)
2. Connect to your subscription.
3. Click on your Storage account and click on the "Actions" tab at the bottom left. Click "Get Shared Access Signature" to generate a "connection string" for your SAS.
4. Here is an example of a SAS connection string that grants read and write permissions at the service, container and object level for the blob service of the Storage account.

```
"SharedAccessSignature=sv=2015-04-05&ss=b&srt=sco&sp=rw&se=2016-07-21T18%3A00%3A00Z&sig=3ABdLOJZosCp0o491T%2BqZGKIhaffF1nlM3mzESDD3Gg%3D;BlobEndpoint=https://youraccount.blob.core.windows.net"
```

As you can see, when using a SAS, you're not exposing your account key in your application. You can learn more about SAS and best practices for using SAS by checking out [Shared Access Signatures: Understanding the SAS model](#).

Asynchronous Operations

NOTE

All methods that perform a request against the service are asynchronous operations. In the code samples, you'll find that these methods have a completion handler. Code inside the completion handler will run **after** the request is completed. Code after the completion handler will run **while** the request is being made.

Create a container

Every blob in Azure Storage must reside in a container. The following example shows how to create a container, called *newcontainer*, in your Storage account if it doesn't already exist. When choosing a name for your container, be mindful of the naming rules mentioned above.

```
-(void)createContainer{
    NSError *accountCreationError;

    // Create a storage account object from a connection string.
    AZSCloudStorageAccount *account = [AZSCloudStorageAccount
accountFromConnectionString:@"DefaultEndpointsProtocol=https;AccountName=your_account_name_here;AccountKey=your_account_key_here" error:&accountCreationError];

    if(accountCreationError){
        NSLog(@"Error in creating account.");
    }

    // Create a blob service client object.
    AZSCloudBlobClient *blobClient = [account getBlobClient];

    // Create a local container object.
    AZSCloudBlobContainer *blobContainer = [blobClient containerReferenceFromName:@"newcontainer"];

    // Create container in your Storage account if the container doesn't already exist
    [blobContainer createContainerIfNotExistsWithCompletionHandler:^(NSError *error, BOOL exists) {
        if (error){
            NSLog(@"Error in creating container.");
        }
    }];
}
```

You can confirm that this works by looking at the [Microsoft Azure Storage Explorer](#) and verifying that *newcontainer* is in the list of containers for your Storage account.

Set Container Permissions

A container's permissions are configured for **Private** access by default. However, containers provide a few different options for container access:

- **Private:** Container and blob data can be read by the account owner only.
- **Blob:** Blob data within this container can be read via anonymous request, but container data is not available. Clients

cannot enumerate blobs within the container via anonymous request.

- **Container:** Container and blob data can be read via anonymous request. Clients can enumerate blobs within the container via anonymous request, but cannot enumerate containers within the storage account.

The following example shows you how to create a container with **Container** access permissions which will allow public, read-only access for all users on the Internet:

```
-(void)createContainerWithPublicAccess{
    NSError *accountCreationError;

    // Create a storage account object from a connection string.
    AZSCloudStorageAccount *account = [AZSCloudStorageAccount
accountFromConnectionString:@"DefaultEndpointsProtocol=https;AccountName=your_account_name_here;AccountKey=your_account
_key_here" error:&accountCreationError];

    if(accountCreationError){
        NSLog(@"Error in creating account.");
    }

    // Create a blob service client object.
    AZSCloudBlobClient *blobClient = [account getBlobClient];

    // Create a local container object.
    AZSCloudBlobContainer *blobContainer = [blobClient containerReferenceFromName:@"containerpublic"];

    // Create container in your Storage account if the container doesn't already exist
    [blobContainer createContainerIfNotExistsWithAccessType:AZSContainerPublicAccessTypeContainer requestOptions:nil
operationContext:nil completionHandler:^(NSError *error, BOOL exists){
        if (error){
            NSLog(@"Error in creating container.");
        }
    }];
}
```

Upload a blob into a container

As mentioned in the [Blob service concepts](#) section, Blob Storage offers three different types of blobs: block blobs, append blobs, and page blobs. At this moment, the Azure Storage iOS library only supports block blobs. In the majority of cases, block blob is the recommended type to use.

The following example shows how to upload a block blob from an `NSString`. If a blob with the same name already exists in this container, the contents of this blob will be overwritten.

```

-(void)uploadBlobToContainer{
    NSError *accountCreationError;

    // Create a storage account object from a connection string.
    AZSCloudStorageAccount *account = [AZSCloudStorageAccount
accountFromConnectionString:@"DefaultEndpointsProtocol=https;AccountName=your_account_name_here;AccountKey=your_account
_key_here" error:&accountCreationError];

    if(accountCreationError){
        NSLog(@"Error in creating account.");
    }

    // Create a blob service client object.
    AZSCloudBlobClient *blobClient = [account getBlobClient];

    // Create a local container object.
    AZSCloudBlobContainer *blobContainer = [blobClient containerReferenceFromName:@"containerpublic"];

    [blobContainer createContainerIfNotExistsWithAccessType:AZSContainerPublicAccessTypeContainer requestOptions:nil
operationContext:nil completionHandler:^(NSError *error, BOOL exists)
    {
        if (error){
            NSLog(@"Error in creating container.");
        }
        else{
            // Create a local blob object
            AZSCloudBlockBlob *blockBlob = [blobContainer blockBlobReferenceFromName:@"sampleblob"];

            // Upload blob to Storage
            [blockBlob uploadFromText:@"This text will be uploaded to Blob Storage." completionHandler:^(NSError
*error) {
                if (error){
                    NSLog(@"Error in creating blob.");
                }
            }];
        }
    }];
}

```

You can confirm that this works by looking at the [Microsoft Azure Storage Explorer](#) and verifying that the container, *containerpublic*, contains the blob, *sampleblob*. In this sample, we used a public container so you can also verify that this worked by going to the blobs URL:

```
https://nameofyourstorageaccount.blob.core.windows.net/containerpublic/sampleblob
```

In addition to uploading a block blob from an `NSString`, similar methods exist for `NSData`, `NSInputStream` or a local file.

List the blobs in a container

The following example shows how to list all blobs in a container. When performing this operation, be mindful of the following parameters:

- **continuationToken** - The continuation token represents where the listing operation should start. If no token is provided, it will list blobs from the beginning. Any number of blobs can be listed, from zero up to a set maximum. Even if this method returns zero results, if `results.continuationToken` is not nil, there may be more blobs on the service that have not been listed.
- **prefix** - You can specify the prefix to use for blob listing. Only blobs that begin with this prefix will be listed.
- **useFlatBlobListing** - As mentioned in the [Naming and referencing containers and blobs](#) section, although the Blob service is a flat storage scheme, you can create a virtual hierarchy by naming blobs with path information. However, non-flat listing is currently not supported; this is coming soon. For now, this value should be **YES**.
- **blobListingDetails** - You can specify which items to include when listing blobs
 - `AZSBlobListingDetailsNone`: List only committed blobs, and do not return blob metadata.
 - `AZSBlobListingDetailsSnapshots`: List committed blobs and blob snapshots.
 - `AZSBlobListingDetailsMetadata`: Retrieve blob metadata for each blob returned in the listing.

- *AZSBlobListingDetailsUncommittedBlobs*: List committed and uncommitted blobs.
- *AZSBlobListingDetailsCopy*: Include copy properties in the listing.
- *AZSBlobListingDetailsAll*: List all available committed blobs, uncommitted blobs, and snapshots, and return all metadata and copy status for those blobs.
- **maxResults** - The maximum number of results to return for this operation. Use -1 to not set a limit.
- **completionHandler** - The block of code to execute with the results of the listing operation.

In this example, a helper method is used to recursively call the list blobs method every time a continuation token is returned.

```

-(void)listBlobsInContainer{
    NSError *accountCreationError;

    // Create a storage account object from a connection string.
    AZSCloudStorageAccount *account = [AZSCloudStorageAccount
accountFromConnectionString:@"DefaultEndpointsProtocol=https;AccountName=your_account_name_here;AccountKey=your_account
_key_here" error:&accountCreationError];

    if(accountCreationError){
        NSLog(@"Error in creating account.");
    }

    // Create a blob service client object.
    AZSCloudBlobClient *blobClient = [account getBlobClient];

    // Create a local container object.
    AZSCloudBlobContainer *blobContainer = [blobClient containerReferenceFromName:@"containerpublic"];

    //List all blobs in container
    [self listBlobsInContainerHelper:blobContainer continuationToken:nil prefix:nil
blobListingDetails:AZSBlobListingDetailsAll maxResults:-1 completionHandler:^(NSError *error) {
        if (error != nil){
            NSLog(@"Error in creating container.");
        }
    }];
}

//List blobs helper method
-(void)listBlobsInContainerHelper:(AZSCloudBlobContainer *)container continuationToken:(AZSContinuationToken
*)continuationToken prefix:(NSString *)prefix blobListingDetails:(AZSBlobListingDetails)blobListingDetails maxResults:
(NSUInteger)maxResults completionHandler:(void (^)(NSError *))completionHandler
{
    [container listBlobsSegmentedWithContinuationToken:continuationToken prefix:prefix useFlatBlobListing:YES
blobListingDetails/blobListingDetails maxResults:maxResults completionHandler:^(NSError *error, AZSblobResultSegment
*results) {
        if (error)
        {
            completionHandler(error);
        }
        else
        {
            for (int i = 0; i < results.blobs.count; i++) {
                NSLog(@"%@",[(AZSCloudBlockBlob *)results.blobs[i] blobName]);
            }
            if (results.continuationToken)
            {
                [self listBlobsInContainerHelper:container continuationToken:results.continuationToken prefix:prefix
blobListingDetails/blobListingDetails maxResults:maxResults completionHandler:completionHandler];
            }
            else
            {
                completionHandler(nil);
            }
        }
    }];
}
}

```

Download a blob

The following example shows how to download a blob to a NSString object.

```
- (void)downloadBlobToString{
    NSError *accountCreationError;

    // Create a storage account object from a connection string.
    AZSCloudStorageAccount *account = [AZSCloudStorageAccount
accountFromConnectionString:@"DefaultEndpointsProtocol=https;AccountName=your_account_name_here;AccountKey=your_account
_key_here" error:&accountCreationError];

    if(accountCreationError){
        NSLog(@"Error in creating account.");
    }

    // Create a blob service client object.
    AZSCloudBlobClient *blobClient = [account getBlobClient];

    // Create a local container object.
    AZSCloudBlobContainer *blobContainer = [blobClient containerReferenceFromName:@"containerpublic"];

    // Create a local blob object
    AZSCloudBlockBlob *blockBlob = [blobContainer blockBlobReferenceFromName:@"sampleblob"];

    // Download blob
    [blockBlob downloadToTextWithCompletionHandler:^(NSError *error, NSString *text) {
        if (error) {
            NSLog(@"Error in downloading blob");
        }
        else{
            NSLog(@"%@",text);
        }
    }];
}
```

Delete a blob

The following example shows how to delete a blob.

```
- (void)deleteBlob{
    NSError *accountCreationError;

    // Create a storage account object from a connection string.
    AZSCloudStorageAccount *account = [AZSCloudStorageAccount
accountFromConnectionString:@"DefaultEndpointsProtocol=https;AccountName=your_account_name_here;AccountKey=your_account
_key_here" error:&accountCreationError];

    if(accountCreationError){
        NSLog(@"Error in creating account.");
    }

    // Create a blob service client object.
    AZSCloudBlobClient *blobClient = [account getBlobClient];

    // Create a local container object.
    AZSCloudBlobContainer *blobContainer = [blobClient containerReferenceFromName:@"containerpublic"];

    // Create a local blob object
    AZSCloudBlockBlob *blockBlob = [blobContainer blockBlobReferenceFromName:@"sampleblob1"];

    // Delete blob
    [blockBlob deleteWithCompletionHandler:^(NSError *error) {
        if (error) {
            NSLog(@"Error in deleting blob.");
        }
    }];
}
```

Delete a blob container

The following example shows how to delete a container.

```
-(void)deleteContainer{
    NSError *accountCreationError;

    // Create a storage account object from a connection string.
    AZSCloudStorageAccount *account = [AZSCloudStorageAccount
accountFromConnectionString:@"DefaultEndpointsProtocol=https;AccountName=your_account_name_here;AccountKey=your_account
_key_here" error:&accountCreationError];

    if(accountCreationError){
        NSLog(@"Error in creating account.");
    }

    // Create a blob service client object.
    AZSCloudBlobClient *blobClient = [account getBlobClient];

    // Create a local container object.
    AZSCloudBlobContainer *blobContainer = [blobClient containerReferenceFromName:@"containerpublic"];

    // Delete container
    [blobContainer deleteContainerIfExistsWithCompletionHandler:^(NSError *error, BOOL success) {
        if(error){
            NSLog(@"Error in deleting container");
        }
    }];
}
```

Next steps

Now that you've learned how to use Blob Storage from iOS, follow these links to learn more about the iOS library and the Storage service.

- [Azure Storage Client Library for iOS](#)
- [Azure Storage iOS Reference Documentation](#)
- [Azure Storage Services REST API](#)
- [Azure Storage Team Blog](#)

If you have questions regarding this library feel free to post to our [MSDN Azure forum](#) or [Stack Overflow](#). If you have feature suggestions for Azure Storage, please post to [Azure Storage Feedback](#).

How to use Blob Storage from Xamarin

1/17/2017 • 8 min to read • [Edit on GitHub](#)

Overview

Xamarin enables developers to use a shared C# codebase to create iOS, Android, and Windows Store apps with their native user interfaces. This tutorial shows you how to use Azure Blob storage with a Xamarin application. If you'd like to learn more about Azure Storage, before diving into the code, see [Introduction to Microsoft Azure Storage](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Configure your application to access Azure Storage

There are two ways to authenticate your application to access Storage services:

- Shared Key: Use Shared Key for testing purposes only
- Shared Access Signature (SAS): Use SAS for production applications

Shared Key

Shared Key authentication means that your application will use your account name and account key to access Storage services. For the purposes of quickly showing how to use this library, we will be using Shared Key authentication in this getting started.

WARNING

Only use Shared Key authentication for testing purposes! Your account name and account key, which give full read/write access to the associated Storage account, will be distributed to every person that downloads your app. This is **not** a good practice as you risk having your key compromised by untrusted clients.

When using Shared Key authentication, you will create a [connection string](#). The connection string is comprised of:

- The **DefaultEndpointsProtocol** - you can choose HTTP or HTTPS. However, using HTTPS is highly recommended.
- The **Account Name** - the name of your storage account
- The **Account Key** - On the [Azure Portal](#), navigate to your storage account and click the **Keys** icon to find this information.
- (Optional) **EndpointSuffix** - This is used for storage services in regions with different endpoint suffixes, such as Azure China or Azure Governance.

Here is an example of connection string using Shared Key authentication:

```
"DefaultEndpointsProtocol=https;AccountName=your_account_name_here;AccountKey=your_account_key_here"
```

Shared Access Signatures (SAS)

For a mobile application, the recommended method for authenticating a request by a client against the Azure Storage service is by using a Shared Access Signature (SAS). SAS allows you to grant a client access to a resource for a specified period of time, with a specified set of permissions. As the storage account owner, you'll need to generate a SAS for your

mobile clients to consume. To generate the SAS, you'll probably want to write a separate service that generates the SAS to be distributed to your clients. For testing purposes, you can use the [Microsoft Azure Storage Explorer](#) or the [Azure Portal](#) to generate a SAS. When you create the SAS, you can specify the time interval over which the SAS is valid, and the permissions that the SAS grants to the client.

The following example shows how to use the Microsoft Azure Storage Explorer to generate a SAS.

1. If you haven't already, [Install the Microsoft Azure Storage Explorer](#)
2. Connect to your subscription.
3. Click on your Storage account and click on the "Actions" tab at the bottom left. Click "Get Shared Access Signature" to generate a "connection string" for your SAS.
4. Here is an example of a SAS connection string that grants read and write permissions at the service, container and object level for the blob service of the Storage account.

```
"SharedAccessSignature=sv=2015-04-05&ss=b&srt=sco&sp=rw&se=2016-07-21T18%3A00%3A00Z&sig=3ABdLOJZosCp0o491T%2BqZGKIhaff1nlM3MzESDDD3Gg%3D;BlobEndpoint=https://youraccount.blob.core.windows.net"
```

As you can see, when using a SAS, you're not exposing your account key in your application. You can learn more about SAS and best practices for using SAS by checking out [Shared Access Signatures: Understanding the SAS model](#).

Create a new Xamarin Application

For this getting started, we'll be creating an app that targets Android, iOS, and Windows. This app will simply create a container and upload a blob into this container. We'll be using Visual Studio on Windows for this getting started, but the same learnings can be applied when creating an app using Xamarin Studio on Mac OS.

Follow these steps to create your application:

1. If you haven't already, download and install [Xamarin for Visual Studio](#).
2. Open Visual Studio, and create a Blank App (Native Shared): **File > New > Project > Cross-Platform > Blank App(Native Shared)**.
3. Right-click your solution in the Solution Explorer pane and select **Manage NuGet Packages for Solution**. Search for **WindowsAzure.Storage** and install the latest stable version to all projects in your solution.
4. Build and run your project.

You should now have an application that allows you to click a button which increments a counter.

NOTE

The Azure Storage Client Library for Xamarin currently supports the following project types: Native Shared, Xamarin.Forms Shared, Xamarin.Android, and Xamarin.iOS.

Create container and upload blob

Next, you'll add some code to the shared class `MyClass.cs` that creates a container and uploads a blob into this container. `MyClass.cs` should look like the following:

```

using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Blob;
using System.Threading.Tasks;

namespace XamarinApp
{
    public class MyClass
    {
        public MyClass ()
        {

        }

        public static async Task createContainerAndUpload()
        {
            // Retrieve storage account from connection string.
            CloudStorageAccount storageAccount =
CloudStorageAccount.Parse("DefaultEndpointsProtocol=https;AccountName=your_account_name_here;AccountKey=your_account_ke
y_here");

            // Create the blob client.
            CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

            // Retrieve reference to a previously created container.
            CloudBlobContainer container = blobClient.GetContainerReference("mycontainer");

            // Create the container if it doesn't already exist.
            await container.CreateIfNotExistsAsync();

            // Retrieve reference to a blob named "myblob".
            CloudBlockBlob blockBlob = container.GetBlockBlobReference("myblob");

            // Create the "myblob" blob with the text "Hello, world!"
            await blockBlob.UploadTextAsync("Hello, world!");
        }
    }
}

```

Make sure to replace "your_account_name_here" and "your_account_key_here" with your actual account name and account key. You can then use this shared class in your iOS, Android, and Windows Phone application. You can simply add `MyClass.createContainerAndUpload()` to each project. For example:

XamarinApp.Droid > MainActivity.cs

```

using Android.App;
using Android.Widget;
using Android.OS;

namespace XamarinApp.Droid
{
    [Activity (Label = "XamarinApp.Droid", MainLauncher = true, Icon = "@drawable/icon")]
    public class MainActivity : Activity
    {
        int count = 1;

        protected override async void OnCreate (Bundle bundle)
        {
            base.OnCreate (bundle);

            // Set our view from the "main" layout resource
            SetContentView (Resource.Layout.Main);

            // Get our button from the layout resource,
            // and attach an event to it
            Button button = FindViewById<Button> (Resource.Id.myButton);

            button.Click += delegate {
                button.Text = string.Format ("{0} clicks!", count++);
            };

            await MyClass.createContainerAndUpload();
        }
    }
}

```

XamarinApp.iOS > ViewController.cs

```

using System;
using UIKit;

namespace XamarinApp.iOS
{
    public partial class ViewController : UIViewController
    {
        int count = 1;

        public ViewController (IntPtr handle) : base (handle)
        {
        }

        public override async void ViewDidLoad ()
        {
            base.ViewDidLoad ();
            // Perform any additional setup after loading the view, typically from a nib.
            Button.AccessibilityIdentifier = "myButton";
            Button.TouchUpInside += delegate {
                var title = string.Format ("{0} clicks!", count++);
                ButtonSetTitle (title, UIControlState.Normal);
            };

            await MyClass.createContainerAndUpload();
        }

        public override void DidReceiveMemoryWarning ()
        {
            base.DidReceiveMemoryWarning ();
            // Release any cached data, images, etc that aren't in use.
        }
    }
}

```

XamarinApp.WinPhone > MainPage.xaml > MainPage.xaml.cs

```

using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at http://go.microsoft.com/fwlink/?LinkId=391641

namespace XamarinApp.WinPhone
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        int count = 1;

        public MainPage()
        {
            this.InitializeComponent();

            this.NavigationCacheMode = NavigationCacheMode.Required;
        }

        /// <summary>
        /// Invoked when this page is about to be displayed in a Frame.
        /// </summary>
        /// <param name="e">Event data that describes how this page was reached.
        /// This parameter is typically used to configure the page.</param>
        protected override async void OnNavigatedTo(NavigationEventArgs e)
        {
            // TODO: Prepare page for display here.

            // TODO: If your application contains multiple pages, ensure that you are
            // handling the hardware Back button by registering for the
            // Windows.Phone.UI.Input.HardwareButtons.BackPressed event.
            // If you are using the NavigationHelper provided by some templates,
            // this event is handled for you.
            Button.Click += delegate {
                var title = string.Format("{0} clicks!", count++);
                Button.Content = title;
            };

            await MyClass.createContainerAndUpload();
        }
    }
}

```

Run the application

You can now run this application in an Android or Windows Phone emulator. You can also run this application in an iOS emulator, but this will require a Mac. For specific instructions on how to do this, please read the documentation for [connecting Visual Studio to a Mac](#)

Once you run your app, it will create the container `mycontainer` in your Storage account. It should contain the blob, `myblob`, which has the text, `Hello, world!`. You can verify this by using the [Microsoft Azure Storage Explorer](#).

Next steps

In this getting started, you learned how to create a cross-platform application in Xamarin that uses Azure Storage. This getting started specifically focused on one scenario in Blob Storage. However, you can do a lot more with, not only Blob Storage, but also with Table, File, and Queue Storage. Please check out the following articles to learn more:

- [Get started with Azure Blob Storage using .NET](#)
- [Get started with Azure Table Storage using .NET](#)
- [Get started with Azure Queue Storage using .NET](#)
- [Get started with Azure File Storage on Windows](#)

TIP**Manage Azure Blob Storage resources with Microsoft Azure Storage Explorer**

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to [manage Azure Blob Storage resources](#).

Using Microsoft Azure Storage Explorer, you can visually create, read, update, and delete blob containers and blobs, as well as manage access to your blobs containers and blobs.

Get started with Azure Queue storage using .NET

1/17/2017 • 13 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

Azure Queue storage provides cloud messaging between application components. In designing applications for scale, application components are often decoupled, so that they can scale independently. Queue storage delivers asynchronous messaging for communication between application components, whether they are running in the cloud, on the desktop, on an on-premises server, or on a mobile device. Queue storage also supports managing asynchronous tasks and building process work flows.

About this tutorial

This tutorial shows how to write .NET code for some common scenarios using Azure Queue storage. Scenarios covered include creating and deleting queues and adding, reading, and deleting queue messages.

Estimated time to complete: 45 minutes

Prerequisites:

- [Microsoft Visual Studio](#)
- [Azure Storage Client Library for .NET](#)
- [Azure Configuration Manager for .NET](#)
- An [Azure storage account](#)

NOTE

We recommend that you use the latest version of the Azure Storage Client Library for .NET to complete this tutorial. The latest version of the library is 7.x, available for download on [Nuget](#). The source for the client library is available on [GitHub](#).

If you are using the storage emulator, note that version 7.x of the client library requires at least version 4.3 of the storage emulator

What is Queue Storage?

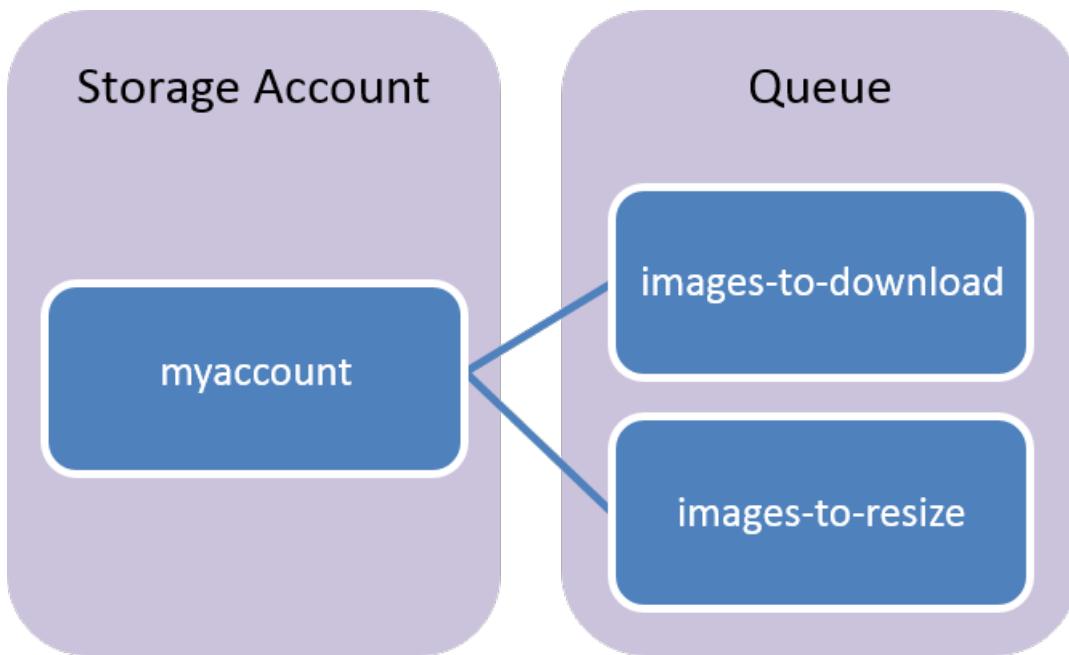
Azure Queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS. A single queue message can be up to 64 KB in size, and a queue can contain millions of messages, up to the total capacity limit of a storage account.

Common uses of Queue storage include:

- Creating a backlog of work to process asynchronously
- Passing messages from an Azure web role to an Azure worker role

Queue Service Concepts

The Queue service contains the following components:



- **URL format:** Queues are addressable using the following URL format:

```
http://<storage account>.queue.core.windows.net/<queue>
```

The following URL addresses a queue in the diagram:

```
http://myaccount.queue.core.windows.net/images-to-download
```

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. The maximum time that a message can remain in the queue is 7 days.

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

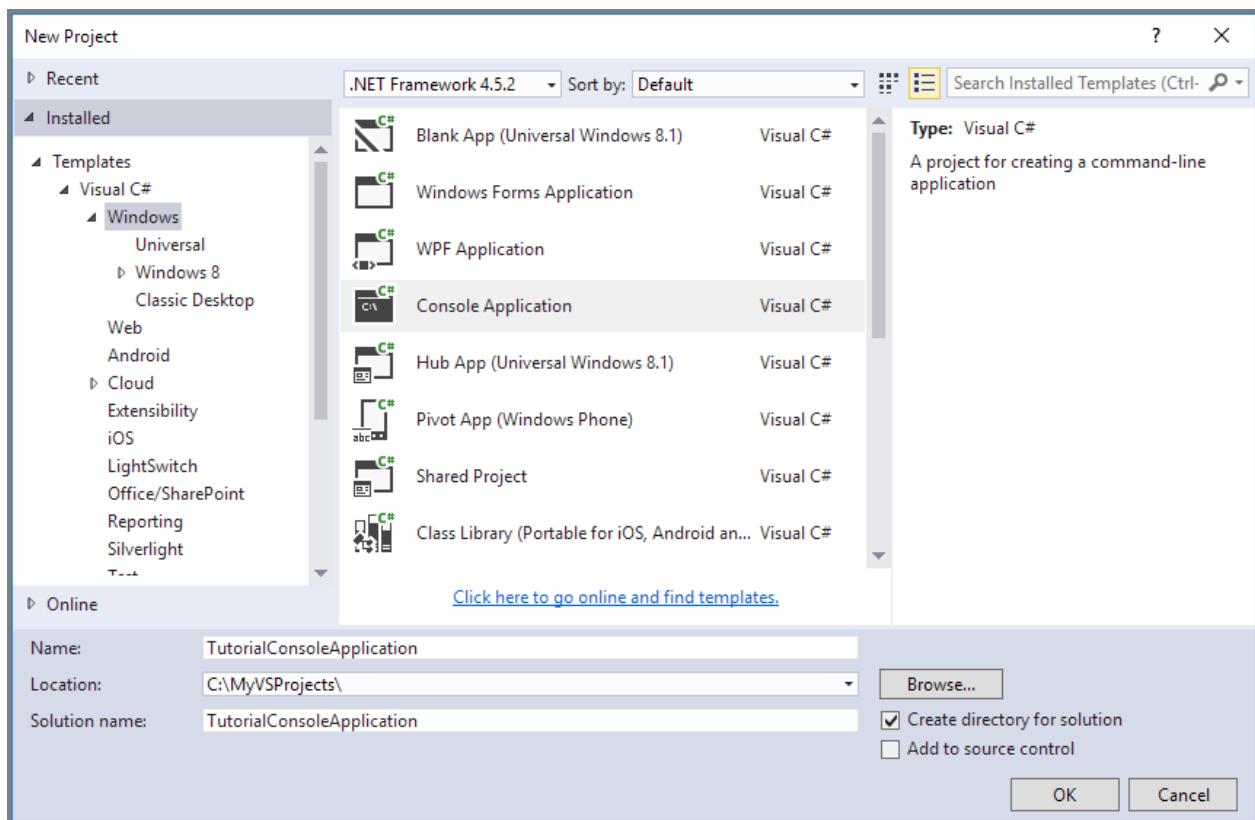
If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Set up your development environment

Next, set up your development environment in Visual Studio so that you are ready to try the code examples provided in this guide.

Create a Windows console application project

In Visual Studio, create a new Windows console application, as shown:



All of the code examples in this tutorial can be added to the **Main()** method in `program.cs` in your console application.

Note that you can use the Azure Storage Client Library from any type of .NET application, including an Azure cloud service, an Azure web app, a desktop application, or a mobile application. In this guide, we use a console application for simplicity.

Use NuGet to install the required packages

There are two packages that you'll need to install to your project to complete this tutorial:

- [Microsoft Azure Storage Client Library for .NET](#): This package provides programmatic access to data resources in your storage account.
- [Microsoft Azure Configuration Manager library for .NET](#): This package provides a class for parsing a connection string from a configuration file, regardless of where your application is running.

You can use NuGet to obtain both packages. Follow these steps:

1. Right-click your project in **Solution Explorer** and choose **Manage NuGet Packages**.
2. Search online for "WindowsAzure.Storage" and click **Install** to install the Storage Client Library and its dependencies.
3. Search online for "ConfigurationManager" and click **Install** to install the Azure Configuration Manager.

NOTE

The Storage Client Library package is also included in the [Azure SDK for .NET](#). However, we recommend that you also install the Storage Client Library from NuGet to ensure that you always have the latest version of the client library.

The ODataLib dependencies in the Storage Client Library for .NET are resolved through the ODataLib (version 5.0.2 and greater) packages available through NuGet, and not through WCF Data Services. The ODataLib libraries can be downloaded directly or referenced by your code project through NuGet. The specific ODataLib packages used by the Storage Client Library are [OData](#), [Edm](#), and [Spatial](#). While these libraries are used by the Azure Table storage classes, they are required dependencies for programming with the Storage Client Library.

Determine your target environment

You have two environment options for running the examples in this guide:

- You can run your code against an Azure Storage account in the cloud.
- You can run your code against the Azure storage emulator. The storage emulator is a local environment that emulates an Azure Storage account in the cloud. The emulator is a free option for testing and debugging your code while your application is under development. The emulator uses a well-known account and key. For more details, see [Use the Azure Storage Emulator for Development and Testing](#)

If you are targeting a storage account in the cloud, copy the primary access key for your storage account from the Azure Portal. For more information, see [View and copy storage access keys](#).

NOTE

You can target the storage emulator to avoid incurring any costs associated with Azure Storage. However, if you do choose to target an Azure storage account in the cloud, costs for performing this tutorial will be negligible.

Configure your storage connection string

The Azure Storage Client Library for .NET supports using a storage connection string to configure endpoints and credentials for accessing storage services. The best way to maintain your storage connection string is in a configuration file.

For more information about connection strings, see [Configure a Connection String to Azure Storage](#).

NOTE

Your storage account key is similar to the root password for your storage account. Always be careful to protect your storage account key. Avoid distributing it to other users, hard-coding it, or saving it in a plain-text file that is accessible to others. Regenerate your key using the Azure Portal if you believe it may have been compromised.

To configure your connection string, open the `app.config` file from Solution Explorer in Visual Studio. Add the contents of the `<appSettings>` element shown below. Replace `account-name` with the name of your storage account, and `account-key` with your account access key:

```
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
  </startup>
  <appSettings>
    <add key="StorageConnectionString" value="DefaultEndpointsProtocol=https;AccountName=account-
name;AccountKey=account-key" />
  </appSettings>
</configuration>
```

For example, your configuration setting will be similar to:

```
<add key="StorageConnectionString"
value="DefaultEndpointsProtocol=https;AccountName=storagesample;AccountKey=nYV0gln6fT7mvY+rxu2iWAEyzPKITGkhM88J
8HUoyofvK7C6fHcZc2kRZp6cKgYRUM74lHI84L50Iau1+9hPjB==" />
```

To target the storage emulator, you can use a shortcut that maps to the well-known account name and key. In that case, your connection string setting will be:

```
<add key="StorageConnectionString" value="UseDevelopmentStorage=true;" />
```

Add namespace declarations

Add the following `using` statements to the top of the `program.cs` file:

```
using Microsoft.Azure; // Namespace for CloudConfigurationManager
using Microsoft.WindowsAzure.Storage; // Namespace for CloudStorageAccount
using Microsoft.WindowsAzure.Storage.Queue; // Namespace for Queue storage types
```

Parse the connection string

The [Microsoft Azure Configuration Manager Library for .NET](#) provides a class for parsing a connection string from a configuration file. The [CloudConfigurationManager](#) class parses configuration settings regardless of whether the client application is running on the desktop, on a mobile device, in an Azure virtual machine, or in an Azure cloud service.

To reference the `CloudConfigurationManager` package, add the following `using` directive:

```
using Microsoft.Azure; //Namespace for CloudConfigurationManager
```

Here's an example that shows how to retrieve a connection string from a configuration file:

```
// Parse the connection string and return a reference to the storage account.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));
```

Using the Azure Configuration Manager is optional. You can also use an API like the .NET Framework's [ConfigurationManager](#) class.

Create the Queue service client

The **CloudQueueClient** class enables you to retrieve queues stored in Queue storage. Here's one way to create the service client:

```
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();
```

Now you are ready to write code that reads data from and writes data to Queue storage.

Create a queue

This example shows how to create a queue if it does not already exist:

```
// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a container.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Create the queue if it doesn't already exist
queue.CreateIfNotExists();
```

Insert a message into a queue

To insert a message into an existing queue, first create a new **CloudQueueMessage**. Next, call the **AddMessage** method. A **CloudQueueMessage** can be created from either a string (in UTF-8 format) or a **byte** array. Here is code which creates a queue (if it doesn't exist) and inserts the message 'Hello, World':

```

// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Create the queue if it doesn't already exist.
queue.CreateIfNotExists();

// Create a message and add it to the queue.
CloudQueueMessage message = new CloudQueueMessage("Hello, World");
queue.AddMessage(message);

```

Peek at the next message

You can peek at the message in the front of a queue without removing it from the queue by calling the [PeekMessage](#) method.

```

// Retrieve storage account from connection string
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Peek at the next message
CloudQueueMessage peekedMessage = queue.PeekMessage();

// Display message.
Console.WriteLine(peekedMessageAsString);

```

Change the contents of a queued message

You can change the contents of a message in-place in the queue. If the message represents a work task, you could use this feature to update the status of the work task. The following code updates the queue message with new contents, and sets the visibility timeout to extend another 60 seconds. This saves the state of work associated with the message, and gives the client another minute to continue working on the message. You could use this technique to track multi-step workflows on queue messages, without having to start over from the beginning if a processing step fails due to hardware or software failure. Typically, you would keep a retry count as well, and if the message is retried more than n times, you would delete it. This protects against a message that triggers an application error each time it is processed.

```

// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Get the message from the queue and update the message contents.
CloudQueueMessage message = queue.GetMessage();
message.SetMessageContent("Updated contents.");
queue.UpdateMessage(message,
    TimeSpan.FromSeconds(60.0), // Make it invisible for another 60 seconds.
    MessageUpdateFields.Content | MessageUpdateFields.Visibility);

```

De-queue the next message

Your code de-queues a message from a queue in two steps. When you call **GetMessage**, you get the next message in a queue. A message returned from **GetMessage** becomes invisible to any other code reading messages from this queue. By default, this message stays invisible for 30 seconds. To finish removing the message from the queue, you must also call **DeleteMessage**. This two-step process of removing a message assures that if your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls **DeleteMessage** right after the message has been processed.

```

// Retrieve storage account from connection string
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Get the next message
CloudQueueMessage retrievedMessage = queue.GetMessage();

//Process the message in less than 30 seconds, and then delete the message
queue.DeleteMessage(retrievedMessage);

```

Use Async-Await pattern with common Queue storage APIs

This example shows how to use the Async-Await pattern with common Queue storage APIs. The sample calls the asynchronous version of each of the given methods, as indicated by the **Async** suffix of each method. When an **async** method is used, the **async**-await pattern suspends local execution until the call completes. This behavior allows the current thread to do other work, which helps avoid performance bottlenecks and improves the overall responsiveness of your application. For more details on using the **Async-Await** pattern in .NET see [Async and Await \(C# and Visual Basic\)](#)

```

// Create the queue if it doesn't already exist
if(await queue.CreateIfNotExistsAsync())
{
    Console.WriteLine("Queue '{0}' Created", queue.Name);
}
else
{
    Console.WriteLine("Queue '{0}' Exists", queue.Name);
}

// Create a message to put in the queue
CloudQueueMessage cloudQueueMessage = new CloudQueueMessage("My message");

// Async enqueue the message
await queue.AddMessageAsync(cloudQueueMessage);
Console.WriteLine("Message added");

// Async dequeue the message
CloudQueueMessage retrievedMessage = await queue.GetMessageAsync();
Console.WriteLine("Retrieved message with content '{0}'", retrievedMessageAsString);

// Async delete the message
await queue.DeleteMessageAsync(retrievedMessage);
Console.WriteLine("Deleted message");

```

Leverage additional options for de-queuing messages

There are two ways you can customize message retrieval from a queue. First, you can get a batch of messages (up to 32). Second, you can set a longer or shorter invisibility timeout, allowing your code more or less time to fully process each message. The following code example uses the **GetMessages** method to get 20 messages in one call. Then it processes each message using a **foreach** loop. It also sets the invisibility timeout to five minutes for each message. Note that the 5 minutes starts for all messages at the same time, so after 5 minutes have passed since the call to **GetMessages**, any messages which have not been deleted will become visible again.

```

// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

foreach (CloudQueueMessage message in queue.GetMessages(20, TimeSpan.FromMinutes(5)))
{
    // Process all messages in less than 5 minutes, deleting each message after processing.
    queue.DeleteMessage(message);
}

```

Get the queue length

You can get an estimate of the number of messages in a queue. The **FetchAttributes** method asks the Queue service to retrieve the queue attributes, including the message count. The **ApproximateMessageCount** property returns the last value retrieved by the **FetchAttributes** method, without calling the Queue service.

```

// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Fetch the queue attributes.
queue.FetchAttributes();

// Retrieve the cached approximate message count.
int? cachedMessageCount = queue.ApproximateMessageCount;

// Display number of messages.
Console.WriteLine("Number of messages in queue: " + cachedMessageCount);

```

Delete a queue

To delete a queue and all the messages contained in it, call the **Delete** method on the queue object.

```

// Retrieve storage account from connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the queue client.
CloudQueueClient queueClient = storageAccount.CreateCloudQueueClient();

// Retrieve a reference to a queue.
CloudQueue queue = queueClient.GetQueueReference("myqueue");

// Delete the queue.
queue.Delete();

```

Next steps

Now that you've learned the basics of Queue storage, follow these links to learn about more complex storage tasks.

- View the Queue service reference documentation for complete details about available APIs:
 - [Storage Client Library for .NET reference](#)
 - [REST API reference](#)
- Learn how to simplify the code you write to work with Azure Storage by using the [Azure WebJobs SDK](#).
- View more feature guides to learn about additional options for storing data in Azure.
 - [Get started with Azure Table storage using .NET](#) to store structured data.
 - [Get started with Azure Blob storage using .NET](#) to store unstructured data.
 - [Connect to SQL Database by using .NET \(C#\)](#) to store relational data.

How to use Queue storage from Java

1/17/2017 • 11 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide will show you how to perform common scenarios using the Azure Queue storage service. The samples are written in Java and use the [Azure Storage SDK for Java](#). The scenarios covered include **inserting**, **peeking**, **getting**, and **deleting** queue messages, as well as **creating** and **deleting** queues. For more information on queues, see the [Next steps](#) section.

Note: An SDK is available for developers who are using Azure Storage on Android devices. For more information, see the [Azure Storage SDK for Android](#).

What is Queue Storage?

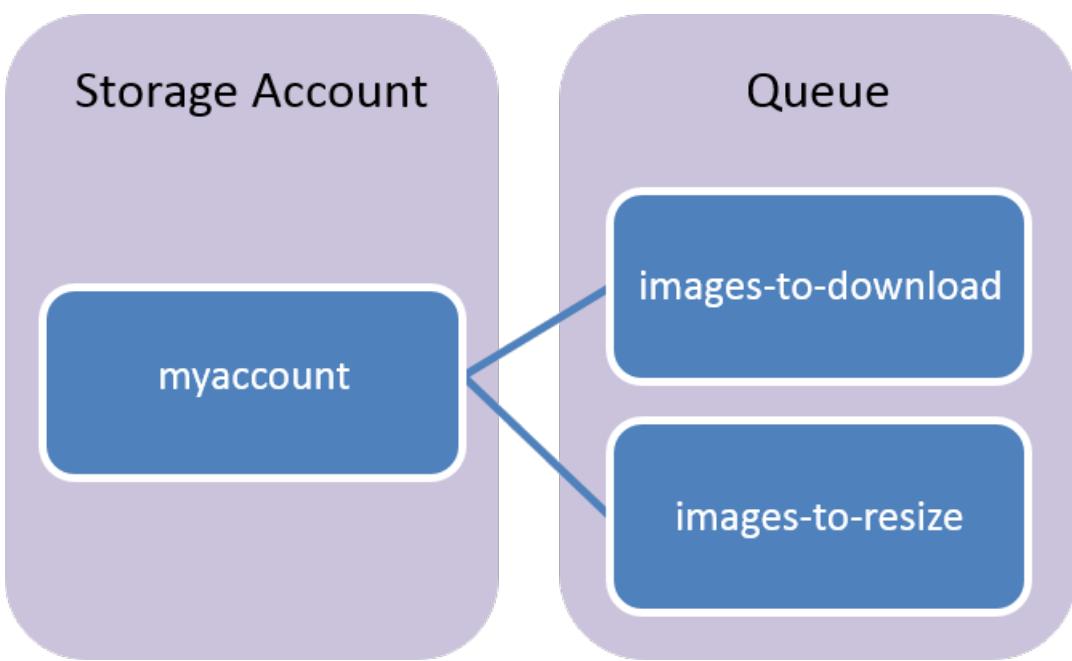
Azure Queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS. A single queue message can be up to 64 KB in size, and a queue can contain millions of messages, up to the total capacity limit of a storage account.

Common uses of Queue storage include:

- Creating a backlog of work to process asynchronously
- Passing messages from an Azure web role to an Azure worker role

Queue Service Concepts

The Queue service contains the following components:



- **URL format:** Queues are addressable using the following URL format:

```
http://<storage account>.queue.core.windows.net/<queue>
```

The following URL addresses a queue in the diagram:

```
http://myaccount.queue.core.windows.net/images-to-download
```

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. The maximum time that a message can remain in the queue is 7 days.

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a Java application

In this guide, you will use storage features which can be run within a Java application locally, or in code running within a web role or worker role in Azure.

To do so, you will need to install the Java Development Kit (JDK) and create an Azure storage account in your Azure subscription. Once you have done so, you will need to verify that your development system meets the minimum requirements and dependencies which are listed in the [Azure Storage SDK for Java](#) repository on GitHub. If your system meets those requirements, you can follow the instructions for downloading and installing the Azure Storage Libraries for Java on your system from that repository. Once you have completed those tasks, you will be able to create a Java application which uses the examples in this article.

Configure your application to access queue storage

Add the following import statements to the top of the Java file where you want to use Azure storage APIs to access queues:

```
// Include the following imports to use queue APIs.  
import com.microsoft.azure.storage.*;  
import com.microsoft.azure.storage.queue.*;
```

Setup an Azure storage connection string

An Azure storage client uses a storage connection string to store endpoints and credentials for accessing data management services. When running in a client application, you must provide the storage connection string in the following format, using the name of your storage account and the Primary access key for the storage account listed in the [Azure Portal](#) for the *AccountName* and *AccountKey* values. This example shows how you can declare a static field to hold the connection string:

```
// Define the connection-string with your values.  
public static final String storageConnectionString =  
    "DefaultEndpointsProtocol=http;" +  
    "AccountName=your_storage_account;" +  
    "AccountKey=your_storage_account_key";
```

In an application running within a role in Microsoft Azure, this string can be stored in the service configuration file, *ServiceConfiguration.cscfg*, and can be accessed with a call to the **RoleEnvironment.getConfigurationSettings** method. Here's an example of getting the connection string from a **Setting** element named *StorageConnectionString* in the service configuration file:

```
// Retrieve storage account from connection-string.  
String storageConnectionString =  
    RoleEnvironment.getConfigurationSettings().get("StorageConnectionString");
```

The following samples assume that you have used one of these two methods to get the storage connection string.

How to: Create a queue

A **CloudQueueClient** object lets you get reference objects for queues. The following code creates a **CloudQueueClient** object. (Note: There are additional ways to create **CloudStorageAccount** objects; for more information, see [CloudStorageAccount](#) in the [Azure Storage Client SDK Reference](#).)

Use the **CloudQueueClient** object to get a reference to the queue you want to use. You can create the queue if it doesn't exist.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the queue client.
    CloudQueueClient queueClient = storageAccount.createCloudQueueClient();

    // Retrieve a reference to a queue.
    CloudQueue queue = queueClient.getQueueReference("myqueue");

    // Create the queue if it doesn't already exist.
    queue.createIfNotExists();
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Add a message to a queue

To insert a message into an existing queue, first create a new **CloudQueueMessage**. Next, call the **addMessage** method. A **CloudQueueMessage** can be created from either a string (in UTF-8 format) or a byte array. Here is code which creates a queue (if it doesn't exist) and inserts the message "Hello, World".

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the queue client.
    CloudQueueClient queueClient = storageAccount.createCloudQueueClient();

    // Retrieve a reference to a queue.
    CloudQueue queue = queueClient.getQueueReference("myqueue");

    // Create the queue if it doesn't already exist.
    queue.createIfNotExists();

    // Create a message and add it to the queue.
    CloudQueueMessage message = new CloudQueueMessage("Hello, World");
    queue.addMessage(message);
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Peek at the next message

You can peek at the message in the front of a queue without removing it from the queue by calling **peekMessage**.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the queue client.
    CloudQueueClient queueClient = storageAccount.createCloudQueueClient();

    // Retrieve a reference to a queue.
    CloudQueue queue = queueClient.getQueueReference("myqueue");

    // Peek at the next message.
    CloudQueueMessage peekedMessage = queue.peekMessage();

    // Output the message value.
    if (peekedMessage != null)
    {
        System.out.println(peekedMessage.getMessageContentAsString());
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Change the contents of a queued message

You can change the contents of a message in-place in the queue. If the message represents a work task, you could use this feature to update the status of the work task. The following code updates the queue message with new contents, and sets the visibility timeout to extend another 60 seconds. This saves the state of work associated with the message, and gives the client another minute to continue working on the message. You could use this technique to track multi-step workflows on queue messages, without having to start over from the beginning if a processing step fails due to hardware or software failure. Typically, you would keep a retry count as well, and if the message is retried more than n times, you would delete it. This protects against a message that triggers an application error each time it is processed.

The following code sample searches through the queue of messages, locates the first message that matches "Hello, World" for the content, then modifies the message content and exits.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the queue client.
    CloudQueueClient queueClient = storageAccount.createCloudQueueClient();

    // Retrieve a reference to a queue.
    CloudQueue queue = queueClient.getQueueReference("myqueue");

    // The maximum number of messages that can be retrieved is 32.
    final int MAX_NUMBER_OF_MESSAGES_TO_PEEK = 32;

    // Loop through the messages in the queue.
    for (CloudQueueMessage message : queue.retrieveMessages(MAX_NUMBER_OF_MESSAGES_TO_PEEK,1,null,null))
    {
        // Check for a specific string.
        if (message.getMessageContentAsString().equals("Hello, World"))
        {
            // Modify the content of the first matching message.
            message.setMessageContent("Updated contents.");
            // Set it to be visible in 30 seconds.
            EnumSet<MessageUpdateFields> updateFields =
                EnumSet.of(MessageUpdateFields.CONTENT,
                           MessageUpdateFields.VISIBILITY);
            // Update the message.
            queue.updateMessage(message, 30, updateFields, null, null);
            break;
        }
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

Alternatively, the following code sample updates just the first visible message on the queue.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the queue client.
    CloudQueueClient queueClient = storageAccount.createCloudQueueClient();

    // Retrieve a reference to a queue.
    CloudQueue queue = queueClient.getQueueReference("myqueue");

    // Retrieve the first visible message in the queue.
    CloudQueueMessage message = queue.retrieveMessage();

    if (message != null)
    {
        // Modify the message content.
        message.setMessageContent("Updated contents.");
        // Set it to be visible in 60 seconds.
        EnumSet<MessageUpdateFields> updateFields =
            EnumSet.of(MessageUpdateFields.CONTENT,
                       MessageUpdateFields.VISIBILITY);
        // Update the message.
        queue.updateMessage(message, 60, updateFields, null, null);
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Get the queue length

You can get an estimate of the number of messages in a queue. The **downloadAttributes** method asks the Queue service for several current values, including a count of how many messages are in a queue. The count is only approximate because messages can be added or removed after the Queue service responds to your request. The **getApproximateMessageCount** method returns the last value retrieved by the call to **downloadAttributes**, without calling the Queue service.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the queue client.
    CloudQueueClient queueClient = storageAccount.createCloudQueueClient();

    // Retrieve a reference to a queue.
    CloudQueue queue = queueClient.getQueueReference("myqueue");

    // Download the approximate message count from the server.
    queue.downloadAttributes();

    // Retrieve the newly cached approximate message count.
    long cachedMessageCount = queue.getApproximateMessageCount();

    // Display the queue length.
    System.out.println(String.format("Queue length: %d", cachedMessageCount));
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Dequeue the next message

Your code dequeues a message from a queue in two steps. When you call **retrieveMessage**, you get the next message in a queue. A message returned from **retrieveMessage** becomes invisible to any other code reading messages from this queue. By default, this message stays invisible for 30 seconds. To finish removing the message from the queue, you must also call **deleteMessage**. This two-step process of removing a message assures that if your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls **deleteMessage** right after the message has been processed.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the queue client.
    CloudQueueClient queueClient = storageAccount.createCloudQueueClient();

    // Retrieve a reference to a queue.
    CloudQueue queue = queueClient.getQueueReference("myqueue");

    // Retrieve the first visible message in the queue.
    CloudQueueMessage retrievedMessage = queue.retrieveMessage();

    if (retrievedMessage != null)
    {
        // Process the message in less than 30 seconds, and then delete the message.
        queue.deleteMessage(retrievedMessage);
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

Additional options for dequeuing messages

There are two ways you can customize message retrieval from a queue. First, you can get a batch of messages (up to 32). Second, you can set a longer or shorter invisibility timeout, allowing your code more or less time to fully process each message.

The following code example uses the **retrieveMessages** method to get 20 messages in one call. Then it processes each message using a **for** loop. It also sets the invisibility timeout to five minutes (300 seconds) for each message. Note that the five minutes starts for all messages at the same time, so when five minutes have passed since the call to **retrieveMessages**, any messages which have not been deleted will become visible again.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the queue client.
    CloudQueueClient queueClient = storageAccount.createCloudQueueClient();

    // Retrieve a reference to a queue.
    CloudQueue queue = queueClient.getQueueReference("myqueue");

    // Retrieve 20 messages from the queue with a visibility timeout of 300 seconds.
    for (CloudQueueMessage message : queue.retrieveMessages(20, 300, null, null)) {
        // Do processing for all messages in less than 5 minutes,
        // deleting each message after processing.
        queue.deleteMessage(message);
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

How to: List the queues

To obtain a list of the current queues, call the **CloudQueueClient.listQueues()** method, which will return a collection of **CloudQueue** objects.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the queue client.
    CloudQueueClient queueClient =
        storageAccount.createCloudQueueClient();

    // Loop through the collection of queues.
    for (CloudQueue queue : queueClient.listQueues())
    {
        // Output each queue name.
        System.out.println(queue.getName());
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Delete a queue

To delete a queue and all the messages contained in it, call the **deleteIfExists** method on the **CloudQueue** object.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the queue client.
    CloudQueueClient queueClient = storageAccount.createCloudQueueClient();

    // Retrieve a reference to a queue.
    CloudQueue queue = queueClient.getQueueReference("myqueue");

    // Delete the queue if it exists.
    queue.deleteIfExists();
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

Next steps

Now that you've learned the basics of queue storage, follow these links to learn about more complex storage tasks.

- [Azure Storage SDK for Java](#)
- [Azure Storage Client SDK Reference](#)
- [Azure Storage Services REST API](#)
- [Azure Storage Team Blog](#)

How to use Queue storage from Node.js

1/17/2017 • 10 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide shows you how to perform common scenarios using the Microsoft Azure Queue service. The samples are written using the Node.js API. The scenarios covered include **inserting**, **peeking**, **getting**, and **deleting** queue messages, as well as **creating and deleting queues**.

What is Queue Storage?

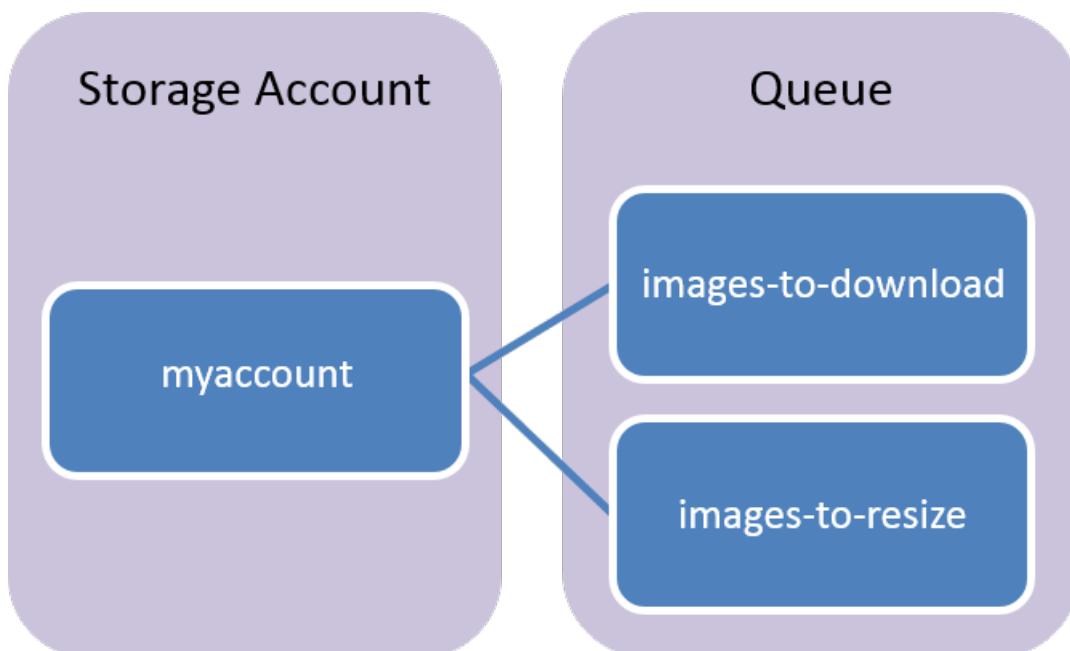
Azure Queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS. A single queue message can be up to 64 KB in size, and a queue can contain millions of messages, up to the total capacity limit of a storage account.

Common uses of Queue storage include:

- Creating a backlog of work to process asynchronously
- Passing messages from an Azure web role to an Azure worker role

Queue Service Concepts

The Queue service contains the following components:



- **URL format:** Queues are addressable using the following URL format:

`http://<storage_account>.queue.core.windows.net/<queue>`

The following URL addresses a queue in the diagram:

```
http://myaccount.queue.core.windows.net/images-to-download
```

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. The maximum time that a message can remain in the queue is 7 days.

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a Node.js Application

Create a blank Node.js application. For instructions creating a Node.js application, see [Create a Node.js web app in Azure App Service](#), [Build and deploy a Node.js application to an Azure Cloud Service](#) using Windows PowerShell, or [Build and deploy a Node.js web app to Azure using Web Matrix](#).

Configure Your Application to Access Storage

To use Azure storage, you need the Azure Storage SDK for Node.js, which includes a set of convenience libraries that communicate with the storage REST services.

Use Node Package Manager (NPM) to obtain the package

1. Use a command-line interface such as **PowerShell** (Windows,) **Terminal** (Mac,) or **Bash** (Unix), navigate to the folder where you created your sample application.
2. Type **npm install azure-storage** in the command window. Output from the command is similar to the following example.

```
azure-storage@0.5.0 node_modules\azure-storage
+-- extend@1.2.1
+-- xmlbuilder@0.4.3
+-- mime@1.2.11
+-- node-uuid@1.4.3
+-- validator@3.22.2
+-- underscore@1.4.4
+-- readable-stream@1.0.33 (string_decoder@0.10.31, isarray@0.0.1, inherits@2.0.1, core-util-is@1.0.1)
+-- xml2js@0.2.7 (sax@0.5.2)
+-- request@2.57.0 (caseless@0.10.0, aws-sign2@0.5.0, forever-agent@0.6.1, stringstream@0.0.4, oauth-sign@0.8.0, tunnel-agent@0.4.1, isstream@0.1.2, json-stringify-safe@5.0.1, bl@0.9.4, combined-stream@1.0.5, qs@3.1.0, mime-types@2.0.14, form-data@0.2.0, http-signature@0.11.0, tough-cookie@2.0.0, hawk@2.3.1, har-validator@1.8.0)
```

3. You can manually run the **ls** command to verify that a **node_modules** folder was created. Inside that folder you will find the **azure-storage** package, which contains the libraries you need to access storage.

Import the package

Using Notepad or another text editor, add the following to the top the **server.js** file of the application where you intend to use storage:

```
var azure = require('azure-storage');
```

Setup an Azure Storage Connection

The azure module will read the environment variables AZURE_STORAGE_ACCOUNT and AZURE_STORAGE_ACCESS_KEY, or AZURE_STORAGE_CONNECTION_STRING for information required to connect to your Azure storage account. If these environment variables are not set, you must specify the account information when calling **createQueueService**.

For an example of setting the environment variables in the [Azure Portal](#) for an Azure Website, see [Node.js web app using the Azure Table Service](#).

How To: Create a Queue

The following code creates a **QueueService** object, which enables you to work with queues.

```
var queueSvc = azure.createQueueService();
```

Use the **createQueueIfNotExists** method, which returns the specified queue if it already exists or creates a new queue with the specified name if it does not already exist.

```
queueSvc.createQueueIfNotExists('myqueue', function(error, result, response){
  if(!error){
    // Queue created or exists
  }
});
```

If the queue is created, `result.created` is true. If the queue exists, `result.created` is false.

Filters

Optional filtering operations can be applied to operations performed using **QueueService**. Filtering operations can include logging, automatically retrying, etc. Filters are objects that implement a method with the signature:

```
function handle (requestOptions, next)
```

After doing its preprocessing on the request options, the method needs to call "next" passing a callback with the following signature:

```
function (returnObject, finalCallback, next)
```

In this callback, and after processing the returnObject (the response from the request to the server), the callback needs to either invoke next if it exists to continue processing other filters or simply invoke finalCallback otherwise to end up the service invocation.

Two filters that implement retry logic are included with the Azure SDK for Node.js, **ExponentialRetryPolicyFilter** and **LinearRetryPolicyFilter**. The following creates a **QueueService** object that uses the **ExponentialRetryPolicyFilter**:

```
var retryOperations = new azure.ExponentialRetryPolicyFilter();
var queueSvc = azure.createQueueService().withFilter(retryOperations);
```

How To: Insert a Message into a Queue

To insert a message into a queue, use the **createMessage** method to create a new message and add it to the queue.

```
queueSvc.createMessage('myqueue', "Hello world!", function(error, result, response){
  if(!error){
    // Message inserted
  }
});
```

How To: Peek at the Next Message

You can peek at the message in the front of a queue without removing it from the queue by calling the **peekMessages** method. By default, **peekMessages** peeks at a single message.

```
queueSvc.peekMessages('myqueue', function(error, result, response){
  if(!error){
    // Message text is in messages[0].messageText
  }
});
```

The `result` contains the message.

NOTE

Using **peekMessages** when there are no messages in the queue will not return an error, however no messages will be returned.

How To: Dequeue the Next Message

Processing a message is a two-stage process:

1. Dequeue the message.
2. Delete the message.

To dequeue a message, use **getMessages**. This makes the messages invisible in the queue, so no other clients can process them. Once your application has processed a message, call **deleteMessage** to delete it from the queue. The following example gets a message, then deletes it:

```
queueSvc.getMessages('myqueue', function(error, result, response){
  if(!error){
    // Message text is in messages[0].messageText
    var message = result[0];
    queueSvc.deleteMessage('myqueue', message.messageId, message.popReceipt, function(error, response){
      if(!error){
        //message deleted
      }
    });
  }
});
```

NOTE

By default, a message is only hidden for 30 seconds, after which it is visible to other clients. You can specify a different value by using `options.visibilityTimeout` with **getMessages**.

NOTE

Using **getMessages** when there are no messages in the queue will not return an error, however no messages will be returned.

How To: Change the Contents of a Queued Message

You can change the contents of a message in-place in the queue using **updateMessage**. The following example updates the text of a message:

```
queueSvc.getMessages('myqueue', function(error, result, response){
  if(!error){
    // Got the message
    var message = result[0];
    queueSvc.updateMessage('myqueue', message.messageId, message.popReceipt, 10, {messageText: 'new text'},
  function(error, result, response){
    if(!error){
      // Message updated successfully
    }
  });
}
});
```

How To: Additional Options for Dequeuing Messages

There are two ways you can customize message retrieval from a queue:

- `options.numOfMessages` - Retrieve a batch of messages (up to 32.)
- `options.visibilityTimeout` - Set a longer or shorter invisibility timeout.

The following example uses the **getMessages** method to get 15 messages in one call. Then it processes each message using a for loop. It also sets the invisibility timeout to five minutes for all messages returned by this method.

```
queueSvc.getMessages('myqueue', {numOfMessages: 15, visibilityTimeout: 5 * 60}, function(error, result,
response){
  if(!error){
    // Messages retrieved
    for(var index in result){
      // text is available in result[index].messageText
      var message = result[index];
      queueSvc.deleteMessage(queueName, message.messageId, message.popReceipt, function(error, response){
        if(!error){
          // Message deleted
        }
      });
    }
  });
});
```

How To: Get the Queue Length

The **getQueueMetadata** returns metadata about the queue, including the approximate number of messages waiting in the queue.

```
queueSvc.getQueueMetadata('myqueue', function(error, result, response){
  if(!error){
    // Queue length is available in result.approximateMessageCount
  }
});
```

How To: List Queues

To retrieve a list of queues, use **listQueuesSegmented**. To retrieve a list filtered by a specific prefix, use **listQueuesSegmentedWithPrefix**.

```
queueSvc.listQueuesSegmented(null, function(error, result, response){
  if(!error){
    // result.entries contains the list of queues
  }
});
```

If all queues cannot be returned, `result.continuationToken` can be used as the first parameter of **listQueuesSegmented** or the second parameter of **listQueuesSegmentedWithPrefix** to retrieve more results.

How To: Delete a Queue

To delete a queue and all the messages contained in it, call the **deleteQueue** method on the queue object.

```
queueSvc.deleteQueue(queueName, function(error, response){
  if(!error){
    // Queue has been deleted
  }
});
```

To clear all messages from a queue without deleting it, use **clearMessages**.

How to: Work with Shared Access Signatures

Shared Access Signatures (SAS) are a secure way to provide granular access to queues without providing your storage account name or keys. SAS are often used to provide limited access to your queues, such as allowing a mobile app to submit messages.

A trusted application such as a cloud-based service generates a SAS using the **generateSharedAccessSignature** of the **QueueService**, and provides it to an untrusted or semi-trusted application. For example, a mobile app. The SAS is generated using a policy, which describes the start and end dates during which the SAS is valid, as well as the access level granted to the SAS holder.

The following example generates a new shared access policy that will allow the SAS holder to add messages to the queue, and expires 100 minutes after the time it is created.

```

var startDate = new Date();
var expiryDate = new Date(startDate);
expiryDate.setMinutes(startDate.getMinutes() + 100);
startDate.setMinutes(startDate.getMinutes() - 100);

var sharedAccessPolicy = {
    AccessPolicy: {
        Permissions: azure.QueueUtilities.SharedAccessPermissions.ADD,
        Start: startDate,
        Expiry: expiryDate
    }
};

var queueSAS = queueSvc.generateSharedAccessSignature('myqueue', sharedAccessPolicy);
var host = queueSvc.host;

```

Note that the host information must be provided also, as it is required when the SAS holder attempts to access the queue.

The client application then uses the SAS with **QueueServiceWithSAS** to perform operations against the queue. The following example connects to the queue and creates a message.

```

var sharedQueueService = azure.createQueueServiceWithSas(host, queueSAS);
sharedQueueService.createMessage('myqueue', 'Hello world from SAS!', function(error, result, response){
    if(!error){
        //message added
    }
});

```

Since the SAS was generated with add access, if an attempt were made to read, update or delete messages, an error would be returned.

Access control lists

You can also use an Access Control List (ACL) to set the access policy for a SAS. This is useful if you wish to allow multiple clients to access the queue, but provide different access policies for each client.

An ACL is implemented using an array of access policies, with an ID associated with each policy. The following example defines two policies; one for 'user1' and one for 'user2':

```

var sharedAccessPolicy = [
    user1: {
        Permissions: azure.QueueUtilities.SharedAccessPermissions.PROCESS,
        Start: startDate,
        Expiry: expiryDate
    },
    user2: {
        Permissions: azure.QueueUtilities.SharedAccessPermissions.ADD,
        Start: startDate,
        Expiry: expiryDate
    }
];

```

The following example gets the current ACL for **myqueue**, then adds the new policies using **setQueueAcl**. This approach allows:

```
var extend = require('extend');
queueSvc.getQueueAcl('myqueue', function(error, result, response) {
  if(!error){
    var newSignedIdentifiers = extend(true, result.signedIdentifiers, sharedAccessPolicy);
    queueSvc.setQueueAcl('myqueue', newSignedIdentifiers, function(error, result, response){
      if(!error){
        // ACL set
      }
    });
  }
});
```

Once the ACL has been set, you can then create a SAS based on the ID for a policy. The following example creates a new SAS for 'user2':

```
queueSAS = queueSvc.generateSharedAccessSignature('myqueue', { Id: 'user2' });
```

Next Steps

Now that you've learned the basics of queue storage, follow these links to learn about more complex storage tasks.

- Visit the [Azure Storage Team Blog](#).
- Visit the [Azure Storage SDK for Node](#) repository on GitHub.

How to use Queue Storage from C++

1/17/2017 • 10 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide will show you how to perform common scenarios using the Azure Queue storage service. The samples are written in C++ and use the [Azure Storage Client Library for C++](#). The scenarios covered include **inserting**, **peeking**, **getting**, and **deleting** queue messages, as well as **creating and deleting queues**.

NOTE

This guide targets the Azure Storage Client Library for C++ version 1.0.0 and above. The recommended version is Storage Client Library 2.2.0, which is available via [NuGet](#) or [GitHub](#).

What is Queue Storage?

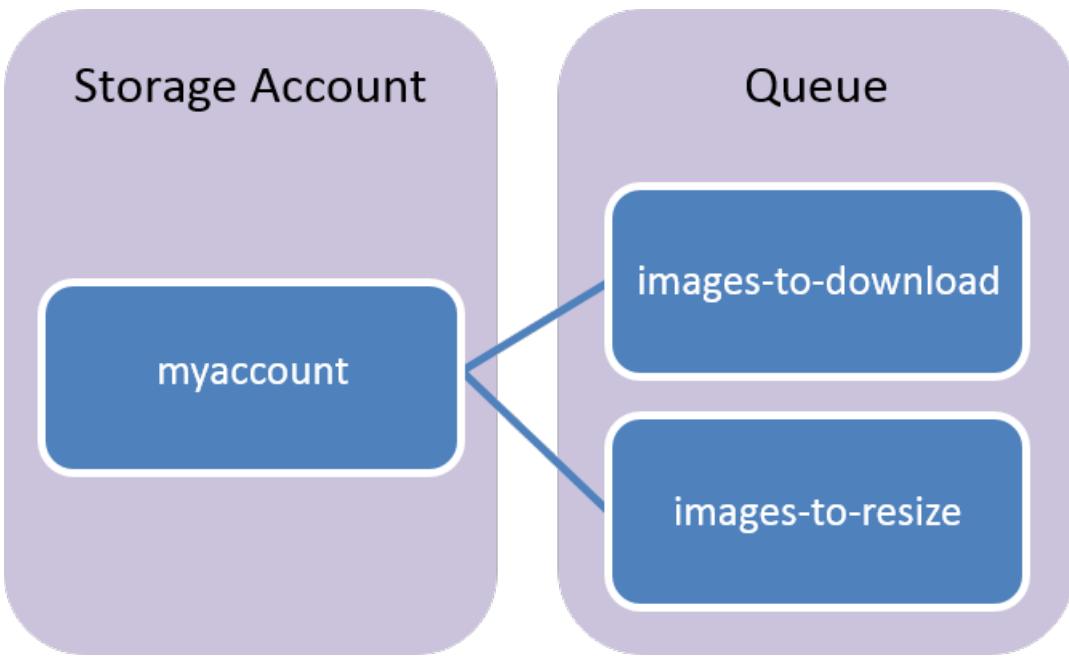
Azure Queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS. A single queue message can be up to 64 KB in size, and a queue can contain millions of messages, up to the total capacity limit of a storage account.

Common uses of Queue storage include:

- Creating a backlog of work to process asynchronously
- Passing messages from an Azure web role to an Azure worker role

Queue Service Concepts

The Queue service contains the following components:



- **URL format:** Queues are addressable using the following URL format:

`http://<storage account>.queue.core.windows.net/<queue>`

The following URL addresses a queue in the diagram:

`http://myaccount.queue.core.windows.net/images-to-download`

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. The maximum time that a message can remain in the queue is 7 days.

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a C++ application

In this guide, you will use storage features which can be run within a C++ application.

To do so, you will need to install the Azure Storage Client Library for C++ and create an Azure storage account in your Azure subscription.

To install the Azure Storage Client Library for C++, you can use the following methods:

- **Linux:** Follow the instructions given in the [Azure Storage Client Library for C++ README](#) page.
- **Windows:** In Visual Studio, click **Tools > NuGet Package Manager > Package Manager Console**. Type the following command into the [NuGet Package Manager console](#) and press **ENTER**.

```
Install-Package wastorage
```

Configure your application to access Queue Storage

Add the following include statements to the top of the C++ file where you want to use the Azure storage APIs to access queues:

```
#include "was/storage_account.h"
#include "was/queue.h"
```

Set up an Azure storage connection string

An Azure storage client uses a storage connection string to store endpoints and credentials for accessing data management services. When running in a client application, you must provide the storage connection string in the following format, using the name of your storage account and the storage access key for the storage account listed in the [Azure Portal](#) for the *AccountName* and *AccountKey* values. For information on storage accounts and access keys, see [About Azure Storage Accounts](#). This example shows how you can declare a static field to hold the connection string:

```
// Define the connection-string with your values.
const utility::string_t
storage_connection_string(U("DefaultEndpointsProtocol=https;AccountName=your_storage_account;AccountKey=your_storage_account_key"));
```

To test your application in your local Windows computer, you can use the Microsoft Azure [storage emulator](#) that is installed with the [Azure SDK](#). The storage emulator is a utility that simulates the Blob, Queue, and Table services available in Azure on your local development machine. The following example shows how you can declare a static field to hold the connection string to your local storage emulator:

```
// Define the connection-string with Azure Storage Emulator.
const utility::string_t storage_connection_string(U("UseDevelopmentStorage=true;"));
```

To start the Azure storage emulator, select the **Start** button or press the **Windows** key. Begin typing **Azure Storage Emulator**, and select **Microsoft Azure Storage Emulator** from the list of applications.

The following samples assume that you have used one of these two methods to get the storage connection string.

Retrieve your connection string

You can use the **cloud_storage_account** class to represent your Storage Account information. To retrieve your storage account information from the storage connection string, you can use the **parse** method.

```
// Retrieve storage account from connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);
```

How to: Create a queue

A **cloud_queue_client** object lets you get reference objects for queues. The following code creates a **cloud_queue_client** object.

```

// Retrieve storage account from connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create a queue client.
azure::storage::cloud_queue_client queue_client = storage_account.create_cloud_queue_client();

```

Use the **cloud_queue_client** object to get a reference to the queue you want to use. You can create the queue if it doesn't exist.

```

// Retrieve a reference to a queue.
azure::storage::cloud_queue queue = queue_client.get_queue_reference(U("my-sample-queue"));

// Create the queue if it doesn't already exist.
queue.create_if_not_exists();

```

How to: Insert a message into a queue

To insert a message into an existing queue, first create a new **cloud_queue_message**. Next, call the **add_message** method. A **cloud_queue_message** can be created from either a string or a **byte** array. Here is code which creates a queue (if it doesn't exist) and inserts the message 'Hello, World':

```

// Retrieve storage account from connection-string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the queue client.
azure::storage::cloud_queue_client queue_client = storage_account.create_cloud_queue_client();

// Retrieve a reference to a queue.
azure::storage::cloud_queue queue = queue_client.get_queue_reference(U("my-sample-queue"));

// Create the queue if it doesn't already exist.
queue.create_if_not_exists();

// Create a message and add it to the queue.
azure::storage::cloud_queue_message message1(U("Hello, World"));
queue.add_message(message1);

```

How to: Peek at the next message

You can peek at the message in the front of a queue without removing it from the queue by calling the **peek_message** method.

```

// Retrieve storage account from connection-string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the queue client.
azure::storage::cloud_queue_client queue_client = storage_account.create_cloud_queue_client();

// Retrieve a reference to a queue.
azure::storage::cloud_queue queue = queue_client.get_queue_reference(U("my-sample-queue"));

// Peek at the next message.
azure::storage::cloud_queue_message peeked_message = queue.peek_message();

// Output the message content.
std::wcout << U("Peeked message content: ") << peeked_message.content_as_string() << std::endl;

```

How to: Change the contents of a queued message

You can change the contents of a message in-place in the queue. If the message represents a work task, you could use this feature to update the status of the work task. The following code updates the queue message with new contents, and sets the visibility timeout to extend another 60 seconds. This saves the state of work associated with the message, and gives the client another minute to continue working on the message. You could use this technique to track multi-step workflows on queue messages, without having to start over from the beginning if a processing step fails due to hardware or software failure. Typically, you would keep a retry count as well, and if the message is retried more than n times, you would delete it. This protects against a message that triggers an application error each time it is processed.

```
// Retrieve storage account from connection-string.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the queue client.  
azure::storage::cloud_queue_client queue_client = storage_account.create_cloud_queue_client();  
  
// Retrieve a reference to a queue.  
azure::storage::cloud_queue queue = queue_client.get_queue_reference(U("my-sample-queue"));  
  
// Get the message from the queue and update the message contents.  
// The visibility timeout "0" means make it visible immediately.  
// The visibility timeout "60" means the client can get another minute to continue  
// working on the message.  
azure::storage::cloud_queue_message changed_message = queue.get_message();  
  
changed_message.set_content(U("Changed message"));  
queue.update_message(changed_message, std::chrono::seconds(60), true);  
  
// Output the message content.  
std::wcout << U("Changed message content: ") << changed_message.content_as_string() << std::endl;
```

How to: De-queue the next message

Your code de-queues a message from a queue in two steps. When you call **get_message**, you get the next message in a queue. A message returned from **get_message** becomes invisible to any other code reading messages from this queue. To finish removing the message from the queue, you must also call **delete_message**. This two-step process of removing a message assures that if your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls **delete_message** right after the message has been processed.

```
// Retrieve storage account from connection-string.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the queue client.  
azure::storage::cloud_queue_client queue_client = storage_account.create_cloud_queue_client();  
  
// Retrieve a reference to a queue.  
azure::storage::cloud_queue queue = queue_client.get_queue_reference(U("my-sample-queue"));  
  
// Get the next message.  
azure::storage::cloud_queue_message dequeued_message = queue.get_message();  
std::wcout << U("Dequeued message: ") << dequeued_message.content_as_string() << std::endl;  
  
// Delete the message.  
queue.delete_message(dequeued_message);
```

How to: Leverage additional options for de-queuing messages

There are two ways you can customize message retrieval from a queue. First, you can get a batch of messages (up to 32). Second, you can set a longer or shorter invisibility timeout, allowing your code more or less time to fully process each message. The following code example uses the **get_messages** method to get 20 messages in one call. Then it processes each message using a **for** loop. It also sets the invisibility timeout to five minutes for each message. Note that the 5 minutes starts for all messages at the same time, so after 5 minutes have passed since the call to **get_messages**, any messages which have not been deleted will become visible again.

```
// Retrieve storage account from connection-string.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the queue client.  
azure::storage::cloud_queue_client queue_client = storage_account.create_cloud_queue_client();  
  
// Retrieve a reference to a queue.  
azure::storage::cloud_queue queue = queue_client.get_queue_reference(U("my-sample-queue"));  
  
// Dequeue some queue messages (maximum 32 at a time) and set their visibility timeout to  
// 5 minutes (300 seconds).  
azure::storage::queue_request_options options;  
azure::storage::operation_context context;  
  
// Retrieve 20 messages from the queue with a visibility timeout of 300 seconds.  
std::vector<azure::storage::cloud_queue_message> messages = queue.get_messages(20, std::chrono::seconds(300),  
options, context);  
  
for (auto it = messages.cbegin(); it != messages.cend(); ++it)  
{  
    // Display the contents of the message.  
    std::wcout << U("Get: ") << it->content_as_string() << std::endl;  
}
```

How to: Get the queue length

You can get an estimate of the number of messages in a queue. The **download_attributes** method asks the Queue service to retrieve the queue attributes, including the message count. The **approximate_message_count** method gets the approximate number of messages in the queue.

```
// Retrieve storage account from connection-string.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the queue client.  
azure::storage::cloud_queue_client queue_client = storage_account.create_cloud_queue_client();  
  
// Retrieve a reference to a queue.  
azure::storage::cloud_queue queue = queue_client.get_queue_reference(U("my-sample-queue"));  
  
// Fetch the queue attributes.  
queue.download_attributes();  
  
// Retrieve the cached approximate message count.  
int cachedMessageCount = queue.approximate_message_count();  
  
// Display number of messages.  
std::wcout << U("Number of messages in queue: ") << cachedMessageCount << std::endl;
```

How to: Delete a queue

To delete a queue and all the messages contained in it, call the **delete_queue_if_exists** method on the queue object.

```
// Retrieve storage account from connection-string.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the queue client.  
azure::storage::cloud_queue_client queue_client = storage_account.create_cloud_queue_client();  
  
// Retrieve a reference to a queue.  
azure::storage::cloud_queue queue = queue_client.get_queue_reference(U("my-sample-queue"));  
  
// If the queue exists and delete it.  
queue.delete_queue_if_exists();
```

Next steps

Now that you've learned the basics of Queue storage, follow these links to learn more about Azure Storage.

- [How to use Blob Storage from C++](#)
- [How to use Table Storage from C++](#)
- [List Azure Storage Resources in C++](#)
- [Storage Client Library for C++ Reference](#)
- [Azure Storage Documentation](#)

How to use Queue storage from Python

1/17/2017 • 5 min to read • [Edit on GitHub](#)

TIP

[Try the Microsoft Azure Storage Explorer](#)

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide shows you how to perform common scenarios using the Azure Queue storage service. The samples are written in Python and use the [Microsoft Azure Storage SDK for Python](#). The scenarios covered include **inserting, peeking, getting, and deleting** queue messages, as well as **creating and deleting queues**. For more information on queues, refer to the [Next Steps] section.

What is Queue Storage?

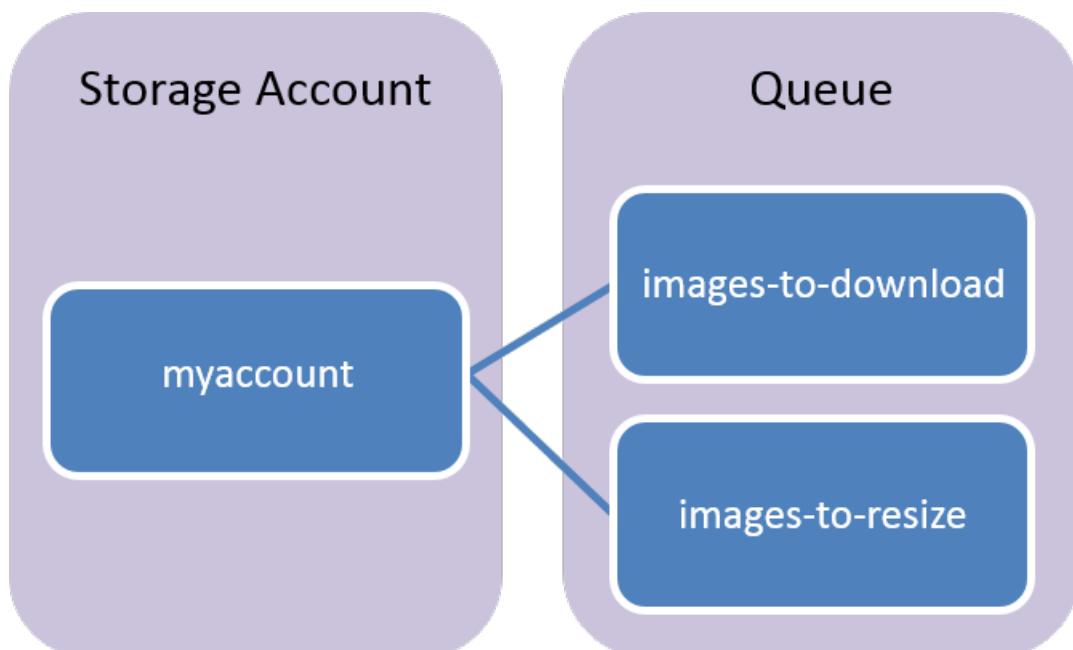
Azure Queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS. A single queue message can be up to 64 KB in size, and a queue can contain millions of messages, up to the total capacity limit of a storage account.

Common uses of Queue storage include:

- Creating a backlog of work to process asynchronously
- Passing messages from an Azure web role to an Azure worker role

Queue Service Concepts

The Queue service contains the following components:



- **URL format:** Queues are addressable using the following URL format:

```
http://<storage account>.queue.core.windows.net/<queue>
```

The following URL addresses a queue in the diagram:

```
http://myaccount.queue.core.windows.net/images-to-download
```

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. The maximum time that a message can remain in the queue is 7 days.

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

How To: Create a Queue

The **QueueService** object lets you work with queues. The following code creates a **QueueService** object. Add the following near the top of any Python file in which you wish to programmatically access Azure Storage:

```
from azure.storage.queue import QueueService
```

The following code creates a **QueueService** object using the storage account name and account key. Replace 'myaccount' and 'mykey' with your account name and key.

```
queue_service = QueueService(account_name='myaccount', account_key='mykey')

queue_service.create_queue('taskqueue')
```

How To: Insert a Message into a Queue

To insert a message into a queue, use the **put_message** method to create a new message and add it to the queue.

```
queue_service.put_message('taskqueue', u'Hello World')
```

How To: Peek at the Next Message

You can peek at the message in the front of a queue without removing it from the queue by calling the **peek_messages** method. By default, **peek_messages** peeks at a single message.

```
messages = queue_service.peek_messages('taskqueue')
for message in messages:
    print(message.content)
```

How To: Dequeue Messages

Your code removes a message from a queue in two steps. When you call **get_messages**, you get the next message in a queue by default. A message returned from **get_messages** becomes invisible to any other code reading messages from this queue. By default, this message stays invisible for 30 seconds. To finish removing the message from the queue, you must also call **delete_message**. This two-step process of removing a message assures that when your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls **delete_message** right after the message has been processed.

```
messages = queue_service.get_messages('taskqueue')
for message in messages:
    print(message.content)
    queue_service.delete_message('taskqueue', message.id, message.pop_receipt)
```

There are two ways you can customize message retrieval from a queue. First, you can get a batch of messages (up to 32). Second, you can set a longer or shorter invisibility timeout, allowing your code more or less time to fully process each message. The following code example uses the **get_messages** method to get 16 messages in one call. Then it processes each message using a for loop. It also sets the invisibility timeout to five minutes for each message.

```
messages = queue_service.get_messages('taskqueue', num_messages=16, visibility_timeout=5*60)
for message in messages:
    print(message.content)
    queue_service.delete_message('taskqueue', message.id, message.pop_receipt)
```

How To: Change the Contents of a Queued Message

You can change the contents of a message in-place in the queue. If the message represents a work task, you could use this feature to update the status of the work task. The code below uses the **update_message** method to update a message. The visibility timeout is set to 0, meaning the message appears immediately and the content is updated.

```
messages = queue_service.get_messages('taskqueue')
for message in messages:
    queue_service.update_message('taskqueue', message.id, message.pop_receipt, 0, u'Hello World Again')
```

How To: Get the Queue Length

You can get an estimate of the number of messages in a queue. The **get_queue_metadata** method asks the queue service to return metadata about the queue, and the **approximate_message_count**. The result is only approximate because messages can be added or removed after the queue service responds to your request.

```
metadata = queue_service.get_queue_metadata('taskqueue')
count = metadata.approximate_message_count
```

How To: Delete a Queue

To delete a queue and all the messages contained in it, call the **delete_queue** method.

```
queue_service.delete_queue('taskqueue')
```

Next Steps

Now that you've learned the basics of Queue storage, follow these links to learn more.

- [Python Developer Center](#)
- [Azure Storage Services REST API](#)
- [Azure Storage Team Blog](#)
- [Microsoft Azure Storage SDK for Python](#)

How to use Queue storage from PHP

1/17/2017 • 10 min to read • [Edit on GitHub](#)

TIP

[Try the Microsoft Azure Storage Explorer](#)

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide will show you how to perform common scenarios by using the Azure Queue storage service. The samples are written via classes from the Windows SDK for PHP. The covered scenarios include inserting, peeking, getting, and deleting queue messages, as well as creating and deleting queues.

What is Queue Storage?

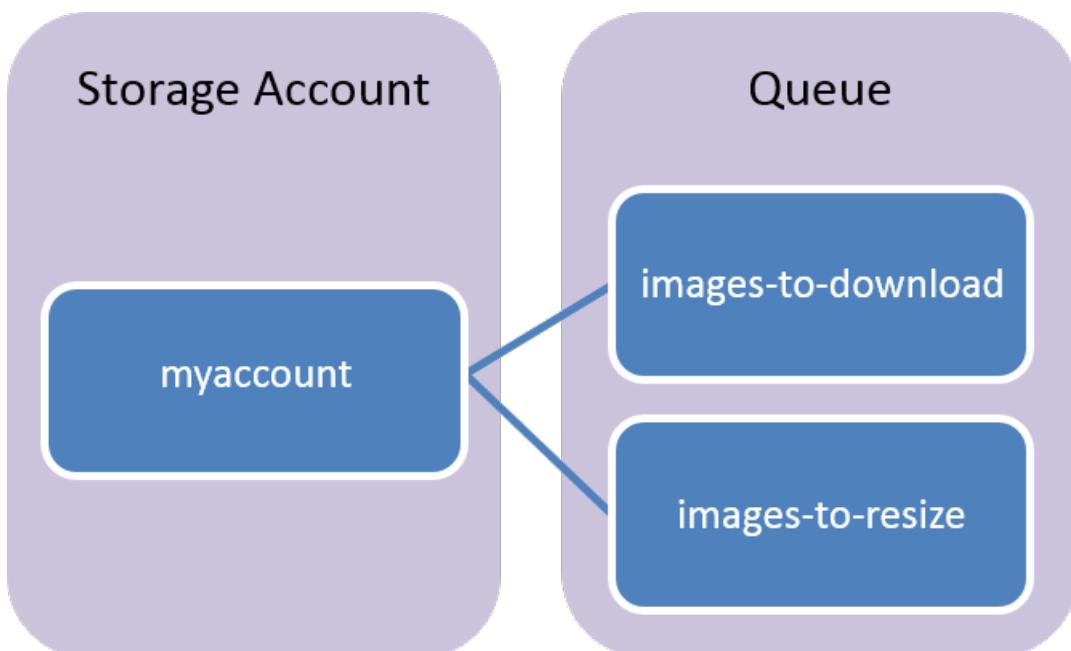
Azure Queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS. A single queue message can be up to 64 KB in size, and a queue can contain millions of messages, up to the total capacity limit of a storage account.

Common uses of Queue storage include:

- Creating a backlog of work to process asynchronously
- Passing messages from an Azure web role to an Azure worker role

Queue Service Concepts

The Queue service contains the following components:



- **URL format:** Queues are addressable using the following URL format:

`http://<storage account>.queue.core.windows.net/<queue>`

The following URL addresses a queue in the diagram:

```
http://myaccount.queue.core.windows.net/images-to-download
```

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. The maximum time that a message can remain in the queue is 7 days.

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a PHP application

The only requirement for creating a PHP application that accesses Azure Queue storage is the referencing of classes from the Azure SDK for PHP from within your code. You can use any development tools to create your application, including Notepad.

In this guide, you will use Queue storage features that can be called within a PHP application locally, or in code running within an Azure web role, worker role, or website.

Get the Azure Client Libraries

Install via Composer

1. [Install Git](#). Note that on Windows, you must also add the Git executable to your PATH environment variable.
2. Create a file named **composer.json** in the root of your project and add the following code to it:

```
{
  "require": {
    "microsoft/windowsazure": "^0.4"
  }
}
```

3. Download [composer.phar](#) in your project root.
4. Open a command prompt and execute the following command in your project root

```
php composer.phar install
```

Configure your application to access Queue storage

To use the APIs for Azure Queue storage, you need to:

1. Reference the autoloader file by using the `require_once` statement.

2. Reference any classes that you might use.

The following example shows how to include the autoloader file and reference the **ServicesBuilder** class.

NOTE

This example (and other examples in this article) assumes that you have installed the PHP Client Libraries for Azure via Composer. If you installed the libraries manually, you will need to reference the `WindowsAzure.php` autoloader file.

```
require_once 'vendor/autoload.php';
use WindowsAzure\Common\ServicesBuilder;
```

In the examples below, the `require_once` statement will be shown always, but only the classes that are necessary for the example to execute will be referenced.

Set up an Azure storage connection

To instantiate an Azure Queue storage client, you must first have a valid connection string. The format for the queue service connection string is as follows.

For accessing a live service:

```
DefaultEndpointsProtocol=[http|https];AccountName=[yourAccount];AccountKey=[yourKey]
```

For accessing the emulator storage:

```
UseDevelopmentStorage=true
```

To create any Azure service client, you need to use the **ServicesBuilder** class. You can use either of the following techniques:

- Pass the connection string directly to it.
- Use **CloudConfigurationManager (CCM)** to check multiple external sources for the connection string:
 - By default, it comes with support for one external source—environmental variables.
 - You can add new sources by extending the **ConnectionStringSource** class.

For the examples outlined here, the connection string will be passed directly.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;

$queueRestProxy = ServicesBuilder::getInstance()->createQueueService($connectionString);
```

Create a queue

A **QueueRestProxy** object lets you create a queue by using the **createQueue** method. When creating a queue, you can set options on the queue, but doing so is not required. (The example below shows how to set metadata on a queue.)

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;
use MicrosoftAzure\Storage\Queue\Models\CreateQueueOptions;

// Create queue REST proxy.
$queueRestProxy = ServicesBuilder::getInstance()->createQueueService($connectionString);

// OPTIONAL: Set queue metadata.
$createQueueOptions = new CreateQueueOptions();
$createQueueOptions->addMetaData("key1", "value1");
$createQueueOptions->addMetaData("key2", "value2");

try {
    // Create queue.
    $queueRestProxy->createQueue("myqueue", $createQueueOptions);
}

catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179446.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

NOTE

You should not rely on case sensitivity for metadata keys. All keys are read from the service in lowercase.

Add a message to a queue

To add a message to a queue, use **QueueRestProxy->createMessage**. The method takes the queue name, the message text, and message options (which are optional).

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;
use MicrosoftAzure\Storage\Queue\Models\CreateMessageOptions;

// Create queue REST proxy.
$queueRestProxy = ServicesBuilder::getInstance()->createQueueService($connectionString);

try {
    // Create message.
    $builder = new ServicesBuilder();
    $queueRestProxy->createMessage("myqueue", "Hello World!");
}

catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179446.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Peek at the next message

You can peek at a message (or messages) at the front of a queue without removing it from the queue by calling **QueueRestProxy->peekMessages**. By default, the **peekMessage** method returns a single message, but you can change that value by using the **PeekMessagesOptions->setNumberOfMessages** method.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;
use MicrosoftAzure\Storage\Queue\Models\.PeekMessagesOptions;

// Create queue REST proxy.
$queueRestProxy = ServicesBuilder::getInstance()->createQueueService($connectionString);

// OPTIONAL: Set peek message options.
$message_options = new PeekMessagesOptions();
$message_options->setNumberOfMessages(1); // Default value is 1.

try {
    $peekMessagesResult = $queueRestProxy->peekMessages("myqueue", $message_options);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179446.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

$messages = $peekMessagesResult->getQueueMessages();

// View messages.
$messageCount = count($messages);
if($messageCount <= 0){
    echo "There are no messages.<br />";
}
else{
    foreach($messages as $message) {
        echo "Peeked message:<br />";
        echo "Message Id: ".$message->getMessageId()."<br />";
        echo "Date: ".date_format($message->getInsertionDate(), 'Y-m-d')."<br />";
        echo "Message text: ".$message->getMessageText()."<br /><br />";
    }
}
```

De-queue the next message

Your code removes a message from a queue in two steps. First, you call **QueueRestProxy->listMessages**, which makes the message invisible to any other code that's reading from the queue. By default, this message will stay invisible for 30 seconds. (If the message is not deleted in this time period, it will become visible on the queue again.) To finish removing the message from the queue, you must call **QueueRestProxy->deleteMessage**. This two-step process of removing a message assures that when your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls **deleteMessage** right after the message has been processed.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create queue REST proxy.
$queueRestProxy = ServicesBuilder::getInstance()->createQueueService($connectionString);

// Get message.
$listMessagesResult = $queueRestProxy->listMessages("myqueue");
$messages = $listMessagesResult->getQueueMessages();
$message = $messages[0];

/* -----
   Process message.
----- */

// Get message ID and pop receipt.
$messageId = $message->getMessageId();
$popReceipt = $message->getPopReceipt();

try {
    // Delete message.
    $queueRestProxy->deleteMessage("myqueue", $messageId, $popReceipt);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179446.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Change the contents of a queued message

You can change the contents of a message in-place in the queue by calling **QueueRestProxy->updateMessage**. If the message represents a work task, you could use this feature to update the status of the work task. The following code updates the queue message with new contents, and it sets the visibility timeout to extend another 60 seconds. This saves the state of work that's associated with the message, and it gives the client another minute to continue working on the message. You could use this technique to track multi-step workflows on queue messages, without having to start over from the beginning if a processing step fails due to hardware or software failure. Typically, you would keep a retry count as well, and if the message is retried more than n times, you would delete it. This protects against a message that triggers an application error each time it is processed.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create queue REST proxy.
$queueRestProxy = ServicesBuilder::getInstance()->createQueueService($connectionString);

// Get message.
$listMessagesResult = $queueRestProxy->listMessages("myqueue");
$messages = $listMessagesResult->getQueueMessages();
$message = $messages[0];

// Define new message properties.
$new_message_text = "New message text.";
$new_visibility_timeout = 5; // Measured in seconds.

// Get message ID and pop receipt.
$messageId = $message->getMessageId();
$popReceipt = $message->getPopReceipt();

try {
    // Update message.
    $queueRestProxy->updateMessage("myqueue",
        $messageId,
        $popReceipt,
        $new_message_text,
        $new_visibility_timeout);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179446.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Additional options for de-queuing messages

There are two ways that you can customize message retrieval from a queue. First, you can get a batch of messages (up to 32). Second, you can set a longer or shorter visibility timeout, allowing your code more or less time to fully process each message. The following code example uses the **getMessages** method to get 16 messages in one call. Then it processes each message by using a **for** loop. It also sets the invisibility timeout to five minutes for each message.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;
use MicrosoftAzure\Storage\Queue\Models\ListMessagesOptions;

// Create queue REST proxy.
$queueRestProxy = ServicesBuilder::getInstance()->createQueueService($connectionString);

// Set list message options.
$message_options = new ListMessagesOptions();
$message_options->setVisibilityTimeoutInSeconds(300);
$message_options->setNumberOfMessages(16);

// Get messages.
try{
    $listMessagesResult = $queueRestProxy->listMessages("myqueue",
                                                       $message_options);
    $messages = $listMessagesResult->getQueueMessages();

    foreach($messages as $message){

        /* -----
           Process message.
        ----- */

        // Get message Id and pop receipt.
        $messageId = $message->getMessageId();
        $popReceipt = $message->getPopReceipt();

        // Delete message.
        $queueRestProxy->deleteMessage("myqueue", $messageId, $popReceipt);
    }
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179446.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Get queue length

You can get an estimate of the number of messages in a queue. The **QueueRestProxy->getQueueMetadata** method asks the queue service to return metadata about the queue. Calling the **getApproximateMessageCount** method on the returned object provides a count of how many messages are in a queue. The count is only approximate because messages can be added or removed after the queue service responds to your request.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create queue REST proxy.
$queueRestProxy = ServicesBuilder::getInstance()->createQueueService($connectionString);

try {
    // Get queue metadata.
    $queue_metadata = $queueRestProxy->getQueueMetadata("myqueue");
    $approx_msg_count = $queue_metadata->getApproximateMessageCount();
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179446.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

echo $approx_msg_count;

```

Delete a queue

To delete a queue and all the messages in it, call the **QueueRestProxy->deleteQueue** method.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create queue REST proxy.
$queueRestProxy = ServicesBuilder::getInstance()->createQueueService($connectionString);

try {
    // Delete queue.
    $queueRestProxy->deleteQueue("myqueue");
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179446.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Next steps

Now that you've learned the basics of Azure Queue storage, follow these links to learn about more complex storage tasks:

- Visit the [Azure Storage Team blog](#).

For more information, see also the [PHP Developer Center](#).

How to use Queue storage from Ruby

1/17/2017 • 6 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide shows you how to perform common scenarios using the Microsoft Azure Queue Storage service. The samples are written using the Ruby Azure API. The scenarios covered include **inserting**, **peeking**, **getting**, and **deleting** queue messages, as well as **creating and deleting queues**.

What is Queue Storage?

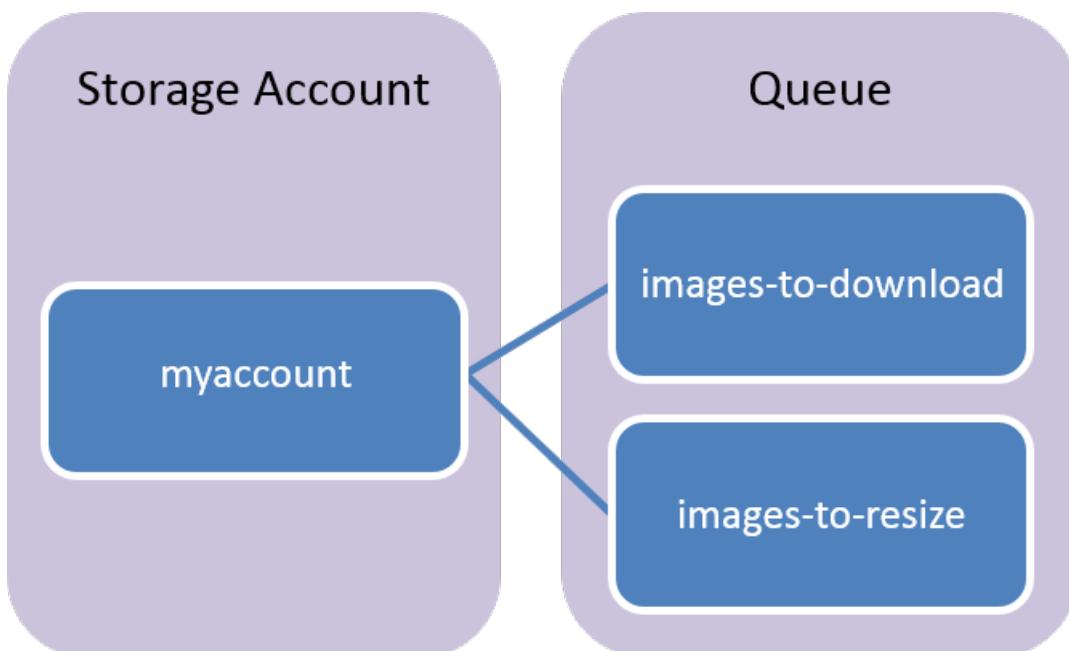
Azure Queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS. A single queue message can be up to 64 KB in size, and a queue can contain millions of messages, up to the total capacity limit of a storage account.

Common uses of Queue storage include:

- Creating a backlog of work to process asynchronously
- Passing messages from an Azure web role to an Azure worker role

Queue Service Concepts

The Queue service contains the following components:



- **URL format:** Queues are addressable using the following URL format:

`http://<storage account>.queue.core.windows.net/<queue>`

The following URL addresses a queue in the diagram:

```
http://myaccount.queue.core.windows.net/images-to-download
```

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Queue:** A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase. For information on naming queues, see [Naming Queues and Metadata](#).
- **Message:** A message, in any format, of up to 64 KB. The maximum time that a message can remain in the queue is 7 days.

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a Ruby Application

Create a Ruby application. For instructions, see [Ruby on Rails Web application on an Azure VM](#).

Configure Your Application to Access Storage

To use Azure storage, you need to download and use the Ruby azure package, which includes a set of convenience libraries that communicate with the storage REST services.

Use RubyGems to obtain the package

1. Use a command-line interface such as **PowerShell** (Windows), **Terminal** (Mac), or **Bash** (Unix).
2. Type "gem install azure" in the command window to install the gem and dependencies.

Import the package

Use your favorite text editor, add the following to the top of the Ruby file where you intend to use storage:

```
require "azure"
```

Setup an Azure Storage Connection

The azure module will read the environment variables **AZURE_STORAGE_ACCOUNT** and **AZURE_STORAGE_ACCESS_KEY** for information required to connect to your Azure storage account. If these environment variables are not set, you must specify the account information before using **Azure::QueueService** with the following code:

```
Azure.config.storage_account_name = "<your azure storage account>"  
Azure.config.storage_access_key = "<your Azure storage access key>"
```

To obtain these values from a classic or Resource Manager storage account in the Azure portal:

1. Log in to the [Azure portal](#).

2. Navigate to the storage account you want to use.
3. In the Settings blade on the right, click **Access Keys**.
4. In the Access keys blade that appears, you'll see the access key 1 and access key 2. You can use either of these.
5. Click the copy icon to copy the key to the clipboard.

To obtain these values from a classic storage account in the classic Azure portal:

1. Log in to the [classic Azure portal](#).
2. Navigate to the storage account you want to use.
3. Click **MANAGE ACCESS KEYS** at the bottom of the navigation pane.
4. In the pop up dialog, you'll see the storage account name, primary access key and secondary access key. For access key, you can use either the primary one or the secondary one.
5. Click the copy icon to copy the key to the clipboard.

How To: Create a Queue

The following code creates a **Azure::QueueService** object, which enables you to work with queues.

```
azure_queue_service = Azure::QueueService.new
```

Use the **create_queue()** method to create a queue with the specified name.

```
begin
  azure_queue_service.create_queue("test-queue")
rescue
  puts $!
end
```

How To: Insert a Message into a Queue

To insert a message into a queue, use the **create_message()** method to create a new message and add it to the queue.

```
azure_queue_service.create_message("test-queue", "test message")
```

How To: Peek at the Next Message

You can peek at the message in the front of a queue without removing it from the queue by calling the **peek_messages()** method. By default, **peek_messages()** peeks at a single message. You can also specify how many messages you want to peek.

```
result = azure_queue_service.peek_messages("test-queue",
  {:number_of_messages => 10})
```

How To: Dequeue the Next Message

You can remove a message from a queue in two steps.

1. When you call **list_messages()**, you get the next message in a queue by default. You can also specify how many messages you want to get. The messages returned from **list_messages()** becomes invisible to any other code reading messages from this queue. You pass in the visibility timeout in seconds as a parameter.
2. To finish removing the message from the queue, you must also call **delete_message()**.

This two-step process of removing a message assures that when your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls **delete_message()** right after the message has been processed.

```
messages = azure_queue_service.list_messages("test-queue", 30)
azure_queue_service.delete_message("test-queue",
    messages[0].id, messages[0].pop_receipt)
```

How To: Change the Contents of a Queued Message

You can change the contents of a message in-place in the queue. The code below uses the **update_message()** method to update a message. The method will return a tuple which contains the pop receipt of the queue message and a UTC date time value that represents when the message will be visible on the queue.

```
message = azure_queue_service.list_messages("test-queue", 30)
pop_receipt, time_next_visible = azure_queue_service.update_message(
    "test-queue", message.id, message.pop_receipt, "updated test message",
    30)
```

How To: Additional Options for Dequeuing Messages

There are two ways you can customize message retrieval from a queue.

1. You can get a batch of message.
2. You can set a longer or shorter invisibility timeout, allowing your code more or less time to fully process each message.

The following code example uses the **list_messages()** method to get 15 messages in one call. Then it prints and deletes each message. It also sets the invisibility timeout to five minutes for each message.

```
azure_queue_service.list_messages("test-queue", 300
  {:number_of_messages => 15}).each do |m|
  puts m.message_text
  azure_queue_service.delete_message("test-queue", m.id, m.pop_receipt)
end
```

How To: Get the Queue Length

You can get an estimation of the number of messages in the queue. The **get_queue_metadata()** method asks the queue service to return the approximate message count and metadata about the queue.

```
message_count, metadata = azure_queue_service.get_queue_metadata(
    "test-queue")
```

How To: Delete a Queue

To delete a queue and all the messages contained in it, call the **delete_queue()** method on the queue object.

```
azure_queue_service.delete_queue("test-queue")
```

Next Steps

Now that you've learned the basics of queue storage, follow these links to learn about more complex storage

tasks.

- Visit the [Azure Storage Team Blog](#)
- Visit the [Azure SDK for Ruby](#) repository on GitHub

For a comparison between the Azure Queue Service discussed in this article and Azure Service Bus Queues discussed in the [How to use Service Bus Queues](#) article, see [Azure Queues and Service Bus Queues - Compared and Contrasted](#)

Get started with Azure Table storage using .NET

1/17/2017 • 20 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

Azure Table storage is a service that stores structured NoSQL data in the cloud. Table storage is a key/attribute store with a schemaless design. Because Table storage is schemaless, it's easy to adapt your data as the needs of your application evolve. Access to data is fast and cost-effective for all kinds of applications. Table storage is typically significantly lower in cost than traditional SQL for similar volumes of data.

You can use Table storage to store flexible datasets, such as user data for web applications, address books, device information, and any other type of metadata that your service requires. You can store any number of entities in a table, and a storage account may contain any number of tables, up to the capacity limit of the storage account.

About this tutorial

This tutorial shows how to write .NET code for some common scenarios using Azure Table storage, including creating and deleting a table and inserting, updating, deleting, and querying table data.

Prerequisites:

- [Microsoft Visual Studio](#)
- [Azure Storage Client Library for .NET](#)
- [Azure Configuration Manager for .NET](#)
- An [Azure storage account](#)

NOTE

We recommend that you use the latest version of the Azure Storage Client Library for .NET to complete this tutorial. The latest version of the library is 7.x, available for download on [Nuget](#). The source for the client library is available on [GitHub](#).

If you are using the storage emulator, note that version 7.x of the client library requires at least version 4.3 of the storage emulator

More samples

For additional examples using Table storage, see [Getting Started with Azure Table Storage in .NET](#). You can download the sample application and run it, or browse the code on GitHub.

What is the Table Service

The Azure Table storage service stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of the Table service include:

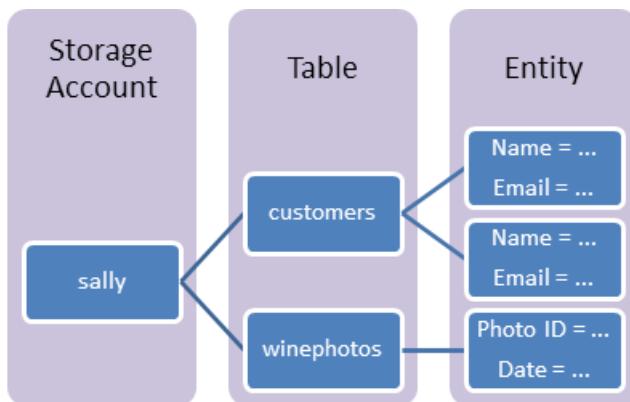
- Storing TBs of structured data capable of serving web scale applications

- Storing datasets that don't require complex joins, foreign keys, or stored procedures and can be denormalized for fast access
- Quickly querying data using a clustered index
- Accessing data using the OData protocol and LINQ queries with WCF Data Service .NET Libraries

You can use the Table service to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table Service Concepts

The Table service contains the following components:



- **URL format:** Code addresses tables in an account using this address format:

`http://<storage account>.table.core.windows.net/<table>`

You can address Azure tables directly using this address with the OData protocol. For more information, see [OData.org](#)

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Table:** A table is a collection of entities. Tables don't enforce a schema on entities, which means a single table can contain entities that have different sets of properties. The number of tables that a storage account can contain is limited only by the storage account capacity limit.
- **Entity:** An entity is a set of properties, similar to a database row. An entity can be up to 1MB in size.
- **Properties:** A property is a name-value pair. Each entity can include up to 252 properties to store data. Each entity also has 3 system properties that specify a partition key, a row key, and a timestamp. Entities with the same partition key can be queried more quickly, and inserted/updated in atomic operations. An entity's row key is its unique identifier within a partition.

For details about naming tables and properties, see [Understanding the Table Service Data Model](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

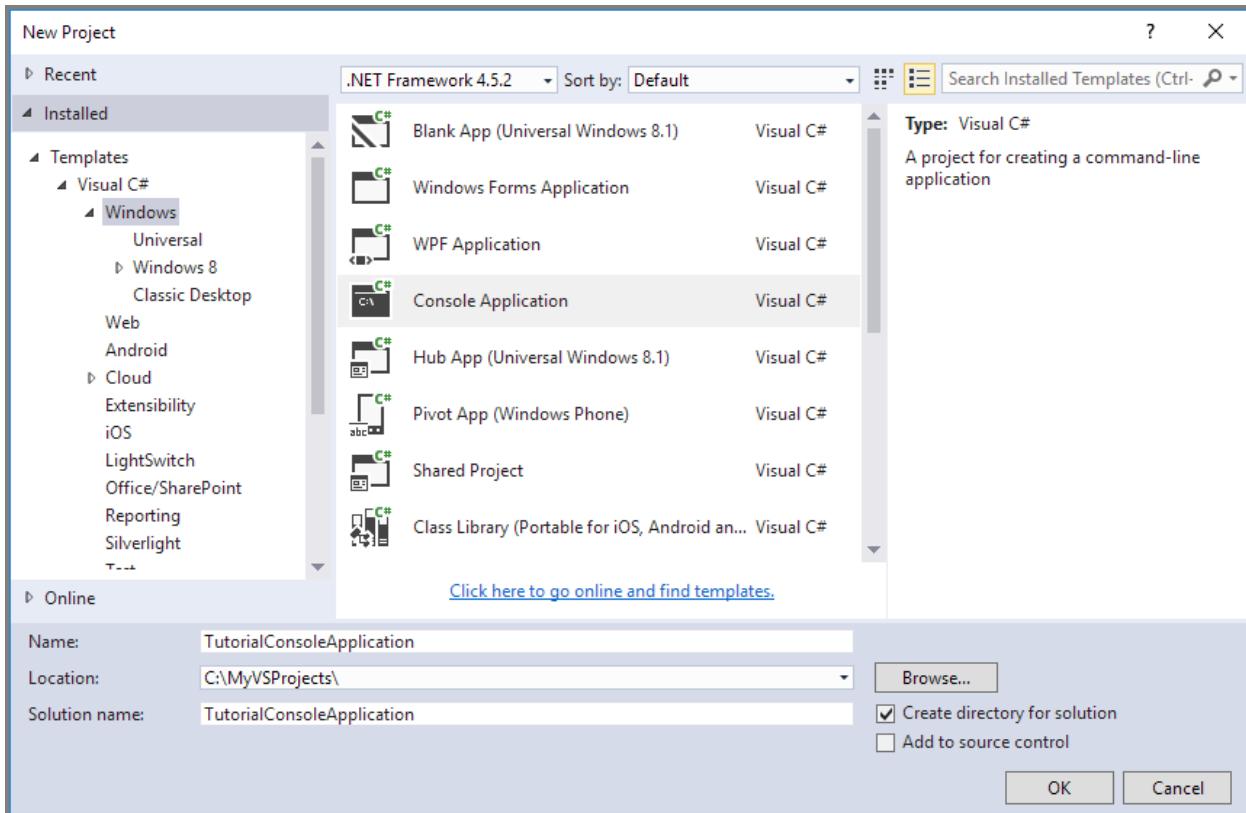
If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Set up your development environment

Next, set up your development environment in Visual Studio so that you are ready to try the code examples provided in this guide.

Create a Windows console application project

In Visual Studio, create a new Windows console application, as shown:



All of the code examples in this tutorial can be added to the `Main()` method in `Program.cs` in your console application.

Note that you can use the Azure Storage Client Library from any type of .NET application, including an Azure cloud service, an Azure web app, a desktop application, or a mobile application. In this guide, we use a console application for simplicity.

Use NuGet to install the required packages

There are two packages that you'll need to install to your project to complete this tutorial:

- [Microsoft Azure Storage Client Library for .NET](#): This package provides programmatic access to data resources in your storage account.
- [Microsoft Azure Configuration Manager library for .NET](#): This package provides a class for parsing a connection string from a configuration file, regardless of where your application is running.

You can use NuGet to obtain both packages. Follow these steps:

1. Right-click your project in **Solution Explorer** and choose **Manage NuGet Packages**.
2. Search online for "WindowsAzure.Storage" and click **Install** to install the Storage Client Library and its dependencies.
3. Search online for "ConfigurationManager" and click **Install** to install the Azure Configuration Manager.

NOTE

The Storage Client Library package is also included in the [Azure SDK for .NET](#). However, we recommend that you also install the Storage Client Library from NuGet to ensure that you always have the latest version of the client library.

The ODataLib dependencies in the Storage Client Library for .NET are resolved through the ODataLib (version 5.0.2 and greater) packages available through NuGet, and not through WCF Data Services. The ODataLib libraries can be downloaded directly or referenced by your code project through NuGet. The specific ODataLib packages used by the Storage Client Library are [OData](#), [Edm](#), and [Spatial](#). While these libraries are used by the Azure Table storage classes, they are required dependencies for programming with the Storage Client Library.

Determine your target environment

You have two environment options for running the examples in this guide:

- You can run your code against an Azure Storage account in the cloud.
- You can run your code against the Azure storage emulator. The storage emulator is a local environment that emulates an Azure Storage account in the cloud. The emulator is a free option for testing and debugging your code while your application is under development. The emulator uses a well-known account and key. For more details, see [Use the Azure Storage Emulator for Development and Testing](#)

If you are targeting a storage account in the cloud, copy the primary access key for your storage account from the Azure Portal. For more information, see [View and copy storage access keys](#).

NOTE

You can target the storage emulator to avoid incurring any costs associated with Azure Storage. However, if you do choose to target an Azure storage account in the cloud, costs for performing this tutorial will be negligible.

Configure your storage connection string

The Azure Storage Client Library for .NET supports using a storage connection string to configure endpoints and credentials for accessing storage services. The best way to maintain your storage connection string is in a configuration file.

For more information about connection strings, see [Configure a Connection String to Azure Storage](#).

NOTE

Your storage account key is similar to the root password for your storage account. Always be careful to protect your storage account key. Avoid distributing it to other users, hard-coding it, or saving it in a plain-text file that is accessible to others. Regenerate your key using the Azure Portal if you believe it may have been compromised.

To configure your connection string, open the `app.config` file from Solution Explorer in Visual Studio. Add the contents of the `<appSettings>` element shown below. Replace `account-name` with the name of your storage account, and `account-key` with your account access key:

```
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
  </startup>
  <appSettings>
    <add key="StorageConnectionString" value="DefaultEndpointsProtocol=https;AccountName=account-
name;AccountKey=account-key" />
  </appSettings>
</configuration>
```

For example, your configuration setting will be similar to:

```
<add key="StorageConnectionString"
value="DefaultEndpointsProtocol=https;AccountName=storagesample;AccountKey=nYV0gln6fT7mvY+rxu2iWAEyzPKITGkhM88
J8HUoyofvK7C6fHcZc2kRZp6cKgYRUM741HI84L50Iau1+9hPjB==" />
```

To target the storage emulator, you can use a shortcut that maps to the well-known account name and key. In that case, your connection string setting will be:

```
<add key="StorageConnectionString" value="UseDevelopmentStorage=true;" />
```

Add namespace declarations

Add the following **using** statements to the top of the `program.cs` file:

```
using Microsoft.Azure; // Namespace for CloudConfigurationManager
using Microsoft.WindowsAzure.Storage; // Namespace for CloudStorageAccount
using Microsoft.WindowsAzure.Storage.Table; // Namespace for Table storage types
```

Parse the connection string

The [Microsoft Azure Configuration Manager Library for .NET](#) provides a class for parsing a connection string from a configuration file. The [CloudConfigurationManager](#) class parses configuration settings regardless of whether the client application is running on the desktop, on a mobile device, in an Azure virtual machine, or in an Azure cloud service.

To reference the [CloudConfigurationManager](#) package, add the following `using` directive:

```
using Microsoft.Azure; //Namespace for CloudConfigurationManager
```

Here's an example that shows how to retrieve a connection string from a configuration file:

```
// Parse the connection string and return a reference to the storage account.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));
```

Using the Azure Configuration Manager is optional. You can also use an API like the .NET Framework's [ConfigurationManager](#) class.

Create the Table service client

The [CloudTableClient](#) class enables you to retrieve tables and entities stored in Table storage. Here's one way to create the service client:

```
// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
```

Now you are ready to write code that reads data from and writes data to Table storage.

Create a table

This example shows how to create a table if it does not already exist:

```

// Retrieve the storage account from the connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

// Retrieve a reference to the table.
CloudTable table = tableClient.GetTableReference("people");

// Create the table if it doesn't exist.
table.CreateIfNotExists();

```

Add an entity to a table

Entities map to C# objects by using a custom class derived from **TableEntity**. To add an entity to a table, create a class that defines the properties of your entity. The following code defines an entity class that uses the customer's first name as the row key and last name as the partition key. Together, an entity's partition and row key uniquely identify the entity in the table. Entities with the same partition key can be queried faster than those with different partition keys, but using diverse partition keys allows for greater scalability of parallel operations. For any property that should be stored in the Table service, the property must be a public property of a supported type that exposes both setting and retrieving values. Also, your entity type *must* expose a parameterless constructor.

```

public class CustomerEntity : TableEntity
{
    public CustomerEntity(string lastName, string firstName)
    {
        this.PartitionKey = lastName;
        this.RowKey = firstName;
    }

    public CustomerEntity() { }

    public string Email { get; set; }

    public string PhoneNumber { get; set; }
}

```

Table operations that involve entities are performed via the **CloudTable** object that you created earlier in the "Create a table" section. The operation to be performed is represented by a **TableOperation** object. The following code example shows the creation of the **CloudTable** object and then a **CustomerEntity** object. To prepare the operation, a **TableOperation** object is created to insert the customer entity into the table. Finally, the operation is executed by calling **CloudTable.Execute**.

```

// Retrieve the storage account from the connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

// Create the CloudTable object that represents the "people" table.
CloudTable table = tableClient.GetTableReference("people");

// Create a new customer entity.
CustomerEntity customer1 = new CustomerEntity("Harp", "Walter");
customer1.Email = "Walter@contoso.com";
customer1.PhoneNumber = "425-555-0101";

// Create the TableOperation object that inserts the customer entity.
TableOperation insertOperation = TableOperation.Insert(customer1);

// Execute the insert operation.
table.Execute(insertOperation);

```

Insert a batch of entities

You can insert a batch of entities into a table in one write operation. Some other notes on batch operations:

- You can perform updates, deletes, and inserts in the same single batch operation.
- A single batch operation can include up to 100 entities.
- All entities in a single batch operation must have the same partition key.
- While it is possible to perform a query as a batch operation, it must be the only operation in the batch.

The following code example creates two entity objects and adds each to **TableBatchOperation** by using the **Insert** method. Then, **CloudTable.Execute** is called to execute the operation.

```

// Retrieve the storage account from the connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

// Create the CloudTable object that represents the "people" table.
CloudTable table = tableClient.GetTableReference("people");

// Create the batch operation.
TableBatchOperation batchOperation = new TableBatchOperation();

// Create a customer entity and add it to the table.
CustomerEntity customer1 = new CustomerEntity("Smith", "Jeff");
customer1.Email = "Jeff@contoso.com";
customer1.PhoneNumber = "425-555-0104";

// Create another customer entity and add it to the table.
CustomerEntity customer2 = new CustomerEntity("Smith", "Ben");
customer2.Email = "Ben@contoso.com";
customer2.PhoneNumber = "425-555-0102";

// Add both customer entities to the batch insert operation.
batchOperation.Insert(customer1);
batchOperation.Insert(customer2);

// Execute the batch operation.
table.ExecuteBatch(batchOperation);

```

Retrieve all entities in a partition

To query a table for all entities in a partition, use a **TableQuery** object. The following code example specifies a filter for entities where 'Smith' is the partition key. This example prints the fields of each entity in the query results to the console.

```
// Retrieve the storage account from the connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

// Create the CloudTable object that represents the "people" table.
CloudTable table = tableClient.GetTableReference("people");

// Construct the query operation for all customer entities where PartitionKey="Smith".
TableQuery<CustomerEntity> query = new TableQuery<CustomerEntity>
().Where(TableQuery.GenerateFilterCondition("PartitionKey", QueryComparisons.Equal, "Smith"));

// Print the fields for each customer.
foreach (CustomerEntity entity in table.ExecuteQuery(query))
{
    Console.WriteLine("{0}, {1}\t{2}\t{3}", entity.PartitionKey, entity.RowKey,
        entity.Email, entity.PhoneNumber);
}
```

Retrieve a range of entities in a partition

If you don't want to query all the entities in a partition, you can specify a range by combining the partition key filter with a row key filter. The following code example uses two filters to get all entities in partition 'Smith' where the row key (first name) starts with a letter earlier than 'E' in the alphabet and then prints the query results.

```
// Retrieve the storage account from the connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

// Create the CloudTable object that represents the "people" table.
CloudTable table = tableClient.GetTableReference("people");

// Create the table query.
TableQuery<CustomerEntity> rangeQuery = new TableQuery<CustomerEntity>().Where(
    TableQuery.CombineFilters(
        TableQuery.GenerateFilterCondition("PartitionKey", QueryComparisons.Equal, "Smith"),
        TableOperators.And,
        TableQuery.GenerateFilterCondition("RowKey", QueryComparisons.LessThan, "E")));

// Loop through the results, displaying information about the entity.
foreach (CustomerEntity entity in table.ExecuteQuery(rangeQuery))
{
    Console.WriteLine("{0}, {1}\t{2}\t{3}", entity.PartitionKey, entity.RowKey,
        entity.Email, entity.PhoneNumber);
}
```

Retrieve a single entity

You can write a query to retrieve a single, specific entity. The following code uses **TableOperation** to specify the customer 'Ben Smith'. This method returns just one entity rather than a collection, and the returned value in

TableResult.Result is a **CustomerEntity** object. Specifying both partition and row keys in a query is the fastest way to retrieve a single entity from the Table service.

```
// Retrieve the storage account from the connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

// Create the CloudTable object that represents the "people" table.
CloudTable table = tableClient.GetTableReference("people");

// Create a retrieve operation that takes a customer entity.
TableOperation retrieveOperation = TableOperation.Retrieve<CustomerEntity>("Smith", "Ben");

// Execute the retrieve operation.
TableResult retrievedResult = table.Execute(retrieveOperation);

// Print the phone number of the result.
if (retrievedResult.Result != null)
    Console.WriteLine(((CustomerEntity)retrievedResult.Result).PhoneNumber);
else
    Console.WriteLine("The phone number could not be retrieved.");
```

Replace an entity

To update an entity, retrieve it from the Table service, modify the entity object, and then save the changes back to the Table service. The following code changes an existing customer's phone number. Instead of calling **Insert**, this code uses **Replace**. This causes the entity to be fully replaced on the server, unless the entity on the server has changed since it was retrieved, in which case the operation will fail. This failure is to prevent your application from inadvertently overwriting a change made between the retrieval and update by another component of your application. The proper handling of this failure is to retrieve the entity again, make your changes (if still valid), and then perform another **Replace** operation. The next section will show you how to override this behavior.

```

// Retrieve the storage account from the connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

// Create the CloudTable object that represents the "people" table.
CloudTable table = tableClient.GetTableReference("people");

// Create a retrieve operation that takes a customer entity.
TableOperation retrieveOperation = TableOperation.Retrieve<CustomerEntity>("Smith", "Ben");

// Execute the operation.
TableResult retrievedResult = table.Execute(retrieveOperation);

// Assign the result to a CustomerEntity object.
CustomerEntity updateEntity = (CustomerEntity)retrievedResult.Result;

if (updateEntity != null)
{
    // Change the phone number.
    updateEntity.PhoneNumber = "425-555-0105";

    // Create the Replace TableOperation.
    TableOperation updateOperation = TableOperation.Replace(updateEntity);

    // Execute the operation.
    table.Execute(updateOperation);

    Console.WriteLine("Entity updated.");
}
else
    Console.WriteLine("Entity could not be retrieved.");

```

Insert-or-replace an entity

Replace operations will fail if the entity has been changed since it was retrieved from the server. Furthermore, you must retrieve the entity from the server first in order for the **Replace** operation to be successful. Sometimes, however, you don't know if the entity exists on the server and the current values stored in it are irrelevant. Your update should overwrite them all. To accomplish this, you would use an **InsertOrReplace** operation. This operation inserts the entity if it doesn't exist, or replaces it if it does, regardless of when the last update was made. In the following code example, the customer entity for Ben Smith is still retrieved, but it is then saved back to the server via **InsertOrReplace**. Any updates made to the entity between the retrieval and update operations will be overwritten.

```

// Retrieve the storage account from the connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

// Create the CloudTable object that represents the "people" table.
CloudTable table = tableClient.GetTableReference("people");

// Create a retrieve operation that takes a customer entity.
TableOperation retrieveOperation = TableOperation.Retrieve<CustomerEntity>("Smith", "Ben");

// Execute the operation.
TableResult retrievedResult = table.Execute(retrieveOperation);

// Assign the result to a CustomerEntity object.
CustomerEntity updateEntity = (CustomerEntity)retrievedResult.Result;

if (updateEntity != null)
{
    // Change the phone number.
    updateEntity.PhoneNumber = "425-555-1234";

    // Create the InsertOrReplace TableOperation.
    TableOperation insertOrReplaceOperation = TableOperation.InsertOrReplace(updateEntity);

    // Execute the operation.
    table.Execute(insertOrReplaceOperation);

    Console.WriteLine("Entity was updated.");
}

else
    Console.WriteLine("Entity could not be retrieved.");

```

Query a subset of entity properties

A table query can retrieve just a few properties from an entity instead of all the entity properties. This technique, called projection, reduces bandwidth and can improve query performance, especially for large entities. The query in the following code returns only the email addresses of entities in the table. This is done by using a query of **DynamicTableEntity** and also **EntityResolver**. You can learn more about projection on the [Introducing Upsert and Query Projection blog post](#). Note that projection is not supported on the local storage emulator, so this code runs only when you're using an account on the Table service.

```

// Retrieve the storage account from the connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

// Create the CloudTable that represents the "people" table.
CloudTable table = tableClient.GetTableReference("people");

// Define the query, and select only the Email property.
TableQuery<DynamicTableEntity> projectionQuery = new TableQuery<DynamicTableEntity>().Select(new string[] {
    "Email"
});

// Define an entity resolver to work with the entity after retrieval.
EntityResolver<string> resolver = (pk, rk, ts, props, etag) => props.ContainsKey("Email") ?
    props["Email"].StringValue : null;

foreach (string projectedEmail in table.ExecuteQuery(projectionQuery, resolver, null, null))
{
    Console.WriteLine(projectedEmail);
}

```

Delete an entity

You can easily delete an entity after you have retrieved it, by using the same pattern shown for updating an entity. The following code retrieves and deletes a customer entity.

```

// Retrieve the storage account from the connection string.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create the table client.
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

// Create the CloudTable that represents the "people" table.
CloudTable table = tableClient.GetTableReference("people");

// Create a retrieve operation that expects a customer entity.
TableOperation retrieveOperation = TableOperation.Retrieve<CustomerEntity>("Smith", "Ben");

// Execute the operation.
TableResult retrievedResult = table.Execute(retrieveOperation);

// Assign the result to a CustomerEntity.
CustomerEntity deleteEntity = (CustomerEntity)retrievedResult.Result;

// Create the Delete TableOperation.
if (deleteEntity != null)
{
    TableOperation deleteOperation = TableOperation.Delete(deleteEntity);

    // Execute the operation.
    table.Execute(deleteOperation);

    Console.WriteLine("Entity deleted.");
}

else
    Console.WriteLine("Could not retrieve the entity.");

```

Delete a table

Finally, the following code example deletes a table from a storage account. A table that has been deleted will be unavailable to be re-created for a period of time following the deletion.

```
// Retrieve the storage account from the connection string.  
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(  
    CloudConfigurationManager.GetSetting("StorageConnectionString"));  
  
// Create the table client.  
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();  
  
// Create the CloudTable that represents the "people" table.  
CloudTable table = tableClient.GetTableReference("people");  
  
// Delete the table if it exists.  
table.DeleteIfExists();
```

Retrieve entities in pages asynchronously

If you are reading a large number of entities, and you want to process/display entities as they are retrieved rather than waiting for them all to return, you can retrieve entities by using a segmented query. This example shows how to return results in pages by using the Async-Await pattern so that execution is not blocked while you're waiting for a large set of results to return. For more details on using the Async-Await pattern in .NET, see [Asynchronous programming with Async and Await \(C# and Visual Basic\)](#).

```
// Initialize a default TableQuery to retrieve all the entities in the table.  
TableQuery<CustomerEntity> tableQuery = new TableQuery<CustomerEntity>();  
  
// Initialize the continuation token to null to start from the beginning of the table.  
TableContinuationToken continuationToken = null;  
  
do  
{  
    // Retrieve a segment (up to 1,000 entities).  
    TableQuerySegment<CustomerEntity> tableQueryResult =  
        await table.ExecuteQuerySegmentedAsync(tableQuery, continuationToken);  
  
    // Assign the new continuation token to tell the service where to  
    // continue on the next iteration (or null if it has reached the end).  
    continuationToken = tableQueryResult.ContinuationToken;  
  
    // Print the number of rows retrieved.  
    Console.WriteLine("Rows retrieved {0}", tableQueryResult.Results.Count);  
  
    // Loop until a null continuation token is received, indicating the end of the table.  
} while(continuationToken != null);
```

Next steps

Now that you've learned the basics of Table storage, follow these links to learn about more complex storage tasks:

- See more Table storage samples in [Getting Started with Azure Table Storage in .NET](#)
- View the Table service reference documentation for complete details about available APIs:
 - [Storage Client Library for .NET reference](#)
 - [REST API reference](#)
- Learn how to simplify the code you write to work with Azure Storage by using the [Azure WebJobs SDK](#)
- View more feature guides to learn about additional options for storing data in Azure.
 - [Get started with Azure Blob storage using .NET](#) to store unstructured data.

- o Connect to SQL Database by using .NET (C#) to store relational data.

How to use Table storage from Java

1/17/2017 • 18 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

Microsoft Azure Storage Explorer is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide will show you how to perform common scenarios using the Azure Table storage service. The samples are written in Java and use the [Azure Storage SDK for Java](#). The scenarios covered include **creating**, **listing**, and **deleting** tables, as well as **inserting**, **querying**, **modifying**, and **deleting** entities in a table. For more information on tables, see the [Next steps](#) section.

Note: An SDK is available for developers who are using Azure Storage on Android devices. For more information, see the [Azure Storage SDK for Android](#).

What is the Table Service

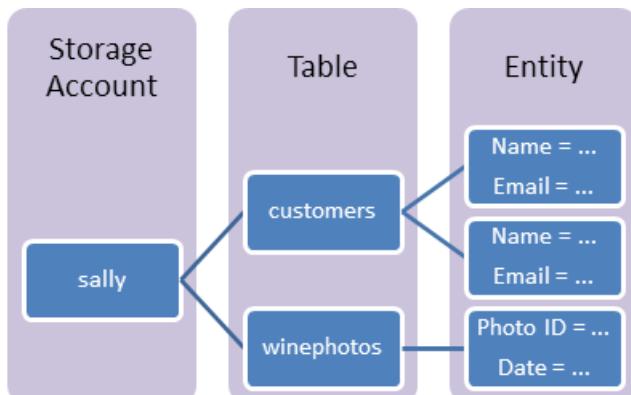
The Azure Table storage service stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of the Table service include:

- Storing TBs of structured data capable of serving web scale applications
- Storing datasets that don't require complex joins, foreign keys, or stored procedures and can be denormalized for fast access
- Quickly querying data using a clustered index
- Accessing data using the OData protocol and LINQ queries with WCF Data Service .NET Libraries

You can use the Table service to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table Service Concepts

The Table service contains the following components:



- **URL format:** Code addresses tables in an account using this address format:

```
http://<storage account>.table.core.windows.net/<table>
```

You can address Azure tables directly using this address with the OData protocol. For more information, see [OData.org](#)

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Table:** A table is a collection of entities. Tables don't enforce a schema on entities, which means a single table can contain entities that have different sets of properties. The number of tables that a storage account can contain is limited only by the storage account capacity limit.
- **Entity:** An entity is a set of properties, similar to a database row. An entity can be up to 1MB in size.
- **Properties:** A property is a name-value pair. Each entity can include up to 252 properties to store data. Each entity also has 3 system properties that specify a partition key, a row key, and a timestamp. Entities with the same partition key can be queried more quickly, and inserted/updated in atomic operations. An entity's row key is its unique identifier within a partition.

For details about naming tables and properties, see [Understanding the Table Service Data Model](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a Java application

In this guide, you will use storage features which can be run within a Java application locally, or in code running within a web role or worker role in Azure.

To do so, you will need to install the Java Development Kit (JDK) and create an Azure storage account in your Azure subscription. Once you have done so, you will need to verify that your development system meets the minimum requirements and dependencies which are listed in the [Azure Storage SDK for Java](#) repository on GitHub. If your system meets those requirements, you can follow the instructions for downloading and installing the Azure Storage Libraries for Java on your system from that repository. Once you have completed those tasks, you will be able to create a Java application which uses the examples in this article.

Configure your application to access table storage

Add the following import statements to the top of the Java file where you want to use Microsoft Azure storage APIs to access tables:

```
// Include the following imports to use table APIs
import com.microsoft.azure.storage.*;
import com.microsoft.azure.storage.table.*;
import com.microsoft.azure.storage.table.TableQuery.*;
```

Set up an Azure storage connection string

An Azure storage client uses a storage connection string to store endpoints and credentials for accessing data management services. When running in a client application, you must provide the storage connection string in the following format, using the name of your storage account and the Primary access key for the storage account listed in the [Azure portal](#) for the *AccountName* and *AccountKey* values. This example shows how you can declare a static field to hold the connection string:

```
// Define the connection-string with your values.  
public static final String storageConnectionString =  
    "DefaultEndpointsProtocol=http;" +  
    "AccountName=your_storage_account;" +  
    "AccountKey=your_storage_account_key";
```

In an application running within a role in Microsoft Azure, this string can be stored in the service configuration file, *ServiceConfiguration.cscfg*, and can be accessed with a call to the **RoleEnvironment.getConfigurationSettings** method. Here's an example of getting the connection string from a **Setting** element named *StorageConnectionString* in the service configuration file:

```
// Retrieve storage account from connection-string.  
String storageConnectionString =  
    RoleEnvironment.getConfigurationSettings().get("StorageConnectionString");
```

The following samples assume that you have used one of these two methods to get the storage connection string.

How to: Create a table

A **CloudTableClient** object lets you get reference objects for tables and entities. The following code creates a **CloudTableClient** object and uses it to create a new **CloudTable** object which represents a table named "people". (Note: There are additional ways to create **CloudStorageAccount** objects; for more information, see **CloudStorageAccount** in the [Azure Storage Client SDK Reference](#).)

```
try  
{  
    // Retrieve storage account from connection-string.  
    CloudStorageAccount storageAccount =  
        CloudStorageAccount.parse(storageConnectionString);  
  
    // Create the table client.  
    CloudTableClient tableClient = storageAccount.createCloudTableClient();  
  
    // Create the table if it doesn't exist.  
    String tableName = "people";  
    CloudTable cloudTable = tableClient.getTableReference(tableName);  
    cloudTable.createIfNotExists();  
}  
catch (Exception e)  
{  
    // Output the stack trace.  
    e.printStackTrace();  
}
```

How to: List the tables

To get a list of tables, call the **CloudTableClient.listTables()** method to retrieve an iterable list of table names.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Loop through the collection of table names.
    for (String table : tableClient.listTables())
    {
        // Output each table name.
        System.out.println(table);
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Add an entity to a table

Entities map to Java objects using a custom class implementing **TableEntity**. For convenience, the **TableServiceEntity** class implements **TableEntity** and uses reflection to map properties to getter and setter methods named for the properties. To add an entity to a table, first create a class that defines the properties of your entity. The following code defines an entity class which uses the customer's first name as the row key, and last name as the partition key. Together, an entity's partition and row key uniquely identify the entity in the table. Entities with the same partition key can be queried faster than those with different partition keys.

```

public class CustomerEntity extends TableServiceEntity {
    public CustomerEntity(String lastName, String firstName) {
        this.partitionKey = lastName;
        this.rowKey = firstName;
    }

    public CustomerEntity() { }

    String email;
    String phoneNumber;

    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPhoneNumber() {
        return this.phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }
}

```

Table operations involving entities require a **TableOperation** object. This object defines the operation to be performed on an entity, which can be executed with a **CloudTable** object. The following code creates a new instance of the **CustomerEntity** class with some customer data to be stored. The code next calls

TableOperation.insertOrReplace to create a **TableOperation** object to insert an entity into a table, and associates the new **CustomerEntity** with it. Finally, the code calls the **execute** method on the **CloudTable** object, specifying the "people" table and the new **TableOperation**, which then sends a request to the storage service to insert the new customer entity into the "people" table, or replace the entity if it already exists.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create a new customer entity.
    CustomerEntity customer1 = new CustomerEntity("Harp", "Walter");
    customer1.setEmail("Walter@contoso.com");
    customer1.setPhoneNumber("425-555-0101");

    // Create an operation to add the new customer to the people table.
    TableOperation insertCustomer1 = TableOperation.insertOrReplace(customer1);

    // Submit the operation to the table service.
    cloudTable.execute(insertCustomer1);
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

How to: Insert a batch of entities

You can insert a batch of entities to the table service in one write operation. The following code creates a **TableBatchOperation** object, then adds three insert operations to it. Each insert operation is added by creating a new entity object, setting its values, and then calling the **insert** method on the **TableBatchOperation** object to associate the entity with a new insert operation. Then the code calls **execute** on the **CloudTable** object, specifying the "people" table and the **TableBatchOperation** object, which sends the batch of table operations to the storage service in a single request.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Define a batch operation.
    TableBatchOperation batchOperation = new TableBatchOperation();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create a customer entity to add to the table.
    CustomerEntity customer = new CustomerEntity("Smith", "Jeff");
    customer.setEmail("Jeff@contoso.com");
    customer.setPhoneNumber("425-555-0104");
    batchOperation.insertOrReplace(customer);

    // Create another customer entity to add to the table.
    CustomerEntity customer2 = new CustomerEntity("Smith", "Ben");
    customer2.setEmail("Ben@contoso.com");
    customer2.setPhoneNumber("425-555-0102");
    batchOperation.insertOrReplace(customer2);

    // Create a third customer entity to add to the table.
    CustomerEntity customer3 = new CustomerEntity("Smith", "Denise");
    customer3.setEmail("Denise@contoso.com");
    customer3.setPhoneNumber("425-555-0103");
    batchOperation.insertOrReplace(customer3);

    // Execute the batch of operations on the "people" table.
    cloudTable.execute(batchOperation);
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

Some things to note on batch operations:

- You can perform up to 100 insert, delete, merge, replace, insert or merge, and insert or replace operations in any combination in a single batch.
- A batch operation can have a retrieve operation, if it is the only operation in the batch.
- All entities in a single batch operation must have the same partition key.
- A batch operation is limited to a 4MB data payload.

How to: Retrieve all entities in a partition

To query a table for entities in a partition, you can use a **TableQuery**. Call **TableQuery.from** to create a query on a particular table that returns a specified result type. The following code specifies a filter for entities where 'Smith' is the partition key. **TableQuery.generateFilterCondition** is a helper method to create filters for queries. Call **where** on the reference returned by the **TableQuery.from** method to apply the filter to the query. When the query is executed with a call to **execute** on the **CloudTable** object, it returns an **Iterator** with the **CustomerEntity** result type specified. You can then use the **Iterator** returned in a for each loop to consume the results. This code prints the fields of each entity in the query results to the console.

```

try
{
    // Define constants for filters.
    final String PARTITION_KEY = "PartitionKey";
    final String ROW_KEY = "RowKey";
    final String TIMESTAMP = "Timestamp";

    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create a filter condition where the partition key is "Smith".
    String partitionFilter = TableQuery.generateFilterCondition(
        PARTITION_KEY,
        QueryComparisons.EQUAL,
        "Smith");

    // Specify a partition query, using "Smith" as the partition key filter.
    TableQuery<CustomerEntity> partitionQuery =
        TableQuery.from(CustomerEntity.class)
        .where(partitionFilter);

    // Loop through the results, displaying information about the entity.
    for (CustomerEntity entity : cloudTable.execute(partitionQuery)) {
        System.out.println(entity.getPartitionKey() +
            " " + entity.getRowKey() +
            "\t" + entity.getEmail() +
            "\t" + entity.getPhoneNumber());
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Retrieve a range of entities in a partition

If you don't want to query all the entities in a partition, you can specify a range by using comparison operators in a filter. The following code combines two filters to get all entities in partition "Smith" where the row key (first name) starts with a letter up to 'E' in the alphabet. Then it prints the query results. If you use the entities added to the table in the batch insert section of this guide, only two entities are returned this time (Ben and Denise Smith); Jeff Smith is not included.

```

try
{
    // Define constants for filters.
    final String PARTITION_KEY = "PartitionKey";
    final String ROW_KEY = "RowKey";
    final String TIMESTAMP = "Timestamp";

    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create a filter condition where the partition key is "Smith".
    String partitionFilter = TableQuery.generateFilterCondition(
        PARTITION_KEY,
        QueryComparisons.EQUAL,
        "Smith");

    // Create a filter condition where the row key is less than the letter "E".
    String rowFilter = TableQuery.generateFilterCondition(
        ROW_KEY,
        QueryComparisons.LESS_THAN,
        "E");

    // Combine the two conditions into a filter expression.
    String combinedFilter = TableQuery.combineFilters(partitionFilter,
        Operators.AND, rowFilter);

    // Specify a range query, using "Smith" as the partition key,
    // with the row key being up to the letter "E".
    TableQuery<CustomerEntity> rangeQuery =
        TableQuery.from(CustomerEntity.class)
        .where(combinedFilter);

    // Loop through the results, displaying information about the entity
    for (CustomerEntity entity : cloudTable.execute(rangeQuery)) {
        System.out.println(entity.getPartitionKey() +
            " " + entity.getRowKey() +
            "\t" + entity.getEmail() +
            "\t" + entity.getPhoneNumber());
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Retrieve a single entity

You can write a query to retrieve a single, specific entity. The following code calls **TableOperation.retrieve** with partition key and row key parameters to specify the customer "Jeff Smith", instead of creating a **TableQuery** and using filters to do the same thing. When executed, the retrieve operation returns just one entity, rather than a collection. The **getResultSetAsType** method casts the result to the type of the assignment target, a **CustomerEntity** object. If this type is not compatible with the type specified for the query, an exception will be thrown. A null value is returned if no entity has an exact partition and row key match. Specifying both partition and row keys in a query is the fastest way to retrieve a single entity from the Table service.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Retrieve the entity with partition key of "Smith" and row key of "Jeff"
    TableOperation retrieveSmithJeff =
        TableOperation.retrieve("Smith", "Jeff", CustomerEntity.class);

    // Submit the operation to the table service and get the specific entity.
    CustomerEntity specificEntity =
        cloudTable.execute(retrieveSmithJeff).getResultAsType();

    // Output the entity.
    if (specificEntity != null)
    {
        System.out.println(specificEntity.getPartitionKey() +
            " " + specificEntity.getRowKey() +
            "\t" + specificEntity.getEmail() +
            "\t" + specificEntity.getPhoneNumber());
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Modify an entity

To modify an entity, retrieve it from the table service, make changes to the entity object, and save the changes back to the table service with a replace or merge operation. The following code changes an existing customer's phone number. Instead of calling **TableOperation.insert** like we did to insert, this code calls

TableOperation.replace. The **CloudTable.execute** method calls the table service, and the entity is replaced, unless another application changed it in the time since this application retrieved it. When that happens, an exception is thrown, and the entity must be retrieved, modified, and saved again. This optimistic concurrency retry pattern is common in a distributed storage system.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Retrieve the entity with partition key of "Smith" and row key of "Jeff".
    TableOperation retrieveSmithJeff =
        TableOperation.retrieve("Smith", "Jeff", CustomerEntity.class);

    // Submit the operation to the table service and get the specific entity.
    CustomerEntity specificEntity =
        cloudTable.execute(retrieveSmithJeff).getResultAsType();

    // Specify a new phone number.
    specificEntity.setPhoneNumber("425-555-0105");

    // Create an operation to replace the entity.
    TableOperation replaceEntity = TableOperation.replace(specificEntity);

    // Submit the operation to the table service.
    cloudTable.execute(replaceEntity);
}

catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Query a subset of entity properties

A query to a table can retrieve just a few properties from an entity. This technique, called projection, reduces bandwidth and can improve query performance, especially for large entities. The query in the following code uses the **select** method to return only the email addresses of entities in the table. The results are projected into a collection of **String** with the help of an **EntityResolver**, which does the type conversion on the entities returned from the server. You can learn more about projection in [Azure Tables: Introducing Upsert and Query Projection](#). Note that projection is not supported on the local storage emulator, so this code runs only when using an account on the table service.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Define a projection query that retrieves only the Email property
    TableQuery<CustomerEntity> projectionQuery =
        TableQuery.from(CustomerEntity.class)
            .select(new String[] {"Email"});

    // Define a Entity resolver to project the entity to the Email value.
    EntityResolver<String> emailResolver = new EntityResolver<String>() {
        @Override
        public String resolve(String PartitionKey, String RowKey, Date timeStamp, HashMap<String,
EntityProperty> properties, String etag) {
            return properties.get("Email").getValueAsString();
        }
    };

    // Loop through the results, displaying the Email values.
    for (String projectedString :
        cloudTable.execute(projectionQuery, emailResolver)) {
        System.out.println(projectedString);
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Insert or Replace an entity

Often you want to add an entity to a table without knowing if it already exists in the table. An insert-or-replace operation allows you to make a single request which will insert the entity if it does not exist or replace the existing one if it does. Building on prior examples, the following code inserts or replaces the entity for "Walter Harp". After creating a new entity, this code calls the **TableOperation.insertOrReplace** method. This code then calls **execute** on the **CloudTable** object with the table and the insert or replace table operation as the parameters. To update only part of an entity, the **TableOperation.insertOrMerge** method can be used instead. Note that insert-or-replace is not supported on the local storage emulator, so this code runs only when using an account on the table service. You can learn more about insert-or-replace and insert-or-merge in this [Azure Tables: Introducing Upsert and Query Projection](#).

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create a new customer entity.
    CustomerEntity customer5 = new CustomerEntity("Harp", "Walter");
    customer5.setEmail("Walter@contoso.com");
    customer5.setPhoneNumber("425-555-0106");

    // Create an operation to add the new customer to the people table.
    TableOperation insertCustomer5 = TableOperation.insertOrReplace(customer5);

    // Submit the operation to the table service.
    cloudTable.execute(insertCustomer5);
}

catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

How to: Delete an entity

You can easily delete an entity after you have retrieved it. Once the entity is retrieved, call **TableOperation.delete** with the entity to delete. Then call **execute** on the **CloudTable** object. The following code retrieves and deletes a customer entity.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create an operation to retrieve the entity with partition key of "Smith" and row key of "Jeff".
    TableOperation retrieveSmithJeff = TableOperation.retrieve("Smith", "Jeff", CustomerEntity.class);

    // Retrieve the entity with partition key of "Smith" and row key of "Jeff".
    CustomerEntity entitySmithJeff =
        cloudTable.execute(retrieveSmithJeff).getResultAsType();

    // Create an operation to delete the entity.
    TableOperation deleteSmithJeff = TableOperation.delete(entitySmithJeff);

    // Submit the delete operation to the table service.
    cloudTable.execute(deleteSmithJeff);
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

How to: Delete a table

Finally, the following code deletes a table from a storage account. A table which has been deleted will be unavailable to be recreated for a period of time following the deletion, usually less than forty seconds.

```

try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Delete the table and all its data if it exists.
    CloudTable cloudTable = tableClient.getTableReference("people");
    cloudTable.deleteIfExists();
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}

```

Next steps

Now that you've learned the basics of table storage, follow these links to learn how to do more complex storage tasks.

- [Azure Storage SDK for Java](#)
- [Azure Storage Client SDK Reference](#)

- [Azure Storage REST API](#)
- [Azure Storage Team Blog](#)

For more information, see also the [Java Developer Center](#).

How to use Azure Table storage from Node.js

1/17/2017 • 14 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This topic shows how to perform common scenarios using the Azure Table service in a Node.js application.

The code examples in this topic assume you already have a Node.js application. For information about how to create a Node.js application in Azure, see any of these topics:

- [Create a Node.js web app in Azure App Service](#)
- [Build and deploy a Node.js web app to Azure using WebMatrix](#)
- [Build and deploy a Node.js application to an Azure Cloud Service \(using Windows PowerShell\)](#)

What is the Table Service

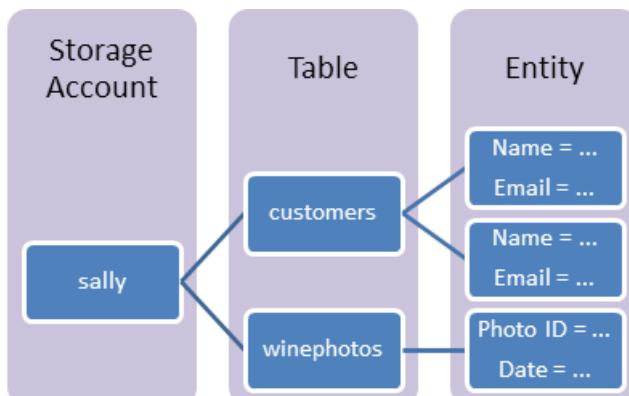
The Azure Table storage service stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of the Table service include:

- Storing TBs of structured data capable of serving web scale applications
- Storing datasets that don't require complex joins, foreign keys, or stored procedures and can be denormalized for fast access
- Quickly querying data using a clustered index
- Accessing data using the OData protocol and LINQ queries with WCF Data Service .NET Libraries

You can use the Table service to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table Service Concepts

The Table service contains the following components:



- **URL format:** Code addresses tables in an account using this address format:
`http://<storage account>.table.core.windows.net/<table>`
- You can address Azure tables directly using this address with the OData protocol. For more information, see [OData.org](#)
- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
 - **Table:** A table is a collection of entities. Tables don't enforce a schema on entities, which means a single table can contain entities that have different sets of properties. The number of tables that a storage account can contain is limited only by the storage account capacity limit.
 - **Entity:** An entity is a set of properties, similar to a database row. An entity can be up to 1MB in size.
 - **Properties:** A property is a name-value pair. Each entity can include up to 252 properties to store data. Each entity also has 3 system properties that specify a partition key, a row key, and a timestamp. Entities with the same partition key can be queried more quickly, and inserted/updated in atomic operations. An entity's row key is its unique identifier within a partition.

For details about naming tables and properties, see [Understanding the Table Service Data Model](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Configure your application to access Azure Storage

To use Azure Storage, you need the Azure Storage SDK for Node.js, which includes a set of convenience libraries that communicate with the storage REST services.

Use Node Package Manager (NPM) to install the package

1. Use a command-line interface such as **PowerShell** (Windows), **Terminal** (Mac), or **Bash** (Unix), and navigate to the folder where you created your application.
2. Type **npm install azure-storage** in the command window. Output from the command is similar to the following example.

```
azure-storage@0.5.0 node_modules\azure-storage
+-- extend@1.2.1
+-- xmlbuilder@0.4.3
+-- mime@1.2.11
+-- node-uuid@1.4.3
+-- validator@3.22.2
+-- underscore@1.4.4
+-- readable-stream@1.0.33 (string_decoder@0.10.31, isarray@0.0.1, inherits@2.0.1, core-util-is@1.0.1)
+-- xml2js@0.2.7 (sax@0.5.2)
+-- request@2.57.0 (caseless@0.10.0, aws-sign2@0.5.0, forever-agent@0.6.1, stringstream@0.0.4, oauth-sign@0.8.0, tunnel-agent@0.4.1, isstream@0.1.2, json-stringify-safe@5.0.1, bl@0.9.4, combined-stream@1.0.5, qs@3.1.0, mime-types@2.0.14, form-data@0.2.0, http-signature@0.11.0, tough-cookie@2.0.0, hawk@2.3.1, har-validator@1.8.0)
```

3. You can manually run the **ls** command to verify that a **node_modules** folder was created. Inside that folder

you will find the **azure-storage** package, which contains the libraries you need to access storage.

Import the package

Add the following code to the top of the **server.js** file in your application:

```
var azure = require('azure-storage');
```

Set up an Azure Storage connection

The azure module will read the environment variables AZURE_STORAGE_ACCOUNT and AZURE_STORAGE_ACCESS_KEY, or AZURE_STORAGE_CONNECTION_STRING for information required to connect to your Azure storage account. If these environment variables are not set, you must specify the account information when calling **TableService**.

For an example of setting the environment variables in the [Azure portal](#) for an Azure Website, see [Node.js web app using the Azure Table Service](#).

Create a table

The following code creates a **TableService** object and uses it to create a new table. Add the following near the top of **server.js**:

```
var tableSvc = azure.createTableService();
```

The call to **createTableIfNotExists** will create a new table with the specified name if it does not already exist. The following example creates a new table named 'mytable' if it does not already exist:

```
tableSvc.createTableIfNotExists('mytable', function(error, result, response){  
    if(!error){  
        // Table exists or created  
    }  
});
```

The `result.created` will be `true` if a new table is created, and `false` if the table already exists. The `response` will contain information about the request.

Filters

Optional filtering operations can be applied to operations performed using **TableService**. Filtering operations can include logging, automatically retrying, etc. Filters are objects that implement a method with the signature:

```
function handle (requestOptions, next)
```

After doing its preprocessing on the request options, the method needs to call "next", passing a callback with the following signature:

```
function (returnObject, finalCallback, next)
```

In this callback, and after processing the `returnObject` (the response from the request to the server), the callback needs to either invoke `next` if it exists to continue processing other filters or simply invoke `finalCallback` otherwise to end the service invocation.

Two filters that implement retry logic are included with the Azure SDK for Node.js, **ExponentialRetryPolicyFilter** and **LinearRetryPolicyFilter**. The following creates a **TableService** object that uses the **ExponentialRetryPolicyFilter**:

```
var retryOperations = new azure.ExponentialRetryPolicyFilter();
var tableSvc = azure.createTableService().withFilter(retryOperations);
```

Add an entity to a table

To add an entity, first create an object that defines your entity properties. All entities must contain a **PartitionKey** and **RowKey**, which are unique identifiers for the entity.

- **PartitionKey** - determines the partition that the entity is stored in
- **RowKey** - uniquely identifies the entity within the partition

Both **PartitionKey** and **RowKey** must be string values. For more information, see [Understanding the Table Service Data Model](#).

The following is an example of defining an entity. Note that **dueDate** is defined as a type of **Edm.DateTime**. Specifying the type is optional, and types will be inferred if not specified.

```
var task = {
    PartitionKey: {'_':'hometasks'},
    RowKey: {'_': '1'},
    description: {'_':'take out the trash'},
    dueDate: {'_':new Date(2015, 6, 20), '$':'Edm.DateTime'}
};
```

NOTE

There is also a **Timestamp** field for each record, which is set by Azure when an entity is inserted or updated.

You can also use the **entityGenerator** to create entities. The following example creates the same task entity using the **entityGenerator**.

```
var entGen = azure.TableUtilities.entityGenerator;
var task = {
    PartitionKey: entGen.String('hometasks'),
    RowKey: entGen.String('1'),
    description: entGen.String('take out the trash'),
    dueDate: entGen.DateTime(new Date(Date.UTC(2015, 6, 20))),
};
```

To add an entity to your table, pass the entity object to the **insertEntity** method.

```
tableSvc.insertEntity('mytable',task, function (error, result, response) {
    if(!error){
        // Entity inserted
    }
});
```

If the operation is successful, **result** will contain the **ETag** of the inserted record and **response** will contain information about the operation.

Example response:

```
{ '.metadata': { etag: 'W/"datetime\''2015-02-25T01%3A22%3A22.5Z\''' } }
```

NOTE

By default, **insertEntity** does not return the inserted entity as part of the `response` information. If you plan on performing other operations on this entity, or wish to cache the information, it can be useful to have it returned as part of the `result`. You can do this by enabling **echoContent** as follows:

```
tableSvc.insertEntity('mytable', task, {echoContent: true}, function (error, result, response) {...})
```

Update an entity

There are multiple methods available to update an existing entity:

- **replaceEntity** - updates an existing entity by replacing it
- **mergeEntity** - updates an existing entity by merging new property values into the existing entity
- **insertOrReplaceEntity** - updates an existing entity by replacing it. If no entity exists, a new one will be inserted
- **insertOrMergeEntity** - updates an existing entity by merging new property values into the existing. If no entity exists, a new one will be inserted

The following example demonstrates updating an entity using **replaceEntity**:

```
tableSvc.replaceEntity('mytable', updatedTask, function(error, result, response){  
    if(!error) {  
        // Entity updated  
    }  
});
```

NOTE

By default, updating an entity does not check to see if the data being updated has previously been modified by another process. To support concurrent updates:

1. Get the ETag of the object being updated. This is returned as part of the `response` for any entity-related operation and can be retrieved through `response['.metadata'].etag`.
2. When performing an update operation on an entity, add the ETag information previously retrieved to the new entity. For example:

```
entity2['.metadata'].etag = currentEtag;
```

1. Perform the update operation. If the entity has been modified since you retrieved the ETag value, such as another instance of your application, an `error` will be returned stating that the update condition specified in the request was not satisfied.

With **replaceEntity** and **mergeEntity**, if the entity that is being updated doesn't exist, then the update operation will fail. Therefore if you wish to store an entity regardless of whether it already exists, use **insertOrReplaceEntity** or **insertOrMergeEntity**.

The `result` for successful update operations will contain the **Etag** of the updated entity.

Work with groups of entities

Sometimes it makes sense to submit multiple operations together in a batch to ensure atomic processing by the server. To accomplish that, use the **TableBatch** class to create a batch, and then use the **executeBatch** method of **TableService** to perform the batched operations.

The following example demonstrates submitting two entities in a batch:

```

var task1 = {
    PartitionKey: {'_':'hometasks'},
    RowKey: {'_': '1'},
    description: {'_':'Take out the trash'},
    dueDate: {'_':new Date(2015, 6, 20)}
};

var task2 = {
    PartitionKey: {'_':'hometasks'},
    RowKey: {'_': '2'},
    description: {'_':'Wash the dishes'},
    dueDate: {'_':new Date(2015, 6, 20)}
};

var batch = new azure.TableBatch();

batch.insertEntity(task1, {echoContent: true});
batch.insertEntity(task2, {echoContent: true});

tableSvc.executeBatch('mytable', batch, function (error, result, response) {
    if(!error) {
        // Batch completed
    }
});
```

For successful batch operations, `result` will contain information for each operation in the batch.

Work with batched operations

Operations added to a batch can be inspected by viewing the `operations` property. You can also use the following methods to work with operations:

- **clear** - clears all operations from a batch
- **getOperations** - gets an operation from the batch
- **hasOperations** - returns true if the batch contains operations
- **removeOperations** - removes an operation
- **size** - returns the number of operations in the batch

Retrieve an entity by key

To return a specific entity based on the **PartitionKey** and **RowKey**, use the **retrieveEntity** method.

```

tableSvc.retrieveEntity('mytable', 'hometasks', '1', function(error, result, response){
    if(!error){
        // result contains the entity
    }
});
```

Once this operation is complete, `result` will contain the entity.

Query a set of entities

To query a table, use the **TableQuery** object to build up a query expression using the following clauses:

- **select** - the fields to be returned from the query
- **where** - the where clause
 - **and** - an `and` where condition
 - **or** - an `or` where condition
- **top** - the number of items to fetch

The following example builds a query that will return the top five items with a PartitionKey of 'hometasks'.

```
var query = new azure.TableQuery()
    .top(5)
    .where('PartitionKey eq ?', 'hometasks');
```

Since **select** is not used, all fields will be returned. To perform the query against a table, use **queryEntities**. The following example uses this query to return entities from 'mytable'.

```
tableSvc.queryEntities('mytable',query, null, function(error, result, response) {
    if(!error) {
        // query was successful
    }
});
```

If successful, `result.entries` will contain an array of entities that match the query. If the query was unable to return all entities, `result.continuationToken` will be non-*null* and can be used as the third parameter of **queryEntities** to retrieve more results. For the initial query, use *null* for the third parameter.

Query a subset of entity properties

A query to a table can retrieve just a few fields from an entity. This reduces bandwidth and can improve query performance, especially for large entities. Use the **select** clause and pass the names of the fields to be returned. For example, the following query will return only the **description** and **dueDate** fields.

```
var query = new azure.TableQuery()
    .select(['description', 'dueDate'])
    .top(5)
    .where('PartitionKey eq ?', 'hometasks');
```

Delete an entity

You can delete an entity using its partition and row keys. In this example, the **task1** object contains the **RowKey** and **PartitionKey** values of the entity to be deleted. Then the object is passed to the **deleteEntity** method.

```
var task = {
    PartitionKey: {'_': 'hometasks'},
    RowKey: {'_': '1'}
};

tableSvc.deleteEntity('mytable', task, function(error, response){
    if(!error) {
        // Entity deleted
    }
});
```

NOTE

Consider using ETags when deleting items, to ensure that the item hasn't been modified by another process. See [Update an entity](#) for information on using ETags.

Delete a table

The following code deletes a table from a storage account.

```
tableSvc.deleteTable('mytable', function(error, response){
    if(!error){
        // Table deleted
    }
});
```

If you are uncertain whether the table exists, use **deleteTableIfExists**.

Use continuation tokens

When you are querying tables for large amounts of results, look for continuation tokens. There may be large amounts of data available for your query that you might not realize if you do not build to recognize when a continuation token is present.

The results object returned during querying entities sets a `continuationToken` property when such a token is present. You can then use this when performing a query to continue to move across the partition and table entities.

When querying, a `continuationToken` parameter may be provided between the query object instance and the callback function:

```
var nextContinuationToken = null;
dc.table.queryEntities(tableName,
    query,
    nextContinuationToken,
    function (error, results) {
        if (error) throw error;

        // iterate through results.entries with results

        if (results.continuationToken) {
            nextContinuationToken = results.continuationToken;
        }

    });
});
```

If you inspect the `continuationToken` object, you will find properties such as `nextPartitionKey`, `nextRowKey`, and `targetLocation`, which can be used to iterate through all the results.

There is also a continuation sample within the Azure Storage Node.js repo on GitHub. Look for `examples/samples/continuationsample.js`.

Work with shared access signatures

Shared access signatures (SAS) are a secure way to provide granular access to tables without providing your storage account name or keys. SAS are often used to provide limited access to your data, such as allowing a mobile app to query records.

A trusted application such as a cloud-based service generates a SAS using the **generateSharedAccessSignature** of the **TableService**, and provides it to an untrusted or semi-trusted application such as a mobile app. The SAS is generated using a policy, which describes the start and end dates during which the SAS is valid, as well as the access level granted to the SAS holder.

The following example generates a new shared access policy that will allow the SAS holder to query ('r') the table, and expires 100 minutes after the time it is created.

```

var startDate = new Date();
var expiryDate = new Date(startDate);
expiryDate.setMinutes(startDate.getMinutes() + 100);
startDate.setMinutes(startDate.getMinutes() - 100);

var sharedAccessPolicy = {
    AccessPolicy: {
        Permissions: azure.TableUtilities.SharedAccessPermissions.QUERY,
        Start: startDate,
        Expiry: expiryDate
    },
};

var tableSAS = tableSvc.generateSharedAccessSignature('mytable', sharedAccessPolicy);
var host = tableSvc.host;

```

Note that the host information must be provided also, as it is required when the SAS holder attempts to access the table.

The client application then uses the SAS with **TableServiceWithSAS** to perform operations against the table. The following example connects to the table and performs a query.

```

var sharedTableService = azure.createTableServiceWithSas(host, tableSAS);
var query = azure.TableQuery()
    .where('PartitionKey eq ?', 'hometasks');

sharedTableService.queryEntities(query, null, function(error, result, response) {
    if(!error) {
        // result contains the entities
    }
});

```

Since the SAS was generated with only query access, if an attempt were made to insert, update, or delete entities, an error would be returned.

Access Control Lists

You can also use an Access Control List (ACL) to set the access policy for a SAS. This is useful if you wish to allow multiple clients to access the table, but provide different access policies for each client.

An ACL is implemented using an array of access policies, with an ID associated with each policy. The following example defines two policies, one for 'user1' and one for 'user2':

```

var sharedAccessPolicy = {
    user1: {
        Permissions: azure.TableUtilities.SharedAccessPermissions.QUERY,
        Start: startDate,
        Expiry: expiryDate
    },
    user2: {
        Permissions: azure.TableUtilities.SharedAccessPermissions.ADD,
        Start: startDate,
        Expiry: expiryDate
    }
};

```

The following example gets the current ACL for the **hometasks** table, and then adds the new policies using **setTableAcl**. This approach allows:

```
var extend = require('extend');
tableSvc.getTableAcl('hometasks', function(error, result, response) {
if(!error){
    var newSignedIdentifiers = extend(true, result.signedIdentifiers, sharedAccessPolicy);
    tableSvc.setTableAcl('hometasks', newSignedIdentifiers, function(error, result, response){
        if(!error){
            // ACL set
        }
    });
}
});
```

Once the ACL has been set, you can then create a SAS based on the ID for a policy. The following example creates a new SAS for 'user2':

```
tableSAS = tableSvc.generateSharedAccessSignature('hometasks', { Id: 'user2' });
```

Next steps

For more information, see the following resources.

- [Azure Storage Team Blog](#).
- [Azure Storage SDK for Node](#) repository on GitHub.
- [Node.js Developer Center](#)

How to use Table storage from C++

1/17/2017 • 15 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide will show you how to perform common scenarios by using the Azure Table storage service. The samples are written in C++ and use the [Azure Storage Client Library for C++](#). The scenarios covered include **creating and deleting a table** and **working with table entities**.

NOTE

This guide targets the Azure Storage Client Library for C++ version 1.0.0 and above. The recommended version is Storage Client Library 2.2.0, which is available via [NuGet](#) or [GitHub](#).

What is the Table Service

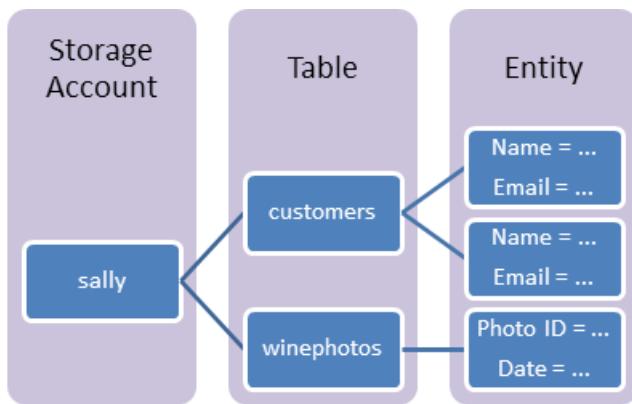
The Azure Table storage service stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of the Table service include:

- Storing TBs of structured data capable of serving web scale applications
- Storing datasets that don't require complex joins, foreign keys, or stored procedures and can be denormalized for fast access
- Quickly querying data using a clustered index
- Accessing data using the OData protocol and LINQ queries with WCF Data Service .NET Libraries

You can use the Table service to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table Service Concepts

The Table service contains the following components:



- **URL format:** Code addresses tables in an account using this address format:

`http://<storage account>.table.core.windows.net/<table>`

You can address Azure tables directly using this address with the OData protocol. For more information, see [OData.org](#)

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Table:** A table is a collection of entities. Tables don't enforce a schema on entities, which means a single table can contain entities that have different sets of properties. The number of tables that a storage account can contain is limited only by the storage account capacity limit.
- **Entity:** An entity is a set of properties, similar to a database row. An entity can be up to 1MB in size.
- **Properties:** A property is a name-value pair. Each entity can include up to 252 properties to store data. Each entity also has 3 system properties that specify a partition key, a row key, and a timestamp. Entities with the same partition key can be queried more quickly, and inserted/updated in atomic operations. An entity's row key is its unique identifier within a partition.

For details about naming tables and properties, see [Understanding the Table Service Data Model](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a C++ application

In this guide, you will use storage features that can be run within a C++ application. To do so, you will need to install the Azure Storage Client Library for C++ and create an Azure storage account in your Azure subscription.

To install the Azure Storage Client Library for C++, you can use the following methods:

- **Linux:** Follow the instructions given on the [Azure Storage Client Library for C++ README](#) page.
- **Windows:** In Visual Studio, click **Tools > NuGet Package Manager > Package Manager Console**. Type the following command into the [NuGet Package Manager console](#) and press Enter.

`Install-Package wastorage`

Configure your application to access Table storage

Add the following include statements to the top of the C++ file where you want to use the Azure storage APIs to access tables:

```
#include "was/storage_account.h"
#include "was/table.h"
```

Set up an Azure storage connection string

An Azure storage client uses a storage connection string to store endpoints and credentials for accessing data management services. When running a client application, you must provide the storage connection string in the following format. Use the name of your storage account and the storage access key for the storage account listed in the [Azure Portal](#) for the *AccountName* and *AccountKey* values. For information on storage accounts and access keys, see [About Azure storage accounts](#). This example shows how you can declare a static field to hold the connection string:

```
// Define the connection string with your values.
const utility::string_t
storage_connection_string(U("DefaultEndpointsProtocol=https;AccountName=your_storage_account;AccountKey=your_storage_account_key"));
```

To test your application in your local Windows-based computer, you can use the Azure [storage emulator](#) that is installed with the [Azure SDK](#). The storage emulator is a utility that simulates the Azure Blob, Queue, and Table services available on your local development machine. The following example shows how you can declare a static field to hold the connection string to your local storage emulator:

```
// Define the connection string with Azure storage emulator.
const utility::string_t storage_connection_string(U("UseDevelopmentStorage=true;"));
```

To start the Azure storage emulator, click the **Start** button or press the Windows key. Begin typing **Azure Storage Emulator**, and then select **Microsoft Azure Storage Emulator** from the list of applications.

The following samples assume that you have used one of these two methods to get the storage connection string.

Retrieve your connection string

You can use the **cloud_storage_account** class to represent your storage account information. To retrieve your storage account information from the storage connection string, you can use the *parse* method.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);
```

Next, get a reference to a **cloud_table_client** class, as it lets you get reference objects for tables and entities stored within the Table storage service. The following code creates a **cloud_table_client** object by using the storage account object we retrieved above:

```
// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();
```

Create a table

A **cloud_table_client** object lets you get reference objects for tables and entities. The following code creates a

cloud_table_client object and uses it to create a new table.

```
// Retrieve the storage account from the connection string.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the table client.  
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();  
  
// Retrieve a reference to a table.  
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));  
  
// Create the table if it doesn't exist.  
table.create_if_not_exists();
```

Add an entity to a table

To add an entity to a table, create a new **table_entity** object and pass it to **table_operation::insert_entity**. The following code uses the customer's first name as the row key and last name as the partition key. Together, an entity's partition and row key uniquely identify the entity in the table. Entities with the same partition key can be queried faster than those with different partition keys, but using diverse partition keys allows for greater parallel operation scalability. For more information, see [Microsoft Azure storage performance and scalability checklist](#).

The following code creates a new instance of **table_entity** with some customer data to be stored. The code next calls **table_operation::insert_entity** to create a **table_operation** object to insert an entity into a table, and associates the new table entity with it. Finally, the code calls the execute method on the **cloud_table** object. And the new **table_operation** sends a request to the Table service to insert the new customer entity into the "people" table.

```
// Retrieve the storage account from the connection string.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the table client.  
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();  
  
// Retrieve a reference to a table.  
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));  
  
// Create the table if it doesn't exist.  
table.create_if_not_exists();  
  
// Create a new customer entity.  
azure::storage::table_entity customer1(U("Harp"), U("Walter"));  
  
azure::storage::table_entity::properties_type& properties = customer1.properties();  
properties.reserve(2);  
properties[U("Email")] = azure::storage::entity_property(U("Walter@contoso.com"));  
  
properties[U("Phone")] = azure::storage::entity_property(U("425-555-0101"));  
  
// Create the table operation that inserts the customer entity.  
azure::storage::table_operation insert_operation = azure::storage::table_operation::insert_entity(customer1);  
  
// Execute the insert operation.  
azure::storage::table_result insert_result = table.execute(insert_operation);
```

Insert a batch of entities

You can insert a batch of entities to the Table service in one write operation. The following code creates a **table_batch_operation** object, and then adds three insert operations to it. Each insert operation is added by

creating a new entity object, setting its values, and then calling the insert method on the **table_batch_operation** object to associate the entity with a new insert operation. Then, **cloud_table.execute** is called to execute the operation.

```
// Retrieve the storage account from the connection string.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the table client.  
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();  
  
// Create a cloud table object for the table.  
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));  
  
// Define a batch operation.  
azure::storage::table_batch_operation batch_operation;  
  
// Create a customer entity and add it to the table.  
azure::storage::table_entity customer1(U("Smith"), U("Jeff"));  
  
azure::storage::table_entity::properties_type& properties1 = customer1.properties();  
properties1.reserve(2);  
properties1[U("Email")] = azure::storage::entity_property(U("Jeff@contoso.com"));  
properties1[U("Phone")] = azure::storage::entity_property(U("425-555-0104"));  
  
// Create another customer entity and add it to the table.  
azure::storage::table_entity customer2(U("Smith"), U("Ben"));  
  
azure::storage::table_entity::properties_type& properties2 = customer2.properties();  
properties2.reserve(2);  
properties2[U("Email")] = azure::storage::entity_property(U("Ben@contoso.com"));  
properties2[U("Phone")] = azure::storage::entity_property(U("425-555-0102"));  
  
// Create a third customer entity to add to the table.  
azure::storage::table_entity customer3(U("Smith"), U("Denise"));  
  
azure::storage::table_entity::properties_type& properties3 = customer3.properties();  
properties3.reserve(2);  
properties3[U("Email")] = azure::storage::entity_property(U("Denise@contoso.com"));  
properties3[U("Phone")] = azure::storage::entity_property(U("425-555-0103"));  
  
// Add customer entities to the batch insert operation.  
batch_operation.insert_or_replace_entity(customer1);  
batch_operation.insert_or_replace_entity(customer2);  
batch_operation.insert_or_replace_entity(customer3);  
  
// Execute the batch operation.  
std::vector<azure::storage::table_result> results = table.execute_batch(batch_operation);
```

Some things to note on batch operations:

- You can perform up to 100 insert, delete, merge, replace, insert-or-merge, and insert-or-replace operations in any combination in a single batch.
- A batch operation can have a retrieve operation, if it is the only operation in the batch.
- All entities in a single batch operation must have the same partition key.
- A batch operation is limited to a 4-MB data payload.

Retrieve all entities in a partition

To query a table for all entities in a partition, use a **table_query** object. The following code example specifies a filter for entities where 'Smith' is the partition key. This example prints the fields of each entity in the query results to the console.

```

// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Construct the query operation for all customer entities where PartitionKey="Smith".
azure::storage::table_query query;

query.set_filter_string(azure::storage::table_query::generate_filter_condition(U("PartitionKey"),
azure::storage::query_comparison_operator::equal, U("Smith")));

// Execute the query.
azure::storage::table_query_iterator it = table.execute_query(query);

// Print the fields for each customer.
azure::storage::table_query_iterator end_of_results;
for (; it != end_of_results; ++it)
{
    const azure::storage::table_entity::properties_type& properties = it->properties();

    std::wcout << U("PartitionKey: ") << it->partition_key() << U(", RowKey: ") << it->row_key()
        << U(", Property1: ") << properties.at(U("Email")).string_value()
        << U(", Property2: ") << properties.at(U("Phone")).string_value() << std::endl;
}

```

The query in this example brings all the entities that match the filter criteria. If you have large tables and need to download the table entities often, we recommend that you store your data in Azure storage blobs instead.

Retrieve a range of entities in a partition

If you don't want to query all the entities in a partition, you can specify a range by combining the partition key filter with a row key filter. The following code example uses two filters to get all entities in partition 'Smith' where the row key (first name) starts with a letter earlier than 'E' in the alphabet and then prints the query results.

```

// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Create the table query.
azure::storage::table_query query;

query.set_filter_string(azure::storage::table_query::combine_filter_conditions(
    azure::storage::table_query::generate_filter_condition(U("PartitionKey"),
    azure::storage::query_comparison_operator::equal, U("Smith")),
    azure::storage::query_logical_operator::op_and,
    azure::storage::table_query::generate_filter_condition(U("RowKey"),
    azure::storage::query_comparison_operator::less_than, U("E"))));

// Execute the query.
azure::storage::table_query_iterator it = table.execute_query(query);

// Loop through the results, displaying information about the entity.
azure::storage::table_query_iterator end_of_results;
for (; it != end_of_results; ++it)
{
    const azure::storage::table_entity::properties_type& properties = it->properties();

    std::wcout << U("PartitionKey: ") << it->partition_key() << U(", RowKey: ") << it->row_key()
        << U(", Property1: ") << properties.at(U("Email")).string_value()
        << U(", Property2: ") << properties.at(U("Phone")).string_value() << std::endl;
}

```

Retrieve a single entity

You can write a query to retrieve a single, specific entity. The following code uses **table_operation::retrieve_entity** to specify the customer 'Jeff Smith'. This method returns just one entity, rather than a collection, and the returned value is in **table_result**. Specifying both partition and row keys in a query is the fastest way to retrieve a single entity from the Table service.

```

azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Retrieve the entity with partition key of "Smith" and row key of "Jeff".
azure::storage::table_operation retrieve_operation =
azure::storage::table_operation::retrieve_entity(U("Smith"), U("Jeff"));
azure::storage::table_result retrieve_result = table.execute(retrieve_operation);

// Output the entity.
azure::storage::table_entity entity = retrieve_result.entity();
const azure::storage::table_entity::properties_type& properties = entity.properties();

std::wcout << U("PartitionKey: ") << entity.partition_key() << U(", RowKey: ") << entity.row_key()
    << U(", Property1: ") << properties.at(U("Email")).string_value()
    << U(", Property2: ") << properties.at(U("Phone")).string_value() << std::endl;

```

Replace an entity

To replace an entity, retrieve it from the Table service, modify the entity object, and then save the changes back to the Table service. The following code changes an existing customer's phone number and email address. Instead of calling **table_operation::insert_entity**, this code uses **table_operation::replace_entity**. This causes the entity to be fully replaced on the server, unless the entity on the server has changed since it was retrieved, in which case the operation will fail. This failure is to prevent your application from inadvertently overwriting a change made between the retrieval and update by another component of your application. The proper handling of this failure is to retrieve the entity again, make your changes (if still valid), and then perform another **table_operation::replace_entity** operation. The next section will show you how to override this behavior.

```
// Retrieve the storage account from the connection string.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the table client.  
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();  
  
// Create a cloud table object for the table.  
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));  
  
// Replace an entity.  
azure::storage::table_entity entity_to_replace(U("Smith"), U("Jeff"));  
azure::storage::table_entity::properties_type& properties_to_replace = entity_to_replace.properties();  
properties_to_replace.reserve(2);  
  
// Specify a new phone number.  
properties_to_replace[U("Phone")] = azure::storage::entity_property(U("425-555-0106"));  
  
// Specify a new email address.  
properties_to_replace[U("Email")] = azure::storage::entity_property(U("Jeffs@contoso.com"));  
  
// Create an operation to replace the entity.  
azure::storage::table_operation replace_operation =  
    azure::storage::table_operation::replace_entity(entity_to_replace);  
  
// Submit the operation to the Table service.  
azure::storage::table_result replace_result = table.execute(replace_operation);
```

Insert-or-replace an entity

table_operation::replace_entity operations will fail if the entity has been changed since it was retrieved from the server. Furthermore, you must retrieve the entity from the server first in order for **table_operation::replace_entity** to be successful. Sometimes, however, you don't know if the entity exists on the server and the current values stored in it are irrelevant—your update should overwrite them all. To accomplish this, you would use a **table_operation::insert_or_replace_entity** operation. This operation inserts the entity if it doesn't exist, or replaces it if it does, regardless of when the last update was made. In the following code example, the customer entity for Jeff Smith is still retrieved, but it is then saved back to the server via **table_operation::insert_or_replace_entity**. Any updates made to the entity between the retrieval and update operation will be overwritten.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Insert-or-replace an entity.
azure::storage::table_entity entity_to_insert_or_replace(U("Smith"), U("Jeff"));
azure::storage::table_entity::properties_type& properties_to_insert_or_replace =
entity_to_insert_or_replace.properties();

properties_to_insert_or_replace.reserve(2);

// Specify a phone number.
properties_to_insert_or_replace[U("Phone")] = azure::storage::entity_property(U("425-555-0107"));

// Specify an email address.
properties_to_insert_or_replace[U("Email")] = azure::storage::entity_property(U("Jeffsm@contoso.com"));

// Create an operation to insert-or-replace the entity.
azure::storage::table_operation insert_or_replace_operation =
azure::storage::table_operation::insert_or_replace_entity(entity_to_insert_or_replace);

// Submit the operation to the Table service.
azure::storage::table_result insert_or_replace_result = table.execute(insert_or_replace_operation);
```

Query a subset of entity properties

A query to a table can retrieve just a few properties from an entity. The query in the following code uses the **table_query::set_select_columns** method to return only the email addresses of entities in the table.

```

// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Define the query, and select only the Email property.
azure::storage::table_query query;
std::vector<utility::string_t> columns;

columns.push_back(U("Email"));
query.set_select_columns(columns);

// Execute the query.
azure::storage::table_query_iterator it = table.execute_query(query);

// Display the results.
azure::storage::table_query_iterator end_of_results;
for (; it != end_of_results; ++it)
{
    std::wcout << U("PartitionKey: ") << it->partition_key() << U(", RowKey: ") << it->row_key();

    const azure::storage::table_entity::properties_type& properties = it->properties();
    for (auto prop_it = properties.begin(); prop_it != properties.end(); ++prop_it)
    {
        std::wcout << ", " << prop_it->first << ":" << prop_it->second.str();
    }

    std::wcout << std::endl;
}

```

NOTE

Querying a few properties from an entity is a more efficient operation than retrieving all properties.

Delete an entity

You can easily delete an entity after you have retrieved it. Once the entity is retrieved, call **table_operation::delete_entity** with the entity to delete. Then call the **cloud_table.execute** method. The following code retrieves and deletes an entity with a partition key of "Smith" and a row key of "Jeff".

```

// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Create an operation to retrieve the entity with partition key of "Smith" and row key of "Jeff".
azure::storage::table_operation retrieve_operation =
azure::storage::table_operation::retrieve_entity(U("Smith"), U("Jeff"));
azure::storage::table_result retrieve_result = table.execute(retrieve_operation);

// Create an operation to delete the entity.
azure::storage::table_operation delete_operation =
azure::storage::table_operation::delete_entity(retrieve_result.entity());

// Submit the delete operation to the Table service.
azure::storage::table_result delete_result = table.execute(delete_operation);

```

Delete a table

Finally, the following code example deletes a table from a storage account. A table that has been deleted will be unavailable to be re-created for a period of time following the deletion.

```

// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Create an operation to retrieve the entity with partition key of "Smith" and row key of "Jeff".
azure::storage::table_operation retrieve_operation =
azure::storage::table_operation::retrieve_entity(U("Smith"), U("Jeff"));
azure::storage::table_result retrieve_result = table.execute(retrieve_operation);

// Create an operation to delete the entity.
azure::storage::table_operation delete_operation =
azure::storage::table_operation::delete_entity(retrieve_result.entity());

// Submit the delete operation to the Table service.
azure::storage::table_result delete_result = table.execute(delete_operation);

```

Next steps

Now that you've learned the basics of table storage, follow these links to learn more about Azure Storage:

- [How to use Blob storage from C++](#)
- [How to use Queue storage from C++](#)
- [List Azure Storage resources in C++](#)
- [Storage Client Library for C++ reference](#)
- [Azure Storage documentation](#)

How to use Table storage from Python

1/17/2017 • 6 min to read • [Edit on GitHub](#)

TIP

[Try the Microsoft Azure Storage Explorer](#)

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide shows you how to perform common scenarios by using the Azure Table storage service. The samples are written in Python and use the [Microsoft Azure Storage SDK for Python](#). The covered scenarios include creating and deleting a table, in addition to inserting and querying entities in a table.

What is the Table Service

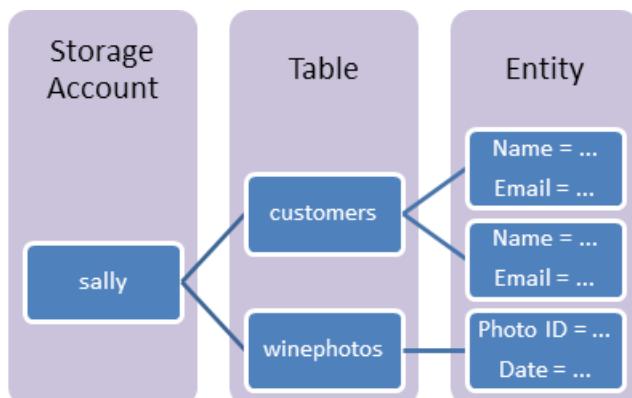
The Azure Table storage service stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of the Table service include:

- Storing TBs of structured data capable of serving web scale applications
- Storing datasets that don't require complex joins, foreign keys, or stored procedures and can be denormalized for fast access
- Quickly querying data using a clustered index
- Accessing data using the OData protocol and LINQ queries with WCF Data Service .NET Libraries

You can use the Table service to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table Service Concepts

The Table service contains the following components:



- **URL format:** Code addresses tables in an account using this address format:

`http://<storage account>.table.core.windows.net/<table>`

You can address Azure tables directly using this address with the OData protocol. For more information,

see [OData.org](#)

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Table:** A table is a collection of entities. Tables don't enforce a schema on entities, which means a single table can contain entities that have different sets of properties. The number of tables that a storage account can contain is limited only by the storage account capacity limit.
- **Entity:** An entity is a set of properties, similar to a database row. An entity can be up to 1MB in size.
- **Properties:** A property is a name-value pair. Each entity can include up to 252 properties to store data. Each entity also has 3 system properties that specify a partition key, a row key, and a timestamp. Entities with the same partition key can be queried more quickly, and inserted/updated in atomic operations. An entity's row key is its unique identifier within a partition.

For details about naming tables and properties, see [Understanding the Table Service Data Model](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a table

The **TableService** object lets you work with table services. The following code creates a **TableService** object. Add the code near the top of any Python file in which you wish to programmatically access Azure Storage:

```
from azure.storage.table import TableService, Entity
```

The following code creates a **TableService** object by using the storage account name and account key. Replace 'myaccount' and 'mykey' with your account name and key.

```
table_service = TableService(account_name='myaccount', account_key='mykey')

table_service.create_table('tasktable')
```

Add an entity to a table

To add an entity, first create a dictionary or Entity that defines your entity property names and values. Note that for every entity, you must specify **PartitionKey** and **RowKey**. These are the unique identifiers of your entities. You can query using these values much faster than you can query your other properties. The system uses **PartitionKey** to automatically distribute the table entities over many storage nodes. Entities that have the same **PartitionKey** are stored on the same node. **RowKey** is the unique ID of the entity within the partition that it belongs to.

To add an entity to your table, pass a dictionary object to the **insert_entity** method.

```
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '1', 'description' : 'Take out the trash', 'priority' : 200}
table_service.insert_entity('tasktable', task)
```

You can also pass an instance of the **Entity** class to the **insert_entity** method.

```
task = Entity()
task.PartitionKey = 'tasksSeattle'
task.RowKey = '2'
task.description = 'Wash the car'
task.priority = 100
table_service.insert_entity('tasktable', task)
```

Update an entity

This code shows how to replace the old version of an existing entity with an updated version.

```
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '1', 'description' : 'Take out the garbage', 'priority' : 250}
table_service.update_entity('tasktable', 'tasksSeattle', '1', task, content_type='application/atom+xml')
```

If the entity that is being updated does not exist, then the update operation will fail. If you want to store an entity regardless of whether it existed before, use **insert_or_replace_entity**. In the following example, the first call will replace the existing entity. The second call will insert a new entity, since no entity with the specified **PartitionKey** and **RowKey** exists in the table.

```
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '1', 'description' : 'Take out the garbage again', 'priority' : 250}
table_service.insert_or_replace_entity('tasktable', 'tasksSeattle', '1', task,
content_type='application/atom+xml')

task = {'PartitionKey': 'tasksSeattle', 'RowKey': '3', 'description' : 'Buy detergent', 'priority' : 300}
table_service.insert_or_replace_entity('tasktable', 'tasksSeattle', '1', task,
content_type='application/atom+xml')
```

Change a group of entities

Sometimes it makes sense to submit multiple operations together in a batch to ensure atomic processing by the server. To accomplish that, you use the **TableBatch** class. When you do want to submit the batch, you call **commit_batch**. Note that all entities must be in the same partition in order to be changed as a batch. The example below adds two entities together in a batch.

```
from azure.storage.table import TableBatch
batch = TableBatch()
task10 = {'PartitionKey': 'tasksSeattle', 'RowKey': '10', 'description' : 'Go grocery shopping', 'priority' : 400}
task11 = {'PartitionKey': 'tasksSeattle', 'RowKey': '11', 'description' : 'Clean the bathroom', 'priority' : 100}
batch.insert_entity(task10)
batch.insert_entity(task11)
table_service.commit_batch('tasktable', batch)
```

Batches can also be used with the context manager syntax:

```
task12 = {'PartitionKey': 'tasksSeattle', 'RowKey': '12', 'description' : 'Go grocery shopping', 'priority' : 400}
task13 = {'PartitionKey': 'tasksSeattle', 'RowKey': '13', 'description' : 'Clean the bathroom', 'priority' : 100}

with table_service.batch('tasktable') as batch:
    batch.insert_entity(task12)
    batch.insert_entity(task13)
```

Query for an entity

To query an entity in a table, use the **get_entity** method by passing **PartitionKey** and **RowKey**.

```
task = table_service.get_entity('tasktable', 'tasksSeattle', '1')
print(task.description)
print(task.priority)
```

Query a set of entities

This example finds all tasks in Seattle based on **PartitionKey**.

```
tasks = table_service.query_entities('tasktable', filter="PartitionKey eq 'tasksSeattle'")
for task in tasks:
    print(task.description)
    print(task.priority)
```

Query a subset of entity properties

A query to a table can retrieve just a few properties from an entity. This technique, called *projection*, reduces bandwidth and can improve query performance, especially for large entities. Use the **select** parameter and pass the names of the properties that you want to bring over to the client.

The query in the following code returns only the descriptions of entities in the table.

NOTE

The following snippet works only against the Azure Storage. It is not supported by the storage emulator.

```
tasks = table_service.query_entities('tasktable', filter="PartitionKey eq 'tasksSeattle'", 
select='description')
for task in tasks:
    print(task.description)
```

Delete an entity

You can delete an entity by using its partition and row key.

```
table_service.delete_entity('tasktable', 'tasksSeattle', '1')
```

Delete a table

The following code deletes a table from a storage account.

```
table_service.delete_table('tasktable')
```

Next steps

Now that you've learned the basics of Table storage, follow these links to learn more.

- [Python Developer Center](#)
- [Azure Storage Services REST API](#)

- [Azure Storage Team Blog](#)
- [Microsoft Azure Storage SDK for Python](#)

How to use table storage from PHP

1/17/2017 • 13 min to read • [Edit on GitHub](#)

TIP

[Try the Microsoft Azure Storage Explorer](#)

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide shows you how to perform common scenarios using the Azure Table service. The samples are written in PHP and use the [Azure SDK for PHP](#). The scenarios covered include **creating and deleting a table, and inserting, deleting, and querying entities in a table**. For more information on the Azure Table service, see the [Next steps](#) section.

What is the Table Service

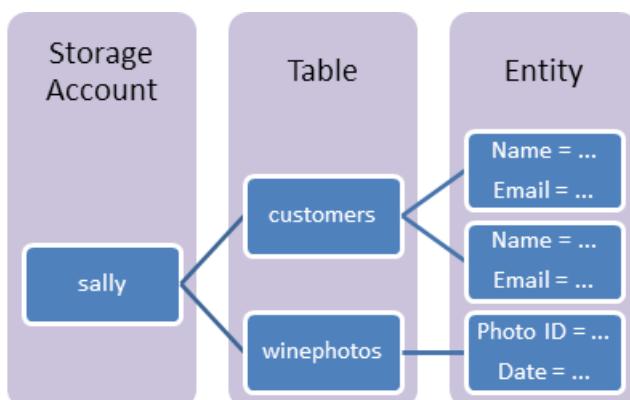
The Azure Table storage service stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of the Table service include:

- Storing TBs of structured data capable of serving web scale applications
- Storing datasets that don't require complex joins, foreign keys, or stored procedures and can be denormalized for fast access
- Quickly querying data using a clustered index
- Accessing data using the OData protocol and LINQ queries with WCF Data Service .NET Libraries

You can use the Table service to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table Service Concepts

The Table service contains the following components:



- **URL format:** Code addresses tables in an account using this address format:
`http://<storage account>.table.core.windows.net/<table>`

You can address Azure tables directly using this address with the OData protocol. For more information, see [OData.org](#)

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Table:** A table is a collection of entities. Tables don't enforce a schema on entities, which means a single table can contain entities that have different sets of properties. The number of tables that a storage account can contain is limited only by the storage account capacity limit.
- **Entity:** An entity is a set of properties, similar to a database row. An entity can be up to 1MB in size.
- **Properties:** A property is a name-value pair. Each entity can include up to 252 properties to store data. Each entity also has 3 system properties that specify a partition key, a row key, and a timestamp. Entities with the same partition key can be queried more quickly, and inserted/updated in atomic operations. An entity's row key is its unique identifier within a partition.

For details about naming tables and properties, see [Understanding the Table Service Data Model](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a PHP application

The only requirement for creating a PHP application that accesses the Azure Table service is the referencing of classes in the Azure SDK for PHP from within your code. You can use any development tools to create your application, including Notepad.

In this guide, you use Table service features which can be called from within a PHP application locally, or in code running within an Azure web role, worker role, or website.

Get the Azure Client Libraries

Install via Composer

1. [Install Git](#). Note that on Windows, you must also add the Git executable to your PATH environment variable.
2. Create a file named **composer.json** in the root of your project and add the following code to it:

```
{  
    "require": {  
        "microsoft/windowsazure": "^0.4"  
    }  
}
```

3. Download [composer.phar](#) in your project root.
4. Open a command prompt and execute the following command in your project root

```
php composer.phar install
```

Configure your application to access the Table service

To use the Azure Table service APIs, you need to:

1. Reference the autoloader file using the `require_once` statement, and
2. Reference any classes you might use.

The following example shows how to include the autoloader file and reference the **ServicesBuilder** class.

NOTE

The examples in this article assume you have installed the PHP Client Libraries for Azure via Composer. If you installed the libraries manually, you need to reference the `WindowsAzure.php` autoloader file.

```
require_once 'vendor/autoload.php';
use WindowsAzure\Common\ServicesBuilder;
```

In the examples below, the `require_once` statement is always shown, but only the classes necessary for the example to execute are referenced.

Set up an Azure storage connection

To instantiate an Azure Table service client, you must first have a valid connection string. The format for the Table service connection string is:

For accessing a live service:

```
DefaultEndpointsProtocol=[http|https];AccountName=[yourAccount];AccountKey=[yourKey]
```

For accessing the emulator storage:

```
UseDevelopmentStorage=true
```

To create any Azure service client, you need to use the **ServicesBuilder** class. You can:

- pass the connection string directly to it or
- use the **CloudConfigurationManager (CCM)** to check multiple external sources for the connection string:
 - by default, it comes with support for one external source - environmental variables
 - you can add new sources by extending the **ConnectionStringSource** class

For the examples outlined here, the connection string will be passed directly.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;

$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);
```

Create a table

A **TableRestProxy** object lets you create a table with the **createTable** method. When creating a table, you can set the Table service timeout. (For more information about the Table service timeout, see [Setting Timeouts for Table Service Operations](#).)

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

try {
    // Create table.
    $tableRestProxy->createTable("mytable");
}
catch(ServiceException $e){
    $code = $e->getCode();
    $error_message = $e->getMessage();
    // Handle exception based on error codes and messages.
    // Error codes and messages can be found here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
}

```

For information about restrictions on table names, see [Understanding the Table Service Data Model](#).

Add an entity to a table

To add an entity to a table, create a new **Entity** object and pass it to **TableRestProxy->insertEntity**. Note that when you create an entity, you must specify a **PartitionKey** and **RowKey**. These are the unique identifiers for an entity and are values that can be queried much faster than other entity properties. The system uses **PartitionKey** to automatically distribute the table's entities over many storage nodes. Entities with the same **PartitionKey** are stored on the same node. (Operations on multiple entities stored on the same node perform better than on entities stored across different nodes.) The **RowKey** is the unique ID of an entity within a partition.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;
use MicrosoftAzure\Storage\Table\Models\Entity;
use MicrosoftAzure\Storage\Table\Models\EdmType;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

$entity = new Entity();
$entity->setPartitionKey("tasksSeattle");
$entity->setRowKey("1");
$entity->addProperty("Description", null, "Take out the trash.");
$entity->addProperty("DueDate",
                     EdmType::DATETIME,
                     new DateTime("2012-11-05T08:15:00-08:00"));
$entity->addProperty("Location", EdmType::STRING, "Home");

try{
    $tableRestProxy->insertEntity("mytable", $entity);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
}

```

For information about Table properties and types, see [Understanding the Table Service Data Model](#).

The **TableRestProxy** class offers two alternative methods for inserting entities: **insertOrMergeEntity** and **insertOrReplaceEntity**. To use these methods, create a new **Entity** and pass it as a parameter to either method. Each method will insert the entity if it does not exist. If the entity already exists, **insertOrMergeEntity** updates property values if the properties already exist and adds new properties if they do not exist, while **insertOrReplaceEntity** completely replaces an existing entity. The following example shows how to use **insertOrMergeEntity**. If the entity with `PartitionKey` "tasksSeattle" and `RowKey` "1" does not already exist, it will be inserted. However, if it has previously been inserted (as shown in the example above), the `DueDate` property will be updated, and the `Status` property will be added. The `Description` and `Location` properties are also updated, but with values that effectively leave them unchanged. If these latter two properties were not added as shown in the example, but existed on the target entity, their existing values would remain unchanged.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;
use MicrosoftAzure\Storage\Table\Models\Entity;
use MicrosoftAzure\Storage\Table\Models\EdmType;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

//Create new entity.
$entity = new Entity();

// PartitionKey and RowKey are required.
$entity->setPartitionKey("tasksSeattle");
$entity->setRowKey("1");

// If entity exists, existing properties are updated with new values and
// new properties are added. Missing properties are unchanged.
$entity->addProperty("Description", null, "Take out the trash.");
$entity->addProperty("DueDate", EdmType::DATETIME, new DateTime()); // Modified the DueDate field.
$entity->addProperty("Location", EdmType::STRING, "Home");
$entity->addProperty("Status", EdmType::STRING, "Complete"); // Added Status field.

try {
    // Calling insertOrReplaceEntity, instead of insertOrMergeEntity as shown,
    // would simply replace the entity with PartitionKey "tasksSeattle" and RowKey "1".
    $tableRestProxy->insertOrMergeEntity("mytable", $entity);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Retrieve a single entity

The **TableRestProxy->getEntity** method allows you to retrieve a single entity by querying for its `PartitionKey` and `RowKey`. In the example below, the partition key `tasksSeattle` and row key `1` are passed to the **getEntity** method.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

try {
    $result = $tableRestProxy->getEntity("mytable", "tasksSeattle", 1);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

$entity = $result->getEntity();

echo $entity->getPartitionKey().".". $entity->getRowKey();

```

Retrieve all entities in a partition

Entity queries are constructed using filters (for more information, see [Querying Tables and Entities](#)). To retrieve all entities in partition, use the filter "PartitionKey eq *partition_name*". The following example shows how to retrieve all entities in the `tasksSeattle` partition by passing a filter to the **queryEntities** method.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

$filter = "PartitionKey eq 'tasksSeattle'";

try {
    $result = $tableRestProxy->queryEntities("mytable", $filter);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

$entities = $result->getEntities();

foreach($entities as $entity){
    echo $entity->getPartitionKey().".". $entity->getRowKey()."<br />";
}

```

Retrieve a subset of entities in a partition

The same pattern used in the previous example can be used to retrieve any subset of entities in a partition. The subset of entities you retrieve are determined by the filter you use (for more information, see [Querying Tables](#)

and Entities). The following example shows how to use a filter to retrieve all entities with a specific `Location` and a `DueDate` less than a specified date.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

$filter = "Location eq 'Office' and DueDate lt '2012-11-5''";

try {
    $result = $tableRestProxy->queryEntities("mytable", $filter);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

$entities = $result->getEntities();

foreach($entities as $entity){
    echo $entity->getPartitionKey().".". $entity->getRowKey()."<br />";
}
```

Retrieve a subset of entity properties

A query can retrieve a subset of entity properties. This technique, called *projection*, reduces bandwidth and can improve query performance, especially for large entities. To specify a property to be retrieved, pass the name of the property to the **Query->addSelectField** method. You can call this method multiple times to add more properties. After executing **TableRestProxy->queryEntities**, the returned entities will only have the selected properties. (If you want to return a subset of Table entities, use a filter as shown in the queries above.)

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;
use MicrosoftAzure\Storage\Table\Models\QueryEntitiesOptions;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

$options = new QueryEntitiesOptions();
$options->addSelectField("Description");

try {
    $result = $tableRestProxy->queryEntities("mytable", $options);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

// All entities in the table are returned, regardless of whether
// they have the Description field.
// To limit the results returned, use a filter.
$entities = $result->getEntities();

foreach($entities as $entity){
    $description = $entity->getProperty("Description")->getValue();
    echo $description."<br />";
}

```

Update an entity

An existing entity can be updated by using the **Entity->setProperty** and **Entity->addProperty** methods on the entity, and then calling **TableRestProxy->updateEntity**. The following example retrieves an entity, modifies one property, removes another property, and adds a new property. Note that you can remove a property by setting its value to **null**.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;
use MicrosoftAzure\Storage\Table\Models\Entity;
use MicrosoftAzure\Storage\Table\Models\EdmType;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

$result = $tableRestProxy->getEntity("mytable", "tasksSeattle", 1);

$entity = $result->getEntity();

$entity->setPropertyValue("DueDate", new DateTime()); //Modified DueDate.

$entity->setPropertyValue("Location", null); //Removed Location.

$entity->addProperty("Status", EdmType::STRING, "In progress"); //Added Status.

try {
    $tableRestProxy->updateEntity("mytable", $entity);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Delete an entity

To delete an entity, pass the table name, and the entity's `PartitionKey` and `RowKey` to the **TableRestProxy->deleteEntity** method.

```

require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

try {
    // Delete entity.
    $tableRestProxy->deleteEntity("mytable", "tasksSeattle", "2");
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

```

Note that for concurrency checks, you can set the Etag for an entity to be deleted by using the **DeleteEntityOptions->setEtag** method and passing the **DeleteEntityOptions** object to **deleteEntity** as a fourth parameter.

Batch table operations

The **TableRestProxy->batch** method allows you to execute multiple operations in a single request. The pattern here involves adding operations to **BatchRequest** object and then passing the **BatchRequest** object to the **TableRestProxy->batch** method. To add an operation to a **BatchRequest** object, you can call any of the following methods multiple times:

- **addInsertEntity** (adds an insertEntity operation)
- **addUpdateEntity** (adds an updateEntity operation)
- **addMergeEntity** (adds a mergeEntity operation)
- **addInsertOrReplaceEntity** (adds an insertOrReplaceEntity operation)
- **addInsertOrMergeEntity** (adds an insertOrMergeEntity operation)
- **addDeleteEntity** (adds a deleteEntity operation)

The following example shows how to execute **insertEntity** and **deleteEntity** operations in a single request:

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;
use MicrosoftAzure\Storage\Table\Models\Entity;
use MicrosoftAzure\Storage\Table\Models\EdmType;
use MicrosoftAzure\Storage\Table\Models\BatchOperations;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

// Create list of batch operation.
$operations = new BatchOperations();

$entity1 = new Entity();
$entity1->setPartitionKey("tasksSeattle");
$entity1->setRowKey("2");
$entity1->addProperty("Description", null, "Clean roof gutters.");
$entity1->addProperty("DueDate",
    EdmType::DATETIME,
    new DateTime("2012-11-05T08:15:00-08:00"));
$entity1->addProperty("Location", EdmType::STRING, "Home");

// Add operation to list of batch operations.
$operations->addInsertEntity("mytable", $entity1);

// Add operation to list of batch operations.
$operations->addDeleteEntity("mytable", "tasksSeattle", "1");

try {
    $tableRestProxy->batch($operations);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code." : ".$error_message."<br />";
}
```

For more information about batching Table operations, see [Performing Entity Group Transactions](#).

Delete a table

Finally, to delete a table, pass the table name to the **TableRestProxy->deleteTable** method.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use MicrosoftAzure\Storage\Common\ServiceException;

// Create table REST proxy.
$tableRestProxy = ServicesBuilder::getInstance()->createTableService($connectionString);

try {
    // Delete table.
    $tableRestProxy->deleteTable("mytable");
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179438.aspx
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

Next steps

Now that you've learned the basics of the Azure Table service, follow these links to learn about more complex storage tasks.

- Visit the [Azure Storage team blog](#)

For more information, see also the [PHP Developer Center](#).

How to use Azure Table Storage from Ruby

1/17/2017 • 7 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This guide shows you how to perform common scenarios using the Azure Table service. The samples are written using the Ruby API. The scenarios covered include **creating and deleting a table, inserting and querying entities in a table**.

What is the Table Service

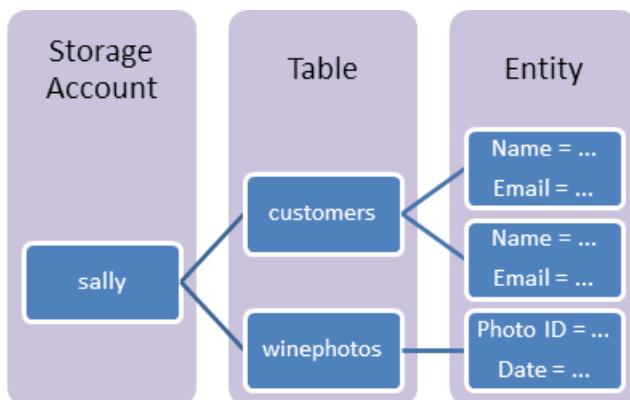
The Azure Table storage service stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of the Table service include:

- Storing TBs of structured data capable of serving web scale applications
- Storing datasets that don't require complex joins, foreign keys, or stored procedures and can be denormalized for fast access
- Quickly querying data using a clustered index
- Accessing data using the OData protocol and LINQ queries with WCF Data Service .NET Libraries

You can use the Table service to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table Service Concepts

The Table service contains the following components:



- **URL format:** Code addresses tables in an account using this address format:

`http://<storage account>.table.core.windows.net/<table>`

You can address Azure tables directly using this address with the OData protocol. For more information,

see [OData.org](#)

- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Table:** A table is a collection of entities. Tables don't enforce a schema on entities, which means a single table can contain entities that have different sets of properties. The number of tables that a storage account can contain is limited only by the storage account capacity limit.
- **Entity:** An entity is a set of properties, similar to a database row. An entity can be up to 1MB in size.
- **Properties:** A property is a name-value pair. Each entity can include up to 252 properties to store data. Each entity also has 3 system properties that specify a partition key, a row key, and a timestamp. Entities with the same partition key can be queried more quickly, and inserted/updated in atomic operations. An entity's row key is its unique identifier within a partition.

For details about naming tables and properties, see [Understanding the Table Service Data Model](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a Ruby application

For instructions how to create a Ruby application, see [Ruby on Rails Web application on an Azure VM](#).

Configure your application to access Storage

To use Azure Storage, you need to download and use the Ruby azure package which includes a set of convenience libraries that communicate with the Storage REST services.

Use RubyGems to obtain the package

1. Use a command-line interface such as **PowerShell** (Windows), **Terminal** (Mac), or **Bash** (Unix).
2. Type **gem install azure** in the command window to install the gem and dependencies.

Import the package

Use your favorite text editor, add the following to the top of the Ruby file where you intend to use Storage:

```
require "azure"
```

Set up an Azure Storage connection

The azure module will read the environment variables **AZURE_STORAGE_ACCOUNT** and **AZURE_STORAGE_ACCESS_KEY** for information required to connect to your Azure Storage account. If these environment variables are not set, you must specify the account information before using **Azure::TableService** with the following code:

```
Azure.config.storage_account_name = "<your azure storage account>"  
Azure.config.storage_access_key = "<your azure storage access key>"
```

To obtain these values from a classic or Resource Manager storage account in the Azure portal:

1. Log in to the [Azure portal](#).
2. Navigate to the storage account you want to use.
3. In the Settings blade on the right, click **Access Keys**.
4. In the Access keys blade that appears, you'll see the access key 1 and access key 2. You can use either of these.
5. Click the copy icon to copy the key to the clipboard.

To obtain these values from a classic storage account in the classic Azure portal:

1. Log in to the [Azure classic portal](#).
2. Navigate to the storage account you want to use.
3. Click **MANAGE ACCESS KEYS** at the bottom of the navigation pane.
4. In the pop-up dialog, you'll see the storage account name, primary access key and secondary access key. For access key, you can use either the primary one or the secondary one.
5. Click the copy icon to copy the key to the clipboard.

Create a table

The **Azure::TableService** object lets you work with tables and entities. To create a table, use the **create_table()** method. The following example creates a table or prints the error if there is any.

```
azure_table_service = Azure::TableService.new
begin
  azure_table_service.create_table("testtable")
rescue
  puts $!
end
```

Add an entity to a table

To add an entity, first create a hash object that defines your entity properties. Note that for every entity you must specify a **PartitionKey** and **RowKey**. These are the unique identifiers of your entities, and are values that can be queried much faster than your other properties. Azure Storage uses **PartitionKey** to automatically distribute the table's entities over many storage nodes. Entities with the same **PartitionKey** are stored on the same node. The **RowKey** is the unique ID of the entity within the partition it belongs to.

```
entity = { "content" => "test entity",
           :PartitionKey => "test-partition-key", :RowKey => "1" }
azure_table_service.insert_entity("testtable", entity)
```

Update an entity

There are multiple methods available to update an existing entity:

- **update_entity()**: Update an existing entity by replacing it.
- **merge_entity()**: Updates an existing entity by merging new property values into the existing entity.
- **insert_or_merge_entity()**: Updates an existing entity by replacing it. If no entity exists, a new one will be inserted.
- **insert_or_replace_entity()**: Updates an existing entity by merging new property values into the existing entity. If no entity exists, a new one will be inserted.

The following example demonstrates updating an entity using **update_entity()**:

```
entity = { "content" => "test entity with updated content",
           :PartitionKey => "test-partition-key", :RowKey => "1" }
azure_table_service.update_entity("testtable", entity)
```

With **update_entity()** and **merge_entity()**, if the entity that you are updating doesn't exist then the update operation will fail. Therefore if you wish to store an entity regardless of whether it already exists, you should instead use **insert_or_replace_entity()** or **insert_or_merge_entity()**.

Work with groups of entities

Sometimes it makes sense to submit multiple operations together in a batch to ensure atomic processing by the server. To accomplish that, you first create a **Batch** object and then use the **execute_batch()** method on **TableService**. The following example demonstrates submitting two entities with RowKey 2 and 3 in a batch. Notice that it only works for entities with the same PartitionKey.

```
azure_table_service = Azure::TableService.new
batch = Azure::Storage::Table::Batch.new("testtable",
                                         "test-partition-key") do
  insert "2", { "content" => "new content 2" }
  insert "3", { "content" => "new content 3" }
end
results = azure_table_service.execute_batch(batch)
```

Query for an entity

To query an entity in a table, use the **get_entity()** method, by passing the table name, **PartitionKey** and **RowKey**.

```
result = azure_table_service.get_entity("testtable", "test-partition-key",
                                         "1")
```

Query a set of entities

To query a set of entities in a table, create a query hash object and use the **query_entities()** method. The following example demonstrates getting all the entities with the same **PartitionKey**:

```
query = { :filter => "PartitionKey eq 'test-partition-key'" }
result, token = azure_table_service.query_entities("testtable", query)
```

NOTE

If the result set is too large for a single query to return, a continuation token will be returned which you can use to retrieve subsequent pages.

Query a subset of entity properties

A query to a table can retrieve just a few properties from an entity. This technique, called "projection", reduces bandwidth and can improve query performance, especially for large entities. Use the select clause and pass the names of the properties you would like to bring over to the client.

```
query = { :filter => "PartitionKey eq 'test-partition-key'",
          :select => ["content"] }
result, token = azure_table_service.query_entities("testtable", query)
```

Delete an entity

To delete an entity, use the **delete_entity()** method. You need to pass in the name of the table which contains the entity, the PartitionKey and RowKey of the entity.

```
azure_table_service.delete_entity("testtable", "test-partition-key", "1")
```

Delete a table

To delete a table, use the **delete_table()** method and pass in the name of the table you want to delete.

```
azure_table_service.delete_table("testtable")
```

Next steps

To learn about more complex storage tasks, follow these links:

- [Azure Storage Team Blog](#)
- [Azure SDK for Ruby](#) repository on GitHub

Get started with Azure File storage on Windows

1/17/2017 • 27 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

Azure File storage is a service that offers file shares in the cloud using the standard [Server Message Block \(SMB\) Protocol](#). Both SMB 2.1 and SMB 3.0 are supported. With Azure File storage, you can migrate legacy applications that rely on file shares to Azure quickly and without costly rewrites. Applications running in Azure virtual machines or cloud services or from on-premises clients can mount a file share in the cloud, just as a desktop application mounts a typical SMB share. Any number of application components can then mount and access the File storage share simultaneously.

Since a File storage share is a standard SMB file share, applications running in Azure can access data in the share via file system I/O APIs. Developers can therefore leverage their existing code and skills to migrate existing applications. IT Pros can use PowerShell cmdlets to create, mount, and manage File storage shares as part of the administration of Azure applications.

You can create Azure file shares using [Azure Portal](#), the Azure Storage PowerShell cmdlets, the Azure Storage client libraries, or the Azure Storage REST API. Additionally, because these file shares are SMB shares, you can access them via standard and familiar file system APIs.

For information on using File storage with Linux, see [How to use Azure File Storage with Linux](#).

For information on scalability and performance targets of File storage, see [Azure Storage Scalability and Performance Targets](#).

NOTE

We recommend that you use the latest version of the Azure Storage Client Library for .NET to complete this tutorial. The latest version of the library is 7.x, available for download on [Nuget](#). The source for the client library is available on [GitHub](#).

If you are using the storage emulator, note that version 7.x of the client library requires at least version 4.3 of the storage emulator.

What is Azure File storage?

File storage offers shared storage for applications using the standard SMB 2.1 or SMB 3.0 protocol. Microsoft Azure virtual machines and cloud services can share file data across application components via mounted shares, and on-premises applications can access file data in a share via the File storage API.

Applications running in Azure virtual machines or cloud services can mount a File storage share to access file data, just as a desktop application would mount a typical SMB share. Any number of Azure virtual machines or roles can mount and access the File storage share simultaneously.

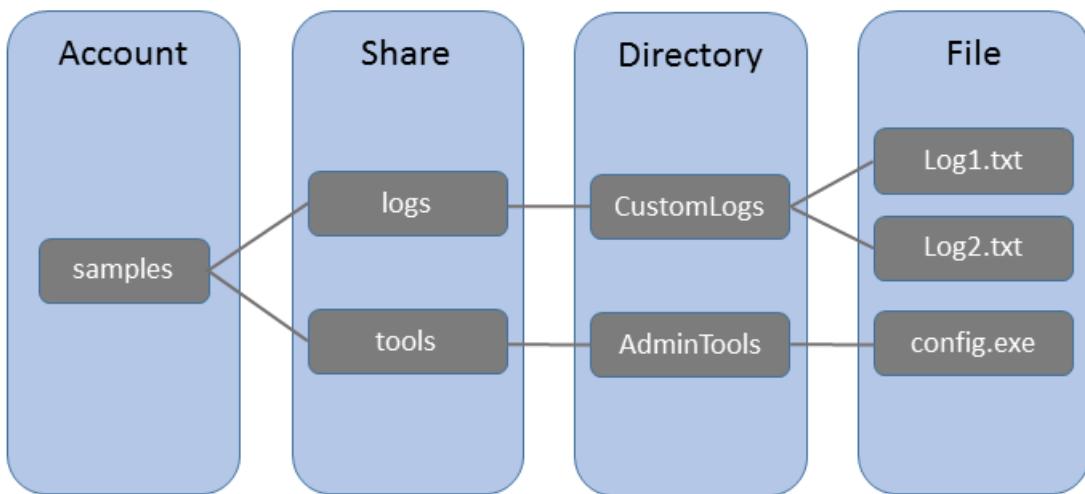
Since a File storage share is a standard file share in Azure using the SMB protocol, applications running in Azure can access data in the share via file I/O APIs. Developers can therefore leverage their existing code and skills to migrate existing applications. IT Pros can use PowerShell cmdlets to create, mount, and manage File storage shares as part of the administration of Azure applications. This guide will show examples of both.

Common uses of File storage include:

- Migrating on-premises applications that rely on file shares to run on Azure virtual machines or cloud services, without expensive rewrites
- Storing shared application settings, for example in configuration files
- Storing diagnostic data such as logs, metrics, and crash dumps in a shared location
- Storing tools and utilities needed for developing or administering Azure virtual machines or cloud services

File storage concepts

File storage contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Share:** A File storage share is an SMB file share in Azure. All directories and files must be created in a parent share. An account can contain an unlimited number of shares, and a share can store an unlimited number of files, up to the 5 TB total capacity of the file share.
- **Directory:** An optional hierarchy of directories.
- **File:** A file in the share. A file may be up to 1 TB in size.
- **URL format:** Files are addressable using the following URL format:
`https://<storage account>.file.core.windows.net/<share>/<directory/directories>/<file>`

The following example URL could be used to address one of the files in the diagram above:

```
http://samples.file.core.windows.net/logs/CustomLogs/Log1.txt
```

For details about how to name shares, directories, and files, see [Naming and Referencing Shares, Directories, Files, and Metadata](#).

Video: Using Azure File storage with Windows

Here's a video that demonstrates how to create and use Azure File shares on Windows.



About this tutorial

This getting started tutorial demonstrates the basics of using Microsoft Azure File storage. In this tutorial, we will:

- Use Azure Portal or PowerShell to create a new Azure File share, add a directory, upload a local file to the share, and list the files in the directory.
- Mount the file share, just as you would mount any SMB share.
- Use the Azure Storage Client Library for .NET to access the file share from an on-premises application. Create a console application and perform these actions with the file share:
 - Write the contents of a file in the share to the console window.
 - Set the quota (maximum size) for the file share.
 - Create a shared access signature for a file that uses a shared access policy defined on the share.
 - Copy a file to another file in the same storage account.
 - Copy a file to a blob in the same storage account.
- Use Azure Storage Metrics for troubleshooting

File storage is now supported for all storage accounts, so you can either use an existing storage account, or you can create a new storage account. See [How to create a storage account](#) for information on creating a new storage account.

Use the Azure Portal to manage a file share

The [Azure Portal](#) provides a user interface for customers to manage file shares. From the portal, you can:

- Create your file share
- Upload and download files to and from your file share
- Monitor the actual usage of each file share
- Adjust the share size quota
- Get the `net use` command to use to mount the file share from a Windows client

Create file share

1. Sign in to the Azure portal.
2. On the navigation menu, click **Storage accounts** or **Storage accounts (classic)**.

The screenshot shows the Microsoft Azure portal homepage. At the top, there are navigation icons for back, forward, and refresh, followed by the URL <https://portal.azure.com>. Below the header, the title "Microsoft Azure" is displayed with a dropdown arrow. A "New" button is located above a list of service links. The list includes: Resource groups, All resources, Recent, App Services, SQL databases, Virtual machines (classic), Virtual machines, Cloud services (classic), Subscriptions, Storage accounts, and Storage accounts (class...). The "Storage accounts" link is highlighted with a red box. At the bottom of the sidebar, there is a "Browse >" button.

3. Choose your storage account.

The screenshot shows the "Storage accounts" list in the Azure portal. The title "Storage accounts" is at the top, followed by a "Default Directory" section. Below are buttons for "Add", "Columns", and "Refresh". A "Filter items ..." input field is present. The main table has columns: NAME, RESOURCE GROUP, LOCATION, and SUBSCRIPTION. A single row is selected and highlighted with a red box, showing the details: NAME is "azureportaldemo", RESOURCE GROUP is "azurecon", LOCATION is "North Central US", and SUBSCRIPTION is "Visual Studio Ultimate with MSDN".

4. Choose "Files" service.

The screenshot shows the "Services" list in the Azure portal. It includes "Blobs" and "Files". The "Files" service is highlighted with a red box. Other services shown are TABLES and QUEUES.

5. Click "File shares" and follow the link to create your first file share.

File service
File service (azurereportaldemo)

Settings

Essentials ^

Storage account
[azurereportaldemo](#)

Status
Primary: Available, Secondary: Available

Location
North Central US, South Central US

Subscription name
[Visual Studio Ultimate with MSDN](#)

Subscription ID
6e0dff25-364c-4782-b836-9091c76c3b0d

File service endpoint
<https://azurereportaldemo.file.core.windows....>

All settings →

File shares

NAME	MODIFIED	QUOTA
You don't have any file shares. Click here to create one.		

- Fill in the file share name and the size of the file share (up to 5120 GB) to create your first file share. Once the file share has been created, you can mount it from any file system that supports SMB 2.1 or SMB 3.0.

New file share
File service (azurereportaldemo)

* Name
demoshare

Quota ⓘ
5120 GB

Upload and download files

- Choose one file share you already created.

The screenshot shows the 'File service' settings page for a file share named 'demoshare'. Key details include:

- Storage account:** azurereportaldemo
- File service endpoint:** https://azurereportaldemo.file.core.windows....
- Status:** Primary: Available, Secondary: Available
- Location:** North Central US, South Central US
- Subscription name:** Visual Studio Ultimate with MSDN
- Subscription ID:** 6e0dff25-364c-4782-b836-9091c76c3b0d

A red box highlights the 'demoshare' row in the 'File shares' table.

2. Click **Upload** to open the user interface for files uploading.

The screenshot shows the 'demoshare' file share interface. The 'Upload' button is highlighted with a red box.

3. Right click on one file and choose **Download** to download it into local.

The screenshot shows the 'demoshare' file share interface. A context menu is open over a file named 'event.txt'. The 'Download' option is highlighted with a red box.

Manage file share

1. Click **Quota** to change the size of the file share (up to 5120 GB).

NAME	TYPE	SIZE
event.txt	File	32.1 KB ...
sample.txt	File	32.1 KB ...

2. Click **Connect** to get the command line for mounting the file share from Windows.

NAME	TYPE	SIZE
event.txt	File	32.1 KB ...
sample.txt	File	32.1 KB ...

To connect to this file share, run this command from any Windows virtual machine on the same subscription and location:

```
net use [drive letter] \\azurereportaldemo.file.core.windows.net\demoshare /u:azurereportaldemo [storage account access key]
```

You can find the access key in the Settings > Access keys blade for your storage account.

TIP

To find the storage account access key for mounting, click **Settings** of your storage account, and then click **Access keys**.

azureportaldemo
Storage account

Settings Delete

Resource group: azurecon
Status: Primary: Available, Secondary: Available
Location: North Central US, South Central US
Subscription name: Visual Studio Ultimate with MSDN
Subscription ID: 6e0dff25-364c-4782-b836-9091c76c3b0d

All settings →

Settings
azureportaldemo

MANAGE

- Properties >
- Access keys > **Access keys**
- Account type >

MONITOR

- Alert rules >
- Audit logs >
- Diagnostics >

RESOURCE MANAGEMENT

- Users >
- Tags >

Use PowerShell to manage a file share

Alternatively, you can use Azure PowerShell to create and manage file shares.

Install the PowerShell cmdlets for Azure Storage

To prepare to use PowerShell, download and install the Azure PowerShell cmdlets. See [How to install and configure Azure PowerShell](#) for the install point and installation instructions.

NOTE

It's recommended that you download and install or upgrade to the latest Azure PowerShell module.

Open an Azure PowerShell window by clicking **Start** and typing **Windows PowerShell**. The PowerShell window loads the Azure Powershell module for you.

Create a context for your storage account and key

Now, create the storage account context. The context encapsulates the storage account name and account key. For instructions on copying your account key from the [Azure Portal](#), see [View and copy storage access keys](#).

Replace `storage-account-name` and `storage-account-key` with your storage account name and key in the following example.

```
# create a context for account and key
$ctx=New-AzureStorageContext storage-account-name storage-account-key
```

Create a new file share

Next, create the new share, named `logs`.

```
# create a new share
$s = New-AzureStorageShare logs -Context $ctx
```

You now have a file share in File storage. Next we'll add a directory and a file.

IMPORTANT

The name of your file share must be all lowercase. For complete details about naming file shares and files, see [Naming and Referencing Shares, Directories, Files, and Metadata](#).

Create a directory in the file share

Next, create a directory in the share. In the following example, the directory is named `CustomLogs`.

```
# create a directory in the share
New-AzureStorageDirectory -Share $s -Path CustomLogs
```

Upload a local file to the directory

Now upload a local file to the directory. The following example uploads a file from `C:\temp\Log1.txt`. Edit the file path so that it points to a valid file on your local machine.

```
# upload a local file to the new directory
Set-AzureStorageFileContent -Share $s -Source C:\temp\Log1.txt -Path CustomLogs
```

List the files in the directory

To see the file in the directory, you can list all of the directory's files. This command returns the files and subdirectories (if there are any) in the `CustomLogs` directory.

```
# list files in the new directory
Get-AzureStorageFile -Share $s -Path CustomLogs | Get-AzureStorageFile
```

`Get-AzureStorageFile` returns a list of files and directories for whatever directory object is passed in. "`Get-AzureStorageFile -Share $s`" returns a list of files and directories in the root directory. To get a list of files in a subdirectory, you have to pass the subdirectory to `Get-AzureStorageFile`. That's what this does -- the first part of the command up to the pipe returns a directory instance of the subdirectory `CustomLogs`. Then that is passed into `Get-AzureStorageFile`, which returns the files and directories in `CustomLogs`.

Copy files

Beginning with version 0.9.7 of Azure PowerShell, you can copy a file to another file, a file to a blob, or a blob to a file. Below we demonstrate how to perform these copy operations using PowerShell cmdlets.

```
# copy a file to the new directory
Start-AzureStorageFileCopy -SrcShareName srcshare -SrcFilePath srkdir/hello.txt -DestShareName destshare -
DestFilePath destdir/hellocopy.txt -Context $srcCtx -DestContext $destCtx

# copy a blob to a file directory
Start-AzureStorageFileCopy -SrcContainerName srcctn -SrcBlobName hello2.txt -DestShareName hello -DestFilePath
helldir/hello2copy.txt -DestContext $ctx -Context $ctx
```

Mount the file share

With support for SMB 3.0, File storage now supports encryption and persistent handles from SMB 3.0 clients. Support for encryption means that SMB 3.0 clients can mount a file share from anywhere, including from:

- An Azure virtual machine in the same region (also supported by SMB 2.1)
- An Azure virtual machine in a different region (SMB 3.0 only)
- An on-premises client application (SMB 3.0 only)

When a client accesses File storage, the SMB version used depends on the SMB version supported by the operating system. The table below provides a summary of support for Windows clients. Please refer to this blog for more details on [SMB versions](#).

WINDOWS CLIENT	SMB VERSION SUPPORTED
Windows 7	SMB 2.1
Windows Server 2008 R2	SMB 2.1
Windows 8	SMB 3.0
Windows Server 2012	SMB 3.0
Windows Server 2012 R2	SMB 3.0
Windows 10	SMB 3.0

Mount the file share from an Azure virtual machine running Windows

To demonstrate how to mount an Azure file share, we'll now create an Azure virtual machine running Windows, and remote into it to mount the share.

1. First, create a new Azure virtual machine by following the instructions in [Create a Windows virtual machine in the Azure Portal](#).
2. Next, remote into the virtual machine by following the instructions in [Log on to a Windows virtual machine using the Azure Portal](#).
3. Open a PowerShell window on the virtual machine.

Persist your storage account credentials for the virtual machine

Before mounting to the file share, first persist your storage account credentials on the virtual machine. This step allows Windows to automatically reconnect to the file share when the virtual machine reboots. To persist your account credentials, run the `cmdkey` command from the PowerShell window on the virtual machine. Replace `<storage-account-name>` with the name of your storage account, and `<storage-account-key>` with your storage account key.

```
cmdkey /add:<storage-account-name>.file.core.windows.net /user:<storage-account-name> /pass:<storage-account-key>
```

Windows will now reconnect to your file share when the virtual machine reboots. You can verify that the share has been reconnected by running the `net use` command from a PowerShell window.

Note that credentials are persisted only in the context in which `cmdkey` runs. If you are developing an application that runs as a service, you will need to persist your credentials in that context as well.

Mount the file share using the persisted credentials

Once you have a remote connection to the virtual machine, you can run the `net use` command to mount the file share, using the following syntax. Replace `<storage-account-name>` with the name of your storage account, and `<share-name>` with the name of your File storage share.

```
net use <drive-letter>: \\<storage-account-name>.file.core.windows.net\<share-name>  
example :  
net use z: \\samples.file.core.windows.net\logs
```

Since you persisted your storage account credentials in the previous step, you do not need to provide them with the `net use` command. If you have not already persisted your credentials, then include them as a parameter passed to the `net use` command, as shown in the following example.

```
net use <drive-letter>: \\<storage-account-name>.file.core.windows.net\<share-name> /u:<storage-account-name><storage-account-key>  
example :  
net use z: \\samples.file.core.windows.net\logs /u:samples <storage-account-key>
```

You can now work with the File Storage share from the virtual machine as you would with any other drive. You can issue standard file commands from the command prompt, or view the mounted share and its contents from File Explorer. You can also run code within the virtual machine that accesses the file share using standard Windows file I/O APIs, such as those provided by the [System.IO namespaces](#) in the .NET Framework.

You can also mount the file share from a role running in an Azure cloud service by remoting into the role.

Mount the file share from an on-premises client running Windows

To mount the file share from an on-premises client, you must first take these steps:

- Install a version of Windows which supports SMB 3.0. Windows will leverage SMB 3.0 encryption to securely transfer data between your on-premises client and the Azure file share in the cloud.
- Open Internet access for port 445 (TCP Outbound) in your local network, as is required by the SMB protocol.

NOTE

Some Internet service providers may block port 445, so you may need to check with your service provider.

Develop with File storage

To write code that calls File storage, you can use the storage client libraries for .NET and Java, or the Azure Storage REST API. The example in this section demonstrates how to work with a file share by using the [Azure Storage Client Library for .NET](#) from a simple console application running on the desktop.

Create the console application and obtain the assembly

To create a new console application in Visual Studio and install the NuGet package containing the Azure Storage

Client Library:

1. In Visual Studio, choose **File > New Project**, and then choose **Windows > Console Application** from the list of Visual C# templates.
2. Provide a name for the console application, and then click **OK**.
3. Once your project has been created, right-click the project in Solution Explorer and choose **Manage NuGet Packages**. Search online for "WindowsAzure.Storage" and click **Install** to install the Azure Storage Client Library for .NET package and dependencies.

The code examples in this article also use the [Microsoft Azure Configuration Manager Library](#) to retrieve the storage connection string from an app.config file in the console application. With Azure Configuration Manager, you can retrieve your connection string at runtime regardless of whether your application is running in Microsoft Azure or from a desktop, mobile, or web application.

To install the Azure Configuration Manager package, right-click the project in Solution Explorer and choose **Manage NuGet Packages**. Search online for "ConfigurationManager" and click **Install** to install the package.

Using Azure Configuration Manager is optional. You can also use an API such as the .NET Framework's [ConfigurationManager class](#).

Save your storage account credentials to the app.config file

Next, save your credentials in your project's app.config file. Edit the app.config file so that it appears similar to the following example, replacing `myaccount` with your storage account name, and `mykey` with your storage account key.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
    <appSettings>
        <add key="StorageConnectionString"
            value="DefaultEndpointsProtocol=https;AccountName=myaccount;AccountKey=StorageAccountKeyEndingIn==" />
    </appSettings>
</configuration>
```

NOTE

The latest version of the Azure storage emulator does not support File storage. Your connection string must target an Azure storage account in the cloud to work with File storage.

Add namespace declarations

Open the `program.cs` file from Solution Explorer, and add the following namespace declarations to the top of the file.

```
using Microsoft.Azure; // Namespace for Azure Configuration Manager
using Microsoft.WindowsAzure.Storage; // Namespace for Storage Client Library
using Microsoft.WindowsAzure.Storage.Blob; // Namespace for Blob storage
using Microsoft.WindowsAzure.Storage.File; // Namespace for File storage
```

The [Microsoft Azure Configuration Manager Library for .NET](#) provides a class for parsing a connection string from a configuration file. The [CloudConfigurationManager](#) class parses configuration settings regardless of whether the client application is running on the desktop, on a mobile device, in an Azure virtual machine, or in an Azure cloud service.

To reference the CloudConfigurationManager package, add the following `using` directive:

```
using Microsoft.Azure; //Namespace for CloudConfigurationManager
```

Here's an example that shows how to retrieve a connection string from a configuration file:

```
// Parse the connection string and return a reference to the storage account.  
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(  
    CloudConfigurationManager.GetSetting("StorageConnectionString"));
```

Using the Azure Configuration Manager is optional. You can also use an API like the .NET Framework's [ConfigurationManager](#) class.

Access the file share programmatically

Next, add the following code to the `Main()` method (after the code shown above) to retrieve the connection string. This code gets a reference to the file we created earlier and outputs its contents to the console window.

```
// Create a CloudFileClient object for credentialized access to File storage.  
CloudFileClient fileClient = storageAccount.CreateCloudFileClient();  
  
// Get a reference to the file share we created previously.  
CloudFileShare share = fileClient.GetShareReference("logs");  
  
// Ensure that the share exists.  
if (share.Exists())  
{  
    // Get a reference to the root directory for the share.  
    CloudFileDirectory rootDir = share.GetRootDirectoryReference();  
  
    // Get a reference to the directory we created previously.  
    CloudFileDirectory sampleDir = rootDir.GetDirectoryReference("CustomLogs");  
  
    // Ensure that the directory exists.  
    if (sampleDir.Exists())  
    {  
        // Get a reference to the file we created previously.  
        CloudFile file = sampleDir.GetFileReference("Log1.txt");  
  
        // Ensure that the file exists.  
        if (file.Exists())  
        {  
            // Write the contents of the file to the console window.  
            Console.WriteLine(file.DownloadTextAsync().Result);  
        }  
    }  
}
```

Run the console application to see the output.

Set the maximum size for a file share

Beginning with version 5.x of the Azure Storage Client Library, you can set set the quota (or maximum size) for a file share, in gigabytes. You can also check to see how much data is currently stored on the share.

By setting the quota for a share, you can limit the total size of the files stored on the share. If the total size of files on the share exceeds the quota set on the share, then clients will be unable to increase the size of existing files or create new files, unless those files are empty.

The example below shows how to check the current usage for a share and how to set the quota for the share.

```
// Parse the connection string for the storage account.  
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(  
    Microsoft.Azure.CloudConfigurationManager.GetSetting("StorageConnectionString"));  
  
// Create a CloudFileClient object for credentialled access to File storage.  
CloudFileClient fileClient = storageAccount.CreateCloudFileClient();  
  
// Get a reference to the file share we created previously.  
CloudFileShare share = fileClient.GetShareReference("logs");  
  
// Ensure that the share exists.  
if (share.Exists())  
{  
    // Check current usage stats for the share.  
    // Note that the ShareStats object is part of the protocol layer for the File service.  
    Microsoft.WindowsAzure.Storage.File.Protocol.ShareStats stats = share.GetStats();  
    Console.WriteLine("Current share usage: {0} GB", stats.Usage.ToString());  
  
    // Specify the maximum size of the share, in GB.  
    // This line sets the quota to be 10 GB greater than the current usage of the share.  
    share.Properties.Quota = 10 + stats.Usage;  
    share.SetProperties();  
  
    // Now check the quota for the share. Call FetchAttributes() to populate the share's properties.  
    share.FetchAttributes();  
    Console.WriteLine("Current share quota: {0} GB", share.Properties.Quota);  
}
```

Generate a shared access signature for a file or file share

Beginning with version 5.x of the Azure Storage Client Library, you can generate a shared access signature (SAS) for a file share or for an individual file. You can also create a shared access policy on a file share to manage shared access signatures. Creating a shared access policy is recommended, as it provides a means of revoking the SAS if it should be compromised.

The following example creates a shared access policy on a share, and then uses that policy to provide the constraints for a SAS on a file in the share.

```

// Parse the connection string for the storage account.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    Microsoft.Azure.CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create a CloudFileClient object for credentialled access to File storage.
CloudFileClient fileClient = storageAccount.CreateCloudFileClient();

// Get a reference to the file share we created previously.
CloudFileShare share = fileClient.GetShareReference("logs");

// Ensure that the share exists.
if (share.Exists())
{
    string policyName = "sampleSharePolicy" + DateTime.UtcNow.Ticks;

    // Create a new shared access policy and define its constraints.
    SharedAccessFilePolicy sharedPolicy = new SharedAccessFilePolicy()
    {
        SharedAccessExpiryTime = DateTime.UtcNow.AddHours(24),
        Permissions = SharedAccessFilePermissions.Read | SharedAccessFilePermissions.Write
    };

    // Get existing permissions for the share.
    FileSharePermissions permissions = share.GetPermissions();

    // Add the shared access policy to the share's policies. Note that each policy must have a unique name.
    permissions.SharedAccessPolicies.Add(policyName, sharedPolicy);
    share.SetPermissions(permissions);

    // Generate a SAS for a file in the share and associate this access policy with it.
    CloudFileDirectory rootDir = share.GetRootDirectoryReference();
    CloudFileDirectory sampleDir = rootDir.GetDirectoryReference("CustomLogs");
    CloudFile file = sampleDir.GetFileReference("Log1.txt");
    string sasToken = file.GetSharedAccessSignature(null, policyName);
    Uri fileSasUri = new Uri(file.StorageUri.PrimaryUri.ToString() + sasToken);

    // Create a new CloudFile object from the SAS, and write some text to the file.
    CloudFile fileSas = new CloudFile(fileSasUri);
    fileSas.UploadText("This write operation is authenticated via SAS.");
    Console.WriteLine(fileSas.DownloadText());
}

```

For more information about creating and using shared access signatures, see [Using Shared Access Signatures \(SAS\)](#) and [Create and use a SAS with Blob storage](#).

Copy files

Beginning with version 5.x of the Azure Storage Client Library, you can copy a file to another file, a file to a blob, or a blob to a file. In the next sections, we demonstrate how to perform these copy operations programmatically.

You can also use AzCopy to copy one file to another or to copy a blob to a file or vice versa. See [Transfer data with the AzCopy Command-Line Utility](#).

NOTE

If you are copying a blob to a file, or a file to a blob, you must use a shared access signature (SAS) to authenticate the source object, even if you are copying within the same storage account.

Copy a file to another file

The following example copies a file to another file in the same share. Because this copy operation copies between files in the same storage account, you can use Shared Key authentication to perform the copy.

```

// Parse the connection string for the storage account.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    Microsoft.Azure.CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create a CloudFileClient object for credentialled access to File storage.
CloudFileClient fileClient = storageAccount.CreateCloudFileClient();

// Get a reference to the file share we created previously.
CloudFileShare share = fileClient.GetShareReference("logs");

// Ensure that the share exists.
if (share.Exists())
{
    // Get a reference to the root directory for the share.
    CloudFileDirectory rootDir = share.GetRootDirectoryReference();

    // Get a reference to the directory we created previously.
    CloudFileDirectory sampleDir = rootDir.GetDirectoryReference("CustomLogs");

    // Ensure that the directory exists.
    if (sampleDir.Exists())
    {
        // Get a reference to the file we created previously.
        CloudFile sourceFile = sampleDir.GetFileReference("Log1.txt");

        // Ensure that the source file exists.
        if (sourceFile.Exists())
        {
            // Get a reference to the destination file.
            CloudFile destFile = sampleDir.GetFileReference("Log1Copy.txt");

            // Start the copy operation.
            destFile.StartCopy(sourceFile);

            // Write the contents of the destination file to the console window.
            Console.WriteLine(destFile.DownloadText());
        }
    }
}

```

Copy a file to a blob

The following example creates a file and copies it to a blob within the same storage account. The example creates a SAS for the source file, which the service uses to authenticate access to the source file during the copy operation.

```

// Parse the connection string for the storage account.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    Microsoft.Azure.CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Create a CloudFileClient object for credentialled access to File storage.
CloudFileClient fileClient = storageAccount.CreateCloudFileClient();

// Create a new file share, if it does not already exist.
CloudFileShare share = fileClient.GetShareReference("sample-share");
share.CreateIfNotExists();

// Create a new file in the root directory.
CloudFile sourceFile = share.GetRootDirectoryReference().GetFileReference("sample-file.txt");
sourceFile.UploadText("A sample file in the root directory.");

// Get a reference to the blob to which the file will be copied.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
CloudBlobContainer container = blobClient.GetContainerReference("sample-container");
container.CreateIfNotExists();
CloudBlockBlob destBlob = container.GetBlockBlobReference("sample-blob.txt");

// Create a SAS for the file that's valid for 24 hours.
// Note that when you are copying a file to a blob, or a blob to a file, you must use a SAS
// to authenticate access to the source object, even if you are copying within the same
// storage account.
string fileSas = sourceFile.GetSharedAccessSignature(new SharedAccessFilePolicy()
{
    // Only read permissions are required for the source file.
    Permissions = SharedAccessFilePermissions.Read,
    SharedAccessExpiryTime = DateTime.UtcNow.AddHours(24)
});

// Construct the URI to the source file, including the SAS token.
Uri fileSasUri = new Uri(sourceFile.StorageUri.PrimaryUri.ToString() + fileSas);

// Copy the file to the blob.
destBlob.StartCopy(fileSasUri);

// Write the contents of the file to the console window.
Console.WriteLine("Source file contents: {0}", sourceFile.DownloadText());
Console.WriteLine("Destination blob contents: {0}", destBlob.DownloadText());

```

You can copy a blob to a file in the same way. If the source object is a blob, then create a SAS to authenticate access to that blob during the copy operation.

Troubleshooting File storage using metrics

Azure Storage Analytics now supports metrics for File storage. With metrics data, you can trace requests and diagnose issues.

You can enable metrics for File storage from the [Azure Portal](#). You can also enable metrics programmatically by calling the Set File Service Properties operation via the REST API, or one of its analogues in the Storage Client Library.

The following code example shows how to use the Storage Client Library for .NET to enable metrics for File storage.

First, add the following `using` statements to your program.cs file, in addition to those you added above:

```

using Microsoft.WindowsAzure.Storage.File.Protocol;
using Microsoft.WindowsAzure.Storage.Shared.Protocol;

```

Note that while Blob, Table, and Queue storage use the shared `ServiceProperties` type in the

`Microsoft.WindowsAzure.Storage.Shared.Protocol` namespace, File storage uses its own type, the `FileServiceProperties` type in the `Microsoft.WindowsAzure.Storage.File.Protocol` namespace. Both namespaces must be referenced from your code, however, for the following code to compile.

```
// Parse your storage connection string from your application's configuration file.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    Microsoft.Azure.CloudConfigurationManager.GetSetting("StorageConnectionString"));
// Create the File service client.
CloudFileClient fileClient = storageAccount.CreateCloudFileClient();

// Set metrics properties for File service.
// Note that the File service currently uses its own service properties type,
// available in the Microsoft.WindowsAzure.Storage.File.Protocol namespace.
fileClient.SetServiceProperties(new FileServiceProperties())
{
    // Set hour metrics
    HourMetrics = new MetricsProperties()
    {
        MetricsLevel = MetricsLevel.ServiceAndApi,
        RetentionDays = 14,
        Version = "1.0"
    },
    // Set minute metrics
    MinuteMetrics = new MetricsProperties()
    {
        MetricsLevel = MetricsLevel.ServiceAndApi,
        RetentionDays = 7,
        Version = "1.0"
    }
});

// Read the metrics properties we just set.
FileServiceProperties serviceProperties = fileClient.GetServiceProperties();
Console.WriteLine("Hour metrics:");
Console.WriteLine(serviceProperties.HourMetrics.MetricsLevel);
Console.WriteLine(serviceProperties.HourMetrics.RetentionDays);
Console.WriteLine(serviceProperties.HourMetrics.Version);
Console.WriteLine();
Console.WriteLine("Minute metrics:");
Console.WriteLine(serviceProperties.MinuteMetrics.MetricsLevel);
Console.WriteLine(serviceProperties.MinuteMetrics.RetentionDays);
Console.WriteLine(serviceProperties.MinuteMetrics.Version);
```

Also, you can refer to [Azure Files Troubleshooting Article](#) for end-to-end troubleshooting guidance.

File storage FAQ

1. Is Active Directory-based authentication supported by File storage?

We currently do not support AD-based authentication or ACLs, but do have it in our list of feature requests. For now, the Azure Storage account keys are used to provide authentication to the file share. We do offer a workaround using shared access signatures (SAS) via the REST API or the client libraries. Using SAS, you can generate tokens with specific permissions that are valid over a specified time interval. For example, you can generate a token with read-only access to a given file. Anyone who possesses this token while it is valid has read-only access to that file.

SAS is only supported via the REST API or client libraries. When you mount the file share via the SMB protocol, you can't use a SAS to delegate access to its contents.

2. Are Azure File shares visible publicly over the Internet, or are they only reachable from Azure?

As long as port 445 (TCP Outbound) is open and your client supports the SMB 3.0 protocol (e.g., Windows 8 or Windows Server 2012), your file share is available via the Internet.

3. Does the network traffic between an Azure virtual machine and a file share count as external bandwidth that is charged to the subscription?

If the file share and virtual machine are in different regions, the traffic between them will be charged as external bandwidth.

4. If network traffic is between a virtual machine and a file share in the same region, is it free?

Yes. It is free if the traffic is in the same region.

5. Does connecting from on-premises virtual machines to Azure File Storage depend on Azure ExpressRoute?

No. If you don't have ExpressRoute, you can still access the file share from on-premises as long as you have port 445 (TCP Outbound) open for Internet access. However, you can use ExpressRoute with File storage if you like.

6. Is a "File Share Witness" for a failover cluster one of the use cases for Azure File Storage?

This is not supported currently.

7. File storage is replicated only via LRS or GRS right now, right?

We plan to support RA-GRS but there is no timeline to share yet.

8. When can I use existing storage accounts for Azure File Storage?

Azure File Storage is now enabled for all storage accounts.

9. Will a Rename operation also be added to the REST API?

Rename is not yet supported in our REST API.

10. Can you have nested shares, in other words, a share under a share?

No. The file share is the virtual driver that you can mount, so nested shares are not supported.

11. Is it possible to specify read-only or write-only permissions on folders within the share?

You don't have this level of control over permissions if you mount the file share via SMB. However, you can achieve this by creating a shared access signature (SAS) via the REST API or client libraries.

12. My performance was slow when trying to unzip files into in File storage. What should I do?

To transfer large numbers of files into File storage, we recommend that you use AzCopy, Azure Powershell (Windows), or the Azure CLI (Linux/Unix), as these tools have been optimized for network transfer.

13. Patch released to fix slow-performance issue with Azure Files

The Windows team recently released a patch to fix a slow performance issue when the customer accesses Azure Files Storage from Windows 8.1 or Windows Server 2012 R2. For more information, please check out the associated KB article, [Slow performance when you access Azure Files Storage from Windows 8.1 or Server 2012 R2](#).

14. Using Azure File Storage with IBM MQ

IBM has released a document to guide IBM MQ customers when configuring Azure File Storage with their service. For more information, please check out [How to setup IBM MQ Multi instance queue manager with Microsoft Azure File Service](#).

15. How do I troubleshoot Azure File Storage errors?

You can refer to [Azure Files Troubleshooting Article](#) for end-to-end troubleshooting guidance.

Next steps

See these links for more information about Azure File storage.

Conceptual articles and videos

- [Azure Files Storage: a frictionless cloud SMB file system for Windows and Linux](#)
- [How to use Azure File Storage with Linux](#)

Tooling support for File storage

- [Using Azure PowerShell with Azure Storage](#)
- [How to use AzCopy with Microsoft Azure Storage](#)
- [Using the Azure CLI with Azure Storage](#)

Reference

- [Storage Client Library for .NET reference](#)
- [File Service REST API reference](#)

Blog posts

- [Azure File storage is now generally available](#)
- [Inside Azure File Storage](#)
- [Introducing Microsoft Azure File Service](#)
- [Persisting connections to Microsoft Azure Files](#)

How to use Azure File Storage with Linux

1/17/2017 • 5 min to read • [Edit on GitHub](#)

Overview

Azure File storage offers file shares in the cloud using the standard SMB protocol. With Azure Files, you can migrate enterprise applications that rely on file servers to Azure. Applications running in Azure can easily mount file shares from Azure virtual machines running Linux. And with the latest release of File storage, you can also mount a file share from an on-premises application that supports SMB 3.0.

You can create Azure file shares using the [Azure Portal](#), the Azure Storage PowerShell cmdlets, the Azure Storage client libraries, or the Azure Storage REST API. Additionally, because file shares are SMB shares, you can access them via standard file system APIs.

File storage is built on the same technology as Blob, Table, and Queue storage, so File storage offers the availability, durability, scalability, and geo-redundancy that is built into the Azure storage platform. For details about File storage performance targets and limits, see [Azure Storage Scalability and Performance Targets](#).

File storage is now generally available and supports both SMB 2.1 and SMB 3.0. For additional details on File storage, see the [File service REST API](#).

NOTE

The Linux SMB client doesn't yet support encryption, so mounting a file share from Linux still requires that the client be in the same Azure region as the file share. However, encryption support for Linux is on the roadmap of Linux developers responsible for SMB functionality. Linux distributions that support encryption in the future will be able to mount an Azure File share from anywhere as well.

Video: Using Azure File storage with Linux

Here's a video that demonstrates how to create and use Azure File shares on Linux.



Choose a Linux distribution to use

When creating a Linux virtual machine in Azure, you can specify a Linux image which supports SMB 2.1 or higher from the Azure image gallery. Below is a list of recommended Linux images:

- Ubuntu Server 14.04+
- RHEL 7+
- CentOS 7+
- Debian 8
- openSUSE 13.2+
- SUSE Linux Enterprise Server 12

- SUSE Linux Enterprise Server 12 (Premium Image)

Mount the file share

To mount the file share from a virtual machine running Linux, you may need to install an SMB/CIFS client if the distribution you are using doesn't have a built-in client. This is the command from Ubuntu to install one choice cifs-utils:

```
sudo apt-get install cifs-utils
```

Next, you need to make a mount point (mkdir mymountpoint), and then issue the mount command that is similar to this:

```
sudo mount -t cifs //myaccountname.file.core.windows.net/mysharename ./mymountpoint -o  
vers=3.0,username=myaccountname,password=StorageAccountKeyEndingIn==,dir_mode=0777,file_mode=0777
```

You can also add settings in your /etc/fstab to mount the share.

Note that 0777 here represent a directory/file permission code that gives execution/read/write permissions to all users. You can replace it with other file permission code following Linux file permission document.

Also to keep a file share mounted after reboot, you can add a setting like below in your /etc/fstab:

```
//myaccountname.file.core.windows.net/mysharename /mymountpoint cifs  
vers=3.0,username=myaccountname,password=StorageAccountKeyEndingIn==,dir_mode=0777,file_mode=0777
```

For example, if you created a Azure VM using Linux image Ubuntu Server 15.04 (which is available from the Azure image gallery), you can mount the file as below:

```
azureuser@azureconubuntu:~$ sudo apt-get install cifs-utils  
azureuser@azureconubuntu:~$ sudo mkdir /mnt/mountpoint  
azureuser@azureconubuntu:~$ sudo mount -t cifs //myaccountname.file.core.windows.net/mysharename /mnt/mountpoint  
-o vers=3.0,user=myaccountname,password=StorageAccountKeyEndingIn==,dir_mode=0777,file_mode=0777  
azureuser@azureconubuntu:~$ df -h /mnt/mountpoint  
Filesystem Size Used Avail Use% Mounted on  
//myaccountname.file.core.windows.net/mysharename 5.0T 64K 5.0T 1% /mnt/mountpoint
```

If you use CentOS 7.1, you can mount the file as below:

```
[azureuser@AzureconCent ~]$ sudo yum install samba-client samba-common cifs-utils  
[azureuser@AzureconCent ~]$ sudo mount -t cifs //myaccountname.file.core.windows.net/mysharename /mnt/mountpoint  
-o vers=3.0,user=myaccountname,password=StorageAccountKeyEndingIn==,dir_mode=0777,file_mode=0777  
[azureuser@AzureconCent ~]$ df -h /mnt/mountpoint  
Filesystem Size Used Avail Use% Mounted on  
//myaccountname.file.core.windows.net/mysharename 5.0T 64K 5.0T 1% /mnt/mountpoint
```

If you use Open SUSE 13.2, you can mount the file as below:

```
azureuser@AzureconSuse:~> sudo zypper install samba*  
azureuser@AzureconSuse:~> sudo mkdir /mnt/mountpoint  
azureuser@AzureconSuse:~> sudo mount -t cifs //myaccountname.file.core.windows.net/mysharename /mnt/mountpoint -  
o vers=3.0,user=myaccountname,password=StorageAccountKeyEndingIn==,dir_mode=0777,file_mode=0777  
azureuser@AzureconSuse:~> df -h /mnt/mountpoint  
Filesystem Size Used Avail Use% Mounted on  
//myaccountname.file.core.windows.net/mysharename 5.0T 64K 5.0T 1% /mnt/mountpoint
```

If you use RHEL 7.3, you can mount the file as below:

```
azureuser@AzureconRedhat:~> sudo yum install cifs-utils
azureuser@AzureconRedhat:~> sudo mkdir /mnt/mountpoint
azureuser@AzureconRedhat:~> sudo mount -t cifs //myaccountname.file.core.windows.net/mysharename /mnt/mountpoint
-o vers=3.0,user=myaccountname,password=StorageAccountKeyEndingIn==,dir_mode=0777,file_mode=0777
azureuser@AzureconRedhat:~> df -h /mnt/mountpoint
Filesystem Size Used Avail Use% Mounted on
//myaccountname.file.core.windows.net/mysharename 5.0T 64K 5.0T 1% /mnt/mountpoint
```

Manage the file share

The [Azure Portal](#) provides a user interface for managing Azure File Storage. You can perform the following actions from your web browser:

- Upload and download files to and from your file share.
- Monitor the actual usage of each file share.
- Adjust the file share size quota.
- Copy the `net use` command to use to mount your file share from a Windows client.

You can also use the Azure Cross-Platform Command-Line Interface (Azure CLI) from Linux to manage the file share. Azure CLI provides a set of open source, cross-platform commands for you to work with Azure Storage, including File storage. It provides much of the same functionality found in the Azure Portal as well as rich data access functionality. For examples, see [Using the Azure CLI with Azure Storage](#).

Develop with File storage

As a developer, you can build an application with File storage by using the [Azure Storage Client Library for Java](#). For code examples, see [How to use File storage from Java](#).

You can also use the [Azure Storage Client Library for Node.js](#) to develop against File storage.

Feedback and more information

Linux users, we want to hear from you!

The Azure File storage for Linux users' group provides a forum for you to share feedback as you evaluate and adopt File storage on Linux. Email [Azure File Storage Linux Users](#) to join the users' group.

Next steps

See these links for more information about Azure File storage.

Conceptual articles and videos

- [Azure Files Storage: a frictionless cloud SMB file system for Windows and Linux](#)
- [Get started with Azure File storage on Windows](#)

Tooling support for File storage

- [Transfer data with the AzCopy Command-Line Utility](#)
- [Create and manage file shares using the Azure CLI](#)

Reference

- [File Service REST API reference](#)
- [Azure Files Troubleshooting Article](#)

Blog posts

- [Azure File storage is now generally available](#)

- Inside Azure File Storage
- Introducing Microsoft Azure File Service
- Persisting connections to Microsoft Azure Files

How to use File Storage from Java

1/17/2017 • 8 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

In this guide you'll learn how to perform basic operations on the Microsoft Azure File storage service. Through samples written in Java you'll learn how to create shares and directories, upload, list, and delete files. If you are new to Microsoft Azure's File Storage service, going through the concepts in the sections that follow will be very helpful in understanding the samples.

What is Azure File storage?

File storage offers shared storage for applications using the standard SMB 2.1 or SMB 3.0 protocol. Microsoft Azure virtual machines and cloud services can share file data across application components via mounted shares, and on-premises applications can access file data in a share via the File storage API.

Applications running in Azure virtual machines or cloud services can mount a File storage share to access file data, just as a desktop application would mount a typical SMB share. Any number of Azure virtual machines or roles can mount and access the File storage share simultaneously.

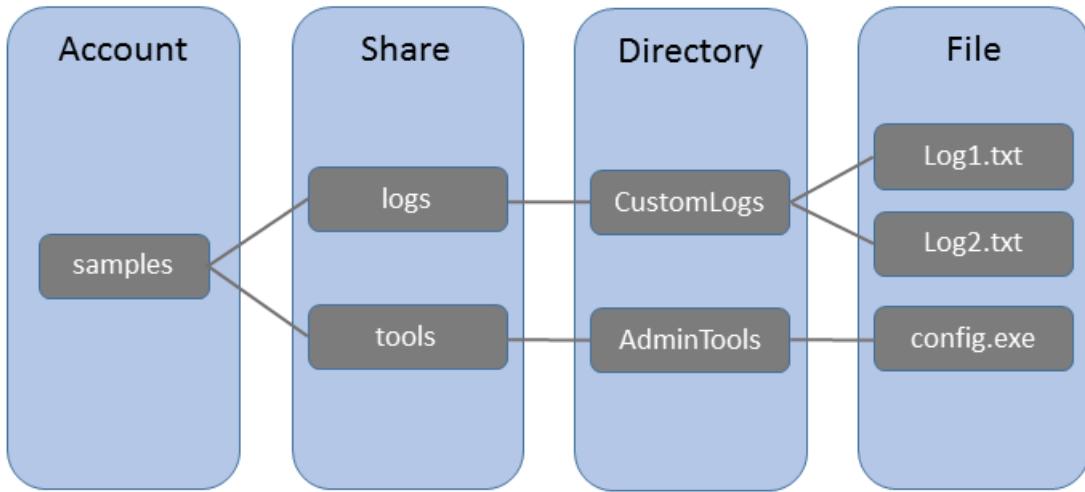
Since a File storage share is a standard file share in Azure using the SMB protocol, applications running in Azure can access data in the share via file I/O APIs. Developers can therefore leverage their existing code and skills to migrate existing applications. IT Pros can use PowerShell cmdlets to create, mount, and manage File storage shares as part of the administration of Azure applications. This guide will show examples of both.

Common uses of File storage include:

- Migrating on-premises applications that rely on file shares to run on Azure virtual machines or cloud services, without expensive rewrites
- Storing shared application settings, for example in configuration files
- Storing diagnostic data such as logs, metrics, and crash dumps in a shared location
- Storing tools and utilities needed for developing or administering Azure virtual machines or cloud services

File storage concepts

File storage contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Share:** A File storage share is an SMB file share in Azure. All directories and files must be created in a parent share. An account can contain an unlimited number of shares, and a share can store an unlimited number of files, up to the 5 TB total capacity of the file share.
- **Directory:** An optional hierarchy of directories.
- **File:** A file in the share. A file may be up to 1 TB in size.
- **URL format:** Files are addressable using the following URL format:
`https://<storage account>.file.core.windows.net/<share>/<directory/directories>/<file>`

The following example URL could be used to address one of the files in the diagram above:

```
http://samples.file.core.windows.net/logs/CustomLogs/Log1.txt
```

For details about how to name shares, directories, and files, see [Naming and Referencing Shares, Directories, Files, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a Java application

To build the samples, you will need the Java Development Kit (JDK) and the [Azure Storage SDK for Java][]. You should also have created an Azure storage account.

Setup your application to use File storage

To use the Azure storage APIs, add the following statement to the top of the Java file where you intend to access the storage service from.

```
// Include the following imports to use blob APIs.  
import com.microsoft.azure.storage.*;  
import com.microsoft.azure.storage.file.*;
```

Setup an Azure storage connection string

To use File storage, you need to connect to your Azure storage account. The first step would be to configure a connection string which we'll use to connect to your storage account. Let's define a static variable to do that.

```
// Configure the connection-string with your values  
public static final String storageConnectionString =  
    "DefaultEndpointsProtocol=http;" +  
    "AccountName=your_storage_account_name;" +  
    "AccountKey=your_storage_account_key";
```

NOTE

Replace `your_storage_account_name` and `your_storage_account_key` with the actual values for your storage account.

Connecting to an Azure storage account

To connect to your storage account, you need to use the **CloudStorageAccount** object, passing a connection string to its **parse** method.

```
// Use the CloudStorageAccount object to connect to your storage account  
try {  
    CloudStorageAccount storageAccount = CloudStorageAccount.parse(storageConnectionString);  
} catch (InvalidKeyException invalidKey) {  
    // Handle the exception  
}
```

CloudStorageAccount.parse throws an `InvalidKeyException` so you'll need to put it inside a try/catch block.

How to: Create a Share

All files and directories in File storage reside in a container called a **Share**. Your storage account can have as much shares as your account capacity allows. To obtain access to a share and its contents, you need to use a File storage client.

```
// Create the file storage client.  
CloudFileClient fileClient = storageAccount.createCloudFileClient();
```

Using the File storage client, you can then obtain a reference to a share.

```
// Get a reference to the file share  
CloudFileShare share = fileClient.getShareReference("sampleshare");
```

To actually create the share, use the **createIfNotExists** method of the **CloudFileShare** object.

```
if (share.createIfNotExists()) {  
    System.out.println("New share created");  
}
```

At this point, **share** holds a reference to a share named **sampleshare**.

How to: Upload a file

An Azure File Storage Share contains at the very least, a root directory where files can reside. In this section, you'll learn how to upload a file from local storage onto the root directory of a share.

The first step in uploading a file is to obtain a reference to the directory where it should reside. You do this by calling the **getRootDirectoryReference** method of the share object.

```
//Get a reference to the root directory for the share.  
CloudFileDirectory rootDir = share.getRootDirectoryReference();
```

Now that you have a reference to the root directory of the share, you can upload a file onto it using the following code.

```
// Define the path to a local file.  
final String filePath = "C:\\\\temp\\\\Readme.txt";  
  
CloudFile cloudFile = rootDir.getFileReference("Readme.txt");  
cloudFile.uploadFromFile(filePath);
```

How to: Create a Directory

You can also organize storage by putting files inside sub-directories instead of having all of them in the root directory. The Azure file storage service allows you to create as much directories as your account will allow. The code below will create a sub-directory named **sampledir** under the root directory.

```
//Get a reference to the root directory for the share.  
CloudFileDirectory rootDir = share.getRootDirectoryReference();  
  
//Get a reference to the sampledir directory  
CloudFileDirectory sampleDir = rootDir.getDirectoryReference("sampledir");  
  
if (sampleDir.createIfNotExists()) {  
    System.out.println("sampledir created");  
} else {  
    System.out.println("sampledir already exists");  
}
```

How to: List files and directories in a share

Obtaining a list of files and directories within a share is easily done by calling **listFilesAndDirectories** on a CloudFileDirectory reference. The method returns a list of ListFileItem objects which you can iterate on. As an example, the following code will list files and directories inside the root directory.

```
//Get a reference to the root directory for the share.  
CloudFileDirectory rootDir = share.getRootDirectoryReference();  
  
for ( ListFileItem fileItem : rootDir.listFilesAndDirectories() ) {  
    System.out.println(fileItem.getUri());  
}
```

How to: Download a file

One of the more frequent operations you will perform against file storage is to download files. In the following example, the code downloads SampleFile.txt and displays its contents.

```

//Get a reference to the root directory for the share.
CloudFileDirectory rootDir = share.getRootDirectoryReference();

//Get a reference to the directory that contains the file
CloudFileDirectory sampleDir = rootDir.getDirectoryReference("sampledir");

//Get a reference to the file you want to download
CloudFile file = sampleDir.getFileReference("SampleFile.txt");

//Write the contents of the file to the console.
System.out.println(file.downloadText());

```

How to: Delete a file

Another common file storage operation is file deletion. The following code deletes a file named SampleFile.txt stored inside a directory named **sampledir**.

```

// Get a reference to the root directory for the share.
CloudFileDirectory rootDir = share.getRootDirectoryReference();

// Get a reference to the directory where the file to be deleted is in
CloudFileDirectory containerDir = rootDir.getDirectoryReference("sampledir");

String filename = "SampleFile.txt"
CloudFile file;

file = containerDir.getFileReference(filename)
if ( file.deleteIfExists() ) {
    System.out.println(filename + " was deleted");
}

```

How to: Delete a directory

Deleting a directory is a fairly simple task, although it should be noted that you cannot delete a directory that still contains files or other directories.

```

// Get a reference to the root directory for the share.
CloudFileDirectory rootDir = share.getRootDirectoryReference();

// Get a reference to the directory you want to delete
CloudFileDirectory containerDir = rootDir.getDirectoryReference("sampledir");

// Delete the directory
if ( containerDir.deleteIfExists() ) {
    System.out.println("Directory deleted");
}

```

How to: Delete a Share

Deleting a share is done by calling the **deleteIfExists** method on a CloudFileShare object. Here's sample code that does that.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount = CloudStorageAccount.parse(storageConnectionString);

    // Create the file client.
    CloudFileClient fileClient = storageAccount.createCloudFileClient();

    // Get a reference to the file share
    CloudFileShare share = fileClient.getShareReference("sampleshare");

    if (share.deleteIfExists()) {
        System.out.println("sampleshare deleted");
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

Next steps

If you would like to learn more about other Azure storage APIs, follow these links.

- [Java Developer Center](#)
- [Azure Storage SDK for Java](#)
- [Azure Storage SDK for Android](#)
- [Azure Storage Client SDK Reference](#)
- [Azure Storage Services REST API](#)
- [Azure Storage Team Blog](#)
- [Transfer data with the AzCopy Command-Line Utility](#)

How to use File Storage from C++

1/17/2017 • 11 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

Azure File storage is a service that offers file shares in the cloud using the standard [Server Message Block \(SMB\) Protocol](#). Both SMB 2.1 and SMB 3.0 are supported. With Azure File storage, you can migrate legacy applications that rely on file shares to Azure quickly and without costly rewrites. Applications running in Azure virtual machines or cloud services or from on-premises clients can mount a file share in the cloud, just as a desktop application mounts a typical SMB share. Any number of application components can then mount and access the File storage share simultaneously.

Since a File storage share is a standard SMB file share, applications running in Azure can access data in the share via file system I/O APIs. Developers can therefore leverage their existing code and skills to migrate existing applications. IT Pros can use PowerShell cmdlets to create, mount, and manage File storage shares as part of the administration of Azure applications.

You can create Azure file shares using [Azure Portal](#), the Azure Storage PowerShell cmdlets, the Azure Storage client libraries, or the Azure Storage REST API. Additionally, because these file shares are SMB shares, you can access them via standard and familiar file system APIs.

About this tutorial

In this tutorial, you'll learn how to perform basic operations on the Microsoft Azure File storage service. Through samples written in C++, you'll learn how to create shares and directories, upload, list, and delete files. If you are new to Microsoft Azure's File Storage service, going through the concepts in the sections that follow will be very helpful in understanding the samples.

What is Azure File storage?

File storage offers shared storage for applications using the standard SMB 2.1 or SMB 3.0 protocol. Microsoft Azure virtual machines and cloud services can share file data across application components via mounted shares, and on-premises applications can access file data in a share via the File storage API.

Applications running in Azure virtual machines or cloud services can mount a File storage share to access file data, just as a desktop application would mount a typical SMB share. Any number of Azure virtual machines or roles can mount and access the File storage share simultaneously.

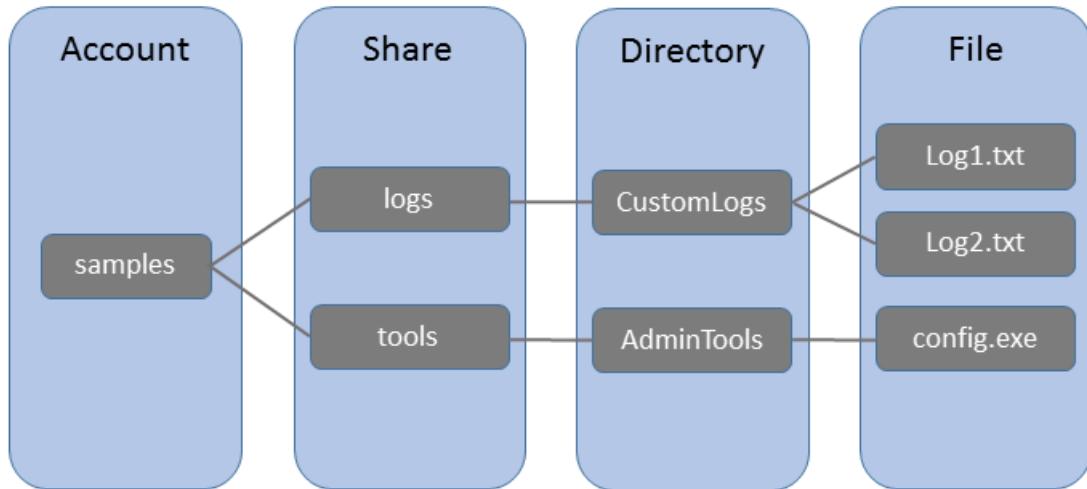
Since a File storage share is a standard file share in Azure using the SMB protocol, applications running in Azure can access data in the share via file I/O APIs. Developers can therefore leverage their existing code and skills to migrate existing applications. IT Pros can use PowerShell cmdlets to create, mount, and manage File storage shares as part of the administration of Azure applications. This guide will show examples of both.

Common uses of File storage include:

- Migrating on-premises applications that rely on file shares to run on Azure virtual machines or cloud services, without expensive rewrites
- Storing shared application settings, for example in configuration files
- Storing diagnostic data such as logs, metrics, and crash dumps in a shared location
- Storing tools and utilities needed for developing or administering Azure virtual machines or cloud services

File storage concepts

File storage contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Share:** A File storage share is an SMB file share in Azure. All directories and files must be created in a parent share. An account can contain an unlimited number of shares, and a share can store an unlimited number of files, up to the 5 TB total capacity of the file share.
- **Directory:** An optional hierarchy of directories.
- **File:** A file in the share. A file may be up to 1 TB in size.
- **URL format:** Files are addressable using the following URL format:
`https://<storage account>.file.core.windows.net/<share>/<directory/directories>/<file>`

The following example URL could be used to address one of the files in the diagram above:

```
http://samples.file.core.windows.net/logs/CustomLogs/Log1.txt
```

For details about how to name shares, directories, and files, see [Naming and Referencing Shares, Directories, Files, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a C++ application

To build the samples, you will need to install the Azure Storage Client Library 2.4.0 for C++. You should also have created an Azure storage account.

To install the Azure Storage Client 2.4.0 for C++, you can use one of the following methods:

- **Linux:** Follow the instructions given in the [Azure Storage Client Library for C++ README](#) page.
- **Windows:** In Visual Studio, click **Tools > NuGet Package Manager > Package Manager Console**. Type the following command into the [NuGet Package Manager console](#) and press **ENTER**.

```
Install-Package wasstorage
```

Set up your application to use File storage

Add the following include statements to the top of the C++ file where you want to use the Azure storage APIs to access files:

```
#include "was/storage_account.h"
#include "was/file.h"
```

Set up an Azure storage connection string

To use File storage, you need to connect to your Azure storage account. The first step would be to configure a connection string which we'll use to connect to your storage account. Let's define a static variable to do that.

```
// Define the connection-string with your values.
const utility::string_t
storage_connection_string(U("DefaultEndpointsProtocol=https;AccountName=your_storage_account;AccountKey=your_storage_account_key"));
```

Connecting to an Azure storage account

You can use the **cloud_storage_account** class to represent your Storage Account information. To retrieve your storage account information from the storage connection string, you can use the **parse** method.

```
// Retrieve storage account from connection string.
azure::storage::cloud_storage_account storage_account =
    azure::storage::cloud_storage_account::parse(storage_connection_string);
```

How to: Create a Share

All files and directories in File storage reside in a container called a **Share**. Your storage account can have as many shares as your account capacity allows. To obtain access to a share and its contents, you need to use a File storage client.

```
// Create the file storage client.
azure::storage::cloud_file_client file_client =
    storage_account.create_cloud_file_client();
```

Using the File storage client, you can then obtain a reference to a share.

```
// Get a reference to the file share
azure::storage::cloud_file_share share =
    file_client.get_share_reference(_XPLATSTR("my-sample-share"));
```

To create the share, use the **create_if_not_exists** method of the **cloud_file_share** object.

```
if (share.create_if_not_exists()) {
    std::wcout << U("New share created") << std::endl;
}
```

At this point, **share** holds a reference to a share named **my-sample-share**.

How to: Upload a file

At the very least, an Azure File Storage Share contains a root directory where files can reside. In this section, you'll learn how to upload a file from local storage onto the root directory of a share.

The first step in uploading a file is to obtain a reference to the directory where it should reside. You do this by calling the **get_root_directory_reference** method of the share object.

```
//Get a reference to the root directory for the share.
azure::storage::cloud_file_directory root_dir = share.get_root_directory_reference();
```

Now that you have a reference to the root directory of the share, you can upload a file onto it. This example uploads from a file, from text, and from a stream.

```
// Upload a file from a stream.
concurrency::streams::istream input_stream =
    concurrency::streams::file_stream<uint8_t>::open_istream(_XPLATSTR("DataFile.txt")).get();

azure::storage::cloud_file file1 =
    root_dir.get_file_reference(_XPLATSTR("my-sample-file-1"));
file1.upload_from_stream(input_stream);

// Upload some files from text.
azure::storage::cloud_file file2 =
    root_dir.get_file_reference(_XPLATSTR("my-sample-file-2"));
file2.upload_text(_XPLATSTR("more text"));

// Upload a file from a file.
azure::storage::cloud_file file4 =
    root_dir.get_file_reference(_XPLATSTR("my-sample-file-3"));
file4.upload_from_file(_XPLATSTR("DataFile.txt"));
```

How to: Create a Directory

You can also organize storage by putting files inside subdirectories instead of having all of them in the root directory. The Azure file storage service allows you to create as much directories as your account will allow. The code below will create a directory named **my-sample-directory** under the root directory as well as a subdirectory named **my-sample-subdirectory**.

```

// Retrieve a reference to a directory
azure::storage::cloud_file_directory directory = share.get_directory_reference(_XPLATSTR("my-sample-
directory"));

// Return value is true if the share did not exist and was successfully created.
directory.create_if_not_exists();

// Create a subdirectory.
azure::storage::cloud_file_directory subdirectory =
    directory.get_subdirectory_reference(_XPLATSTR("my-sample-subdirectory"));
subdirectory.create_if_not_exists();

```

How to: List files and directories in a share

Obtaining a list of files and directories within a share is easily done by calling **list_files_and_directories** on a **cloud_file_directory** reference. To access the rich set of properties and methods for a returned **list_file_and_directory_item**, you must call the **list_file_and_directory_item.as_file** method to get a **cloud_file** object, or the **list_file_and_directory_item.as_directory** method to get a **cloud_file_directory** object.

The following code demonstrates how to retrieve and output the URI of each item in the root directory of the share.

```

//Get a reference to the root directory for the share.
azure::storage::cloud_file_directory root_dir =
    share.get_root_directory_reference();

// Output URI of each item.
azure::storage::list_file_and_directory_result_iterator end_of_results;

for (auto it = directory.list_files_and_directories(); it != end_of_results; ++it)
{
    if(it->is_directory())
    {
        ucout << "Directory: " << it->as_directory().uri().primary_uri().to_string() << std::endl;
    }
    else if (it->is_file())
    {
        ucout << "File: " << it->as_file().uri().primary_uri().to_string() << std::endl;
    }
}

```

How to: Download a file

To download files, first retrieve a file reference and then call the **download_to_stream** method to transfer the file contents to a stream object which you can then persist to a local file. Alternatively, you can use the **download_to_file** method to download the contents of a file to a local file. You can use the **download_text** method to download the contents of a file as a text string.

The following example uses the **download_to_stream** and **download_text** methods to demonstrate downloading the files which were created in previous sections.

```

// Download as text
azure::storage::cloud_file text_file =
    root_dir.get_file_reference(_XPLATSTR("my-sample-file-2"));
utility::string_t text = text_file.download_text();
ucout << "File Text: " << text << std::endl;

// Download as a stream.
azure::storage::cloud_file stream_file =
    root_dir.get_file_reference(_XPLATSTR("my-sample-file-3"));

concurrency::streams::container_buffer<std::vector<uint8_t>> buffer;
concurrency::streams::ostream output_stream(buffer);
stream_file.download_to_stream(output_stream);
std::ofstream outfile("DownloadFile.txt", std::ofstream::binary);
std::vector<unsigned char>& data = buffer.collection();
outfile.write((char *)&data[0], buffer.size());
outfile.close();

```

How to: Delete a file

Another common file storage operation is file deletion. The following code deletes a file named my-sample-file-3 stored under the root directory.

```

// Get a reference to the root directory for the share.
azure::storage::cloud_file_share share =
    file_client.get_share_reference(_XPLATSTR("my-sample-share"));

azure::storage::cloud_file_directory root_dir =
    share.get_root_directory_reference();

azure::storage::cloud_file file =
    root_dir.get_file_reference(_XPLATSTR("my-sample-file-3"));

file.delete_file_if_exists();

```

How to: Delete a directory

Deleting a directory is a simple task, although it should be noted that you cannot delete a directory that still contains files or other directories.

```

// Get a reference to the share.
azure::storage::cloud_file_share share =
    file_client.get_share_reference(_XPLATSTR("my-sample-share"));

// Get a reference to the directory.
azure::storage::cloud_file_directory directory =
    share.get_directory_reference(_XPLATSTR("my-sample-directory"));

// Get a reference to the subdirectory you want to delete.
azure::storage::cloud_file_directory sub_directory =
    directory.get_subdirectory_reference(_XPLATSTR("my-sample-subdirectory"));

// Delete the subdirectory and the sample directory.
sub_directory.delete_directory_if_exists();

directory.delete_directory_if_exists();

```

How to: Delete a Share

Deleting a share is done by calling the **delete_if_exists** method on a `cloud_file_share` object. Here's sample code

that does that.

```
// Get a reference to the share.  
azure::storage::cloud_file_share share =  
    file_client.get_share_reference(_XPLATSTR("my-sample-share"));  
  
// delete the share if exists  
share.delete_share_if_exists();
```

Set the maximum size for a file share

You can set the quota (or maximum size) for a file share, in gigabytes. You can also check to see how much data is currently stored on the share.

By setting the quota for a share, you can limit the total size of the files stored on the share. If the total size of files on the share exceeds the quota set on the share, then clients will be unable to increase the size of existing files or create new files, unless those files are empty.

The example below shows how to check the current usage for a share and how to set the quota for the share.

```
// Parse the connection string for the storage account.  
azure::storage::cloud_storage_account storage_account =  
    azure::storage::cloud_storage_account::parse(storage_connection_string);  
  
// Create the file client.  
azure::storage::cloud_file_client file_client =  
    storage_account.create_cloud_file_client();  
  
// Get a reference to the share.  
azure::storage::cloud_file_share share =  
    file_client.get_share_reference(_XPLATSTR("my-sample-share"));  
if (share.exists())  
{  
    std::cout << "Current share usage: " << share.download_share_usage() << "/" << share.properties().quota();  
  
    //This line sets the quota to 2560GB  
    share.resize(2560);  
  
    std::cout << "Quota increased: " << share.download_share_usage() << "/" << share.properties().quota();  
}
```

Generate a shared access signature for a file or file share

You can generate a shared access signature (SAS) for a file share or for an individual file. You can also create a shared access policy on a file share to manage shared access signatures. Creating a shared access policy is recommended, as it provides a means of revoking the SAS if it should be compromised.

The following example creates a shared access policy on a share, and then uses that policy to provide the constraints for a SAS on a file in the share.

```

// Parse the connection string for the storage account.
azure::storage::cloud_storage_account storage_account =
    azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the file client and get a reference to the share
azure::storage::cloud_file_client file_client =
    storage_account.create_cloud_file_client();

azure::storage::cloud_file_share share =
    file_client.get_share_reference(_XPLATSTR("my-sample-share"));

if (share.exists())
{
    // Create and assign a policy
    utility::string_t policy_name = _XPLATSTR("sampleSharePolicy");

    azure::storage::file_shared_access_policy sharedPolicy =
        azure::storage::file_shared_access_policy();

    //set permissions to expire in 90 minutes
    sharedPolicy.set_expiry(utility::datetime::utc_now() +
        utility::datetime::from_minutes(90));

    //give read and write permissions
    sharedPolicy.set_permissions(azure::storage::file_shared_access_policy::permissions::write |
        azure::storage::file_shared_access_policy::permissions::read);

    //set permissions for the share
    azure::storage::file_share_permissions permissions;

    //retrieve the current list of shared access policies
    azure::storage::shared_access_policies<azure::storage::file_shared_access_policy> policies;

    //add the new shared policy
    policies.insert(std::make_pair(policy_name, sharedPolicy));

    //save the updated policy list
    permissions.set_policies(policies);
    share.upload_permissions(permissions);

    //Retrieve the root directory and file references
    azure::storage::cloud_file_directory root_dir =
        share.get_root_directory_reference();
    azure::storage::cloud_file file =
        root_dir.get_file_reference(_XPLATSTR("my-sample-file-1"));

    // Generate a SAS for a file in the share
    // and associate this access policy with it.
    utility::string_t sas_token = file.get_shared_access_signature(sharedPolicy);

    // Create a new CloudFile object from the SAS, and write some text to the file.
    azure::storage::cloud_file
    file_with_sas(azure::storage::storage_credentials(sas_token).transform_uri(file.uri().primary_uri()));
    utility::string_t text = _XPLATSTR("My sample content");
    file_with_sas.upload_text(text);

    //Download and print URL with SAS.
    utility::string_t downloaded_text = file_with_sas.download_text();
    ucout << downloaded_text << std::endl;
    ucout << azure::storage::storage_credentials(sas_token).transform_uri(file.uri().primary_uri()).to_string()
    << std::endl;
}

}

```

For more information about creating and using shared access signatures, see [Using Shared Access Signatures \(SAS\)](#).

Next Steps

To learn more about Azure Storage, explore these resources:

- [Storage Client Library for C++](#)
- [Azure Storage Explorer](#)
- [Azure Storage Documentation](#)

How to use Azure File storage from Python

1/17/2017 • 5 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

This article will show you how to perform common scenarios using File storage. The samples are written in Python and use the [Microsoft Azure Storage SDK for Python](#). The scenarios covered include uploading, listing, downloading, and deleting files.

What is Azure File storage?

File storage offers shared storage for applications using the standard SMB 2.1 or SMB 3.0 protocol. Microsoft Azure virtual machines and cloud services can share file data across application components via mounted shares, and on-premises applications can access file data in a share via the File storage API.

Applications running in Azure virtual machines or cloud services can mount a File storage share to access file data, just as a desktop application would mount a typical SMB share. Any number of Azure virtual machines or roles can mount and access the File storage share simultaneously.

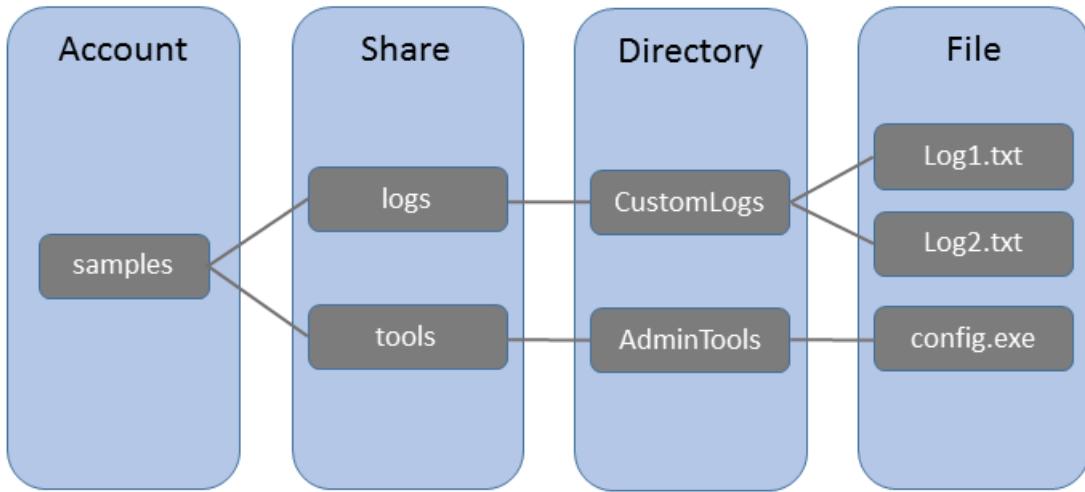
Since a File storage share is a standard file share in Azure using the SMB protocol, applications running in Azure can access data in the share via file I/O APIs. Developers can therefore leverage their existing code and skills to migrate existing applications. IT Pros can use PowerShell cmdlets to create, mount, and manage File storage shares as part of the administration of Azure applications. This guide will show examples of both.

Common uses of File storage include:

- Migrating on-premises applications that rely on file shares to run on Azure virtual machines or cloud services, without expensive rewrites
- Storing shared application settings, for example in configuration files
- Storing diagnostic data such as logs, metrics, and crash dumps in a shared location
- Storing tools and utilities needed for developing or administering Azure virtual machines or cloud services

File storage concepts

File storage contains the following components:



- **Storage Account:** All access to Azure Storage is done through a storage account. See [Azure Storage Scalability and Performance Targets](#) for details about storage account capacity.
- **Share:** A File storage share is an SMB file share in Azure. All directories and files must be created in a parent share. An account can contain an unlimited number of shares, and a share can store an unlimited number of files, up to the 5 TB total capacity of the file share.
- **Directory:** An optional hierarchy of directories.
- **File:** A file in the share. A file may be up to 1 TB in size.
- **URL format:** Files are addressable using the following URL format:
`https://<storage account>.file.core.windows.net/<share>/<directory/directories>/<file>`

The following example URL could be used to address one of the files in the diagram above:

```
http://samples.file.core.windows.net/logs/CustomLogs/Log1.txt
```

For details about how to name shares, directories, and files, see [Naming and Referencing Shares, Directories, Files, and Metadata](#).

Create an Azure storage account

The easiest way to create your first Azure storage account is by using the [Azure Portal](#). To learn more, see [Create a storage account](#).

You can also create an Azure storage account by using [Azure PowerShell](#), [Azure CLI](#), or the [Storage Resource Provider Client Library for .NET](#).

If you prefer not to create a storage account at this time, you can also use the Azure storage emulator to run and test your code in a local environment. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

Create a share

The **FileService** object lets you work with shares, directories and files. The following code creates a **FileService** object. Add the following near the top of any Python file in which you wish to programmatically access Azure Storage.

```
from azure.storage.file import FileService
```

The following code creates a **FileService** object using the storage account name and account key. Replace 'myaccount' and 'mykey' with your account name and key.

```
file_service = **FileService** (account_name='myaccount', account_key='mykey')
```

In the following code example, you can use a **FileService** object to create the share if it doesn't exist.

```
file_service.create_share('myshare')
```

Upload a file into a share

An Azure File Storage Share contains at the very least, a root directory where files can reside. In this section, you'll learn how to upload a file from local storage onto the root directory of a share.

To create a file and upload data, use the **create_file_from_path**, **create_file_from_stream**, **create_file_from_bytes** or **create_file_from_text** methods. They are high-level methods that perform the necessary chunking when the size of the data exceeds 64 MB.

create_file_from_path uploads the contents of a file from the specified path, and **create_file_from_stream** uploads the contents from an already opened file/stream. **create_file_from_bytes** uploads an array of bytes, and **create_file_from_text** uploads the specified text value using the specified encoding (defaults to UTF-8).

The following example uploads the contents of the **sunset.png** file into the **myfile** file.

```
from azure.storage.file import ContentSettings
file_service.create_file_from_path(
    'myshare',
    None, # We want to create this blob in the root directory, so we specify None for the directory_name
    'myfile',
    'sunset.png',
    content_settings=ContentSettings(content_type='image/png'))
```

How to: Create a Directory

You can also organize storage by putting files inside sub-directories instead of having all of them in the root directory. The Azure file storage service allows you to create as many directories as your account will allow. The code below will create a sub-directory named **sampledir** under the root directory.

```
file_service.create_directory('myshare', 'sampledir')
```

How to: List files and directories in a share

To list the files and directories in a share, use the **list_directories_and_files** method. This method returns a generator. The following code outputs the **name** of each file and directory in a share to the console.

```
generator = file_service.listDirectoriesAndFiles('myshare')
for file_or_dir in generator:
    print(file_or_dir.name)
```

Download files

To download data from a file, use **get_file_to_path**, **get_file_to_stream**, **get_file_to_bytes**, or **get_file_to_text**. They are high-level methods that perform the necessary chunking when the size of the data exceeds 64 MB.

The following example demonstrates using **get_file_to_path** to download the contents of the **myfile** file and store it to the **out-sunset.png** file.

```
file_service.get_file_to_path('myshare', None, 'myfile', 'out-sunset.png')
```

Delete a file

Finally, to delete a file, call **delete_file**.

```
file_service.delete_file('myshare', None, 'myfile')
```

Next steps

Now that you've learned the basics of File storage, follow these links to learn more.

- [Python Developer Center](#)
- [Azure Storage Services REST API](#)
- [Azure Storage Team Blog](#)
- [Microsoft Azure Storage SDK for Python](#)

About Azure storage accounts

1/17/2017 • 11 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

An Azure storage account provides a unique namespace to store and access your Azure Storage data objects. All objects in a storage account are billed together as a group. By default, the data in your account is available only to you, the account owner.

There are two types of storage accounts:

General-purpose Storage Accounts

A general-purpose storage account gives you access to Azure Storage services such as Tables, Queues, Files, Blobs and Azure virtual machine disks under a single account. This type of storage account has two performance tiers:

- A standard storage performance tier which allows you to store Tables, Queues, Files, Blobs and Azure virtual machine disks.
- A premium storage performance tier which currently only supports Azure virtual machine disks. See [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#) for an in-depth overview of Premium storage.

Blob Storage Accounts

A Blob storage account is a specialized storage account for storing your unstructured data as blobs (objects) in Azure Storage. Blob storage accounts are similar to your existing general-purpose storage accounts and share all the great durability, availability, scalability, and performance features that you use today including 100% API consistency for block blobs and append blobs. For applications requiring only block or append blob storage, we recommend using Blob storage accounts.

NOTE

Blob storage accounts support only block and append blobs, and not page blobs.

Blob storage accounts expose the **Access Tier** attribute which can be specified during account creation and modified later as needed. There are two types of access tiers that can be specified based on your data access pattern:

- A **Hot** access tier which indicates that the objects in the storage account will be more frequently accessed. This allows you to store data at a lower access cost.
- A **Cool** access tier which indicates that the objects in the storage account will be less frequently accessed. This allows you to store data at a lower data storage cost.

If there is a change in the usage pattern of your data, you can also switch between these access tiers at any

time. Changing the access tier may result in additional charges. Please see [Pricing and billing for Blob storage accounts](#) for more details.

For more details on Blob storage accounts, see [Azure Blob Storage: Cool and Hot tiers](#).

Before you can create a storage account, you must have an Azure subscription, which is a plan that gives you access to a variety of Azure services. You can get started with Azure with a [free account](#). Once you decide to purchase a subscription plan, you can choose from a variety of [purchase options](#). If you're an [MSDN subscriber](#), you get free monthly credits that you can use with Azure services, including Azure Storage. See [Azure Storage Pricing](#) for information on volume pricing.

To learn how to create a storage account, see [Create a storage account](#) for more details. You can create up to 100 uniquely named storage accounts with a single subscription. See [Azure Storage Scalability and Performance Targets](#) for details about storage account limits.

Storage account billing

You are billed for Azure Storage usage based on your storage account. Storage costs are based on the following factors: region/location, account type, storage capacity, replication scheme, storage transactions, and data egress.

- Region refers to the geographical region in which your account is based.
- Account type refers to whether you are using a general-purpose storage account or a Blob storage account. With a Blob storage account, the access tier also determines the billing model for the account.
- Storage capacity refers to how much of your storage account allotment you are using to store data.
- Replication determines how many copies of your data are maintained at one time, and in what locations.
- Transactions refer to all read and write operations to Azure Storage.
- Data egress refers to data transferred out of an Azure region. When the data in your storage account is accessed by an application that is not running in the same region, you are charged for data egress. (For Azure services, you can take steps to group your data and services in the same data centers to reduce or eliminate data egress charges.)

The [Azure Storage Pricing](#) page provides detailed pricing information based on account type, storage capacity, replication, and transactions. The [Data Transfers Pricing Details](#) provides detailed pricing information for data egress. You can use the [Azure Storage Pricing Calculator](#) to help estimate your costs.

NOTE

When you create an Azure virtual machine, a storage account is created for you automatically in the deployment location if you do not already have a storage account in that location. So it's not necessary to follow the steps below to create a storage account for your virtual machine disks. The storage account name will be based on the virtual machine name. See the [Azure Virtual Machines documentation](#) for more details.

Storage account endpoints

Every object that you store in Azure Storage has a unique URL address. The storage account name forms the subdomain of that address. The combination of subdomain and domain name, which is specific to each service, forms an *endpoint* for your storage account.

For example, if your storage account is named *mystorageaccount*, then the default endpoints for your storage account are:

- Blob service: <http://mystorageaccount.blob.core.windows.net>
- Table service: <http://mystorageaccount.table.core.windows.net>

- Queue service: <http://mystorageaccount.queue.core.windows.net>
- File service: <http://mystorageaccount.file.core.windows.net>

NOTE

A Blob storage account only exposes the Blob service endpoint.

The URL for accessing an object in a storage account is built by appending the object's location in the storage account to the endpoint. For example, a blob address might have this format:
<http://mystorageaccount.blob.core.windows.net/mycontainer/myblob>.

You can also configure a custom domain name to use with your storage account. For classic storage accounts, see [Configure a custom domain Name for your Blob Storage Endpoint](#) for details. For Resource Manager storage accounts, this capability has not been added to the [Azure portal](#) yet, but you can configure it with PowerShell. For more information, see the [Set-AzureRmStorageAccount](#) cmdlet.

Create a storage account

1. Sign in to the [Azure portal](#).
2. On the Hub menu, select **New -> Storage -> Storage account**.
3. Enter a name for your storage account. See [Storage account endpoints](#) for details about how the storage account name will be used to address your objects in Azure Storage.

NOTE

Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only.

Your storage account name must be unique within Azure. The Azure portal will indicate if the storage account name you select is already in use.

4. Specify the deployment model to be used: **Resource Manager** or **Classic**. **Resource Manager** is the recommended deployment model. For more information, see [Understanding Resource Manager deployment and classic deployment](#).

NOTE

Blob storage accounts can only be created using the Resource Manager deployment model.

5. Select the type of storage account: **General purpose** or **Blob storage**. **General purpose** is the default.

If **General purpose** was selected, then specify the performance tier: **Standard** or **Premium**. The default is **Standard**. For more details on standard and premium storage accounts, see [Introduction to Microsoft Azure Storage](#) and [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#).

If **Blob Storage** was selected, then specify the access tier: **Hot** or **Cool**. The default is **Hot**. See [Azure Blob Storage: Cool and Hot tiers](#) for more details.

6. Select the replication option for the storage account: **LRS**, **GRS**, **RA-GRS**, or **ZRS**. The default is **RA-GRS**. For more details on Azure Storage replication options, see [Azure Storage replication](#).
7. Select the subscription in which you want to create the new storage account.
8. Specify a new resource group or select an existing resource group. For more information on resource

- groups, see [Azure Resource Manager overview](#).
9. Select the geographic location for your storage account. See [Azure Regions](#) for more information about what services are available in which region.
 10. Click **Create** to create the storage account.

Manage your storage account

Change your account configuration

After you create your storage account, you can modify its configuration, such as changing the replication option used for the account or changing the access tier for a Blob storage account. In the [Azure portal](#), navigate to your storage account, click **All settings** and then click **Configuration** to view and/or change the account configuration.

NOTE

Depending on the performance tier you chose when creating the storage account, some replication options may not be available.

Changing the replication option will change your pricing. For more details, see [Azure Storage Pricing](#) page.

For Blob storage accounts, changing the access tier may incur charges for the change in addition to changing your pricing. Please see the [Blob storage accounts - Pricing and Billing](#) for more details.

Manage your storage access keys

When you create a storage account, Azure generates two 512-bit storage access keys, which are used for authentication when the storage account is accessed. By providing two storage access keys, Azure enables you to regenerate the keys with no interruption to your storage service or access to that service.

NOTE

We recommend that you avoid sharing your storage access keys with anyone else. To permit access to storage resources without giving out your access keys, you can use a *shared access signature*. A shared access signature provides access to a resource in your account for an interval that you define and with the permissions that you specify. See [Using Shared Access Signatures \(SAS\)](#) for more information.

View and copy storage access keys

In the [Azure portal](#), navigate to your storage account, click **All settings** and then click **Access keys** to view, copy, and regenerate your account access keys. The **Access Keys** blade also includes pre-configured connection strings using your primary and secondary keys that you can copy to use in your applications.

Regenerate storage access keys

We recommend that you change the access keys to your storage account periodically to help keep your storage connections secure. Two access keys are assigned so that you can maintain connections to the storage account by using one access key while you regenerate the other access key.

WARNING

Regenerating your access keys can affect services in Azure as well as your own applications that are dependent on the storage account. All clients that use the access key to access the storage account must be updated to use the new key.

Media services - If you have media services that are dependent on your storage account, you must re-sync the access keys with your media service after you regenerate the keys.

Applications - If you have web applications or cloud services that use the storage account, you will lose the connections if you regenerate keys, unless you roll your keys.

Storage Explorers - If you are using any [storage explorer applications](#), you will probably need to update the storage key used by those applications.

Here is the process for rotating your storage access keys:

1. Update the connection strings in your application code to reference the secondary access key of the storage account.
2. Regenerate the primary access key for your storage account. On the **Access Keys** blade, click **Regenerate Key1**, and then click **Yes** to confirm that you want to generate a new key.
3. Update the connection strings in your code to reference the new primary access key.
4. Regenerate the secondary access key in the same manner.

Delete a storage account

To remove a storage account that you are no longer using, navigate to the storage account in the [Azure portal](#), and click **Delete**. Deleting a storage account deletes the entire account, including all data in the account.

WARNING

It's not possible to restore a deleted storage account or retrieve any of the content that it contained before deletion. Be sure to back up anything you want to save before you delete the account. This also holds true for any resources in the account—once you delete a blob, table, queue, or file, it is permanently deleted.

To delete a storage account that is associated with an Azure virtual machine, you must first ensure that any virtual machine disks have been deleted. If you do not first delete your virtual machine disks, then when you attempt to delete your storage account, you will see an error message similar to:

```
Failed to delete storage account <vm-storage-account-name>. Unable to delete storage account <vm-storage-account-name>: 'Storage account <vm-storage-account-name> has some active image(s) and/or disk(s). Ensure these image(s) and/or disk(s) are removed before deleting this storage account.'
```

If the storage account uses the Classic deployment model, you can remove the virtual machine disk by following these steps in the [Azure portal](#):

1. Navigate to the [classic Azure portal](#).
2. Navigate to the Virtual Machines tab.
3. Click the Disks tab.
4. Select your data disk, then click Delete Disk.
5. To delete disk images, navigate to the Images tab and delete any images that are stored in the account.

For more information, see the [Azure Virtual Machine documentation](#).

Next steps

- [Azure Blob Storage: Cool and Hot tiers](#)
- [Azure Storage replication](#)
- [Configure Azure Storage Connection Strings](#)
- [Transfer data with the AzCopy Command-Line Utility](#)
- Visit the [Azure Storage Team Blog](#).

Azure Blob Storage: Hot and cool storage tiers

1/17/2017 • 18 min to read • [Edit on GitHub](#)

Overview

Azure Storage now offers two storage tiers for Blob object storage so that you can store your data most cost-effectively depending on how you use it. The Azure **hot storage tier** is optimized for storing data that is accessed frequently. The Azure **cool storage tier** is optimized for storing data that is infrequently accessed and long-lived. Data in the cool storage tier can tolerate slightly lower availability, but still requires high durability and similar time-to-access and throughput characteristics as hot data. For cool data a slightly lower availability SLA and higher access costs are acceptable trade-offs for much lower storage costs.

Today, data stored in the cloud is growing at an exponential pace. To manage costs for your expanding storage needs, it's helpful to organize your data based on attributes like frequency-of-access and planned retention period. Data stored in the cloud can be different in terms of how it is generated, processed, and accessed over its lifetime. Some data is actively accessed and modified throughout its lifetime. Some data is accessed frequently early in its lifetime, with access dropping drastically as the data ages. Some data remains idle in the cloud and is rarely, if ever, accessed once stored.

Each of these data access scenarios described above benefits from a differentiated tier of storage that is optimized for a particular access pattern. With the introduction of hot and cool storage tiers, Azure Blob storage now addresses this need for differentiated storage tiers with separate pricing models.

Blob storage accounts

Blob storage accounts are specialized storage accounts for storing your unstructured data as blobs (objects) in Azure Storage. With Blob storage accounts, you can now choose between hot and cool storage tiers to store your less frequently accessed cool data at a lower storage cost, and store more frequently accessed hot data at a lower access cost. Blob storage accounts are similar to your existing general-purpose storage accounts and share all the great durability, availability, scalability, and performance features that you use today, including 100% API consistency for block blobs and append blobs.

NOTE

Blob storage accounts support only block and append blobs, and not page blobs.

Blob storage accounts expose the **Access Tier** attribute, which allows you to specify the storage tier as **Hot** or **Cool** depending on the data stored in the account. If there is a change in the usage pattern of your data, you can also switch between these storage tiers at any time.

NOTE

Changing the storage tier may result in additional charges. Please see the [Pricing and Billing](#) section for more details.

Example usage scenarios for the hot storage tier include:

- Data that is in active use or expected to be accessed (read from and written to) frequently.
- Data that is staged for processing and eventual migration to the cool storage tier.

Example usage scenarios for the cool storage tier include:

- Backup, archival and disaster recovery datasets.
- Older media content not viewed frequently anymore but is expected to be available immediately when accessed.
- Large data sets that need to be stored cost effectively while more data is being gathered for future processing. (e.g., long-term storage of scientific data, raw telemetry data from a manufacturing facility)
- Original (raw) data that must be preserved, even after it has been processed into final usable form. (e.g., Raw media files after transcoding into other formats)
- Compliance and archival data that needs to be stored for a long time and is hardly ever accessed. (e.g., Security camera footage, old X-Rays/MRIs for healthcare organizations, audio recordings and transcripts of customer calls for financial services)

See [About Azure storage accounts](#) for more information on storage accounts.

For applications requiring only block or append blob storage, we recommend using Blob storage accounts, to take advantage of the differentiated pricing model of tiered storage. However, we understand this might not be possible under certain circumstances where using general-purpose storage accounts would be the way to go, such as:

- You need to use tables, queues, or files and want your blobs stored in the same storage account. Note that there is no technical advantage to storing these in the same account other than having the same shared keys.
- You still need to use the Classic deployment model. Blob storage accounts are only available via the Azure Resource Manager deployment model.
- You need to use page blobs. Blob storage accounts do not support page blobs. We generally recommend using block blobs unless you have a specific need for page blobs.
- You use a version of the [Storage Services REST API](#) that is earlier than 2014-02-14 or a client library with a version lower than 4.x, and cannot upgrade your application.

NOTE

Blob storage accounts are currently supported in all Azure regions.

Comparison between the storage tiers

The following table highlights the comparison between the two storage tiers:

	Hot storage tier	Cool storage tier
Availability	99.9%	99%
Availability (RA-GRS reads)	99.99%	99.9%
Usage charges	Higher storage costs Lower access and transaction costs	Lower storage costs Higher access and transaction costs
Minimum object size		N/A
Minimum storage duration		N/A
Latency (Time to first byte)		milliseconds

Scalability and performance targets	Same as general-purpose storage accounts
--	--

NOTE

Blob storage accounts support the same performance and scalability targets as general-purpose storage accounts. See [Azure Storage Scalability and Performance Targets](#) for more information.

Pricing and Billing

Blob storage accounts use a new pricing model for blob storage based on the storage tier. When using a Blob storage account, the following billing considerations apply:

- **Storage costs:** In addition to the amount of data stored, the cost of storing data varies depending on the storage tier. The per-gigabyte cost is lower for the cool storage tier than for the hot storage tier.
- **Data access costs:** For data in the cool storage tier, you will be charged a per-gigabyte data access charge for reads and writes.
- **Transaction costs:** There is a per-transaction charge for both tiers. However, the per-transaction cost for the cool storage tier is higher than that for the hot storage tier.
- **Geo-Replication data transfer costs:** This only applies to accounts with geo-replication configured, including GRS and RA-GRS. Geo-replication data transfer incurs a per-gigabyte charge.
- **Outbound data transfer costs:** Outbound data transfers (data that is transferred out of an Azure region) incur billing for bandwidth usage on a per-gigabyte basis, consistent with general-purpose storage accounts.
- **Changing the storage tier:** Changing the storage tier from cool to hot will incur a charge equal to reading all the data existing in the storage account for every transition. On the other hand, changing the storage tier from hot to cool will be free of cost.

NOTE

In order to allow users to try out the new storage tiers and validate functionality post launch, the charge for changing the storage tier from cool to hot will be waived off until June 30th 2016. Starting July 1st 2016, the charge will be applied to all transitions from cool to hot. For more details on the pricing model for Blob storage accounts see, [Azure Storage Pricing](#) page. For more details on the outbound data transfer charges see, [Data Transfers Pricing Details](#) page.

Quick Start

In this section we will demonstrate the following scenarios using the Azure portal:

- How to create a Blob storage account.
- How to manage a Blob storage account.

Using the Azure portal

Create a Blob storage account using the Azure portal

1. Sign in to the [Azure portal](#).
2. On the Hub menu, select **New > Data + Storage > Storage account**.
3. Enter a name for your storage account.

This name must be globally unique; it is used as part of the URL used to access the objects in the storage account.

4. Select **Resource Manager** as the deployment model.

Tiered storage can only be used with Resource Manager storage accounts; this is the recommended

deployment model for new resources. For more information, check out the [Azure Resource Manager overview](#).

5. In the Account Kind dropdown list, select **Blob Storage**.

This is where you select the type of storage account. Tiered storage is not available in general-purpose storage; it is only available in the Blob storage type account.

Note that when you select this, the performance tier is set to Standard. Tiered storage is not available with the Premium performance tier.

6. Select the replication option for the storage account: **LRS**, **GRS**, or **RA-GRS**. The default is **RA-GRS**.

LRS = locally redundant storage; GRS = geo-redundant storage (2 regions); RA-GRS is read-access geo-redundant storage (2 regions with read access to the second).

For more details on Azure Storage replication options, check out [Azure Storage replication](#).

7. Select the right storage tier for your needs: Set the **Access tier** to either **Cool** or **Hot**. The default is **Hot**.

8. Select the subscription in which you want to create the new storage account.

9. Specify a new resource group or select an existing resource group. For more information on resource groups, see [Azure Resource Manager overview](#).

10. Select the region for your storage account.

11. Click **Create** to create the storage account.

Change the storage tier of a Blob storage account using the Azure portal

1. Sign in to the [Azure portal](#).
2. To navigate to your storage account, select All Resources, then select your storage account.
3. In the Settings blade, click **Configuration** to view and/or change the account configuration.
4. Select the right storage tier for your needs: Set the **Access tier** to either **Cool** or **Hot**.
5. Click Save at the top of the blade.

NOTE

Changing the storage tier may result in additional charges. Please see the [Pricing and Billing](#) section for more details.

Evaluating and migrating to Blob storage accounts

The purpose of this section is to help users to make a smooth transition to using Blob storage accounts. There are two user scenarios:

- You have an existing general-purpose storage account and want to evaluate a change to a Blob storage account with the right storage tier.
- You have decided to use a Blob storage account or already have one and want to evaluate whether you should use the hot or cool storage tier.

In both cases, the first order of business is to estimate the cost of storing and accessing your data stored in a Blob storage account and compare that against your current costs.

Evaluating Blob storage account tiers

In order to estimate the cost of storing and accessing data stored in a Blob storage account, you will need to evaluate your existing usage pattern or approximate your expected usage pattern. In general, you will want to know:

- Your storage consumption - How much data is being stored and how does this change on a monthly basis?

- Your storage access pattern - How much data is being read from and written to the account (including new data)? How many transactions are used for data access, and what kinds of transactions are they?

Monitoring existing storage accounts

To monitor your existing storage accounts and gather this data, you can make use of Azure Storage Analytics which performs logging and provides metrics data for a storage account. Storage Analytics can store metrics that include aggregated transaction statistics and capacity data about requests to the Blob storage service for both general-purpose storage accounts as well as Blob storage accounts. This data is stored in well-known tables in the same storage account.

For more details, please see [About Storage Analytics Metrics](#) and [Storage Analytics Metrics Table Schema](#)

NOTE

Blob storage accounts expose the table service endpoint only for storing and accessing the metrics data for that account.

To monitor the storage consumption for the Blob storage service, you will need to enable the capacity metrics. With this enabled, capacity data is recorded daily for a storage account's Blob service, and recorded as a table entry that is written to the `$MetricsCapacityBlob` table within the same storage account.

To monitor the data access pattern for the Blob storage service, you will need to enable the hourly transaction metrics at an API level. With this enabled, per API transactions are aggregated every hour, and recorded as a table entry that is written to the `$MetricsHourPrimaryTransactionsBlob` table within the same storage account. The `$MetricsHourSecondaryTransactionsBlob` table records the transactions to the secondary endpoint in case of RA-GRS storage accounts.

NOTE

In case you have a general-purpose storage account in which you have stored page blobs and virtual machine disks alongside block and append blob data, this estimation process is not applicable. This is because you will have no way of distinguishing capacity and transaction metrics based on the type of blob for only block and append blobs which can be migrated to a Blob storage account.

To get a good approximation of your data consumption and access pattern, we recommend you choose a retention period for the metrics that is representative of your regular usage, and extrapolate. One option is to retain the metrics data for 7 days and collect the data every week, for analysis at the end of the month. Another option is to retain the metrics data for the last 30 days and collect and analyze the data at the end of the 30-day period.

For details on enabling, collecting and viewing metrics data, please see, [Enabling Azure Storage metrics and viewing metrics data](#).

NOTE

Storing, accessing and downloading analytics data is also charged just like regular user data.

Utilizing usage metrics to estimate costs

Storage costs

The latest entry in the capacity metrics table `$MetricsCapacityBlob` with the row key 'data' shows the storage capacity consumed by user data. The latest entry in the capacity metrics table `$MetricsCapacityBlob` with the row key 'analytics' shows the storage capacity consumed by the analytics logs.

This total capacity consumed by both user data and analytics logs (if enabled) can then be used to estimate the cost of storing data in the storage account. The same method can also be used for estimating storage costs for

block and append blobs in general-purpose storage accounts.

Transaction costs

The sum of '*TotalBillableRequests*', across all entries for an API in the transaction metrics table indicates the total number of transactions for that particular API. e.g., the total number of '*GetBlob*' transactions in a given period can be calculated by the sum of total billable requests for all entries with the row key '*user;GetBlob*'.

In order to estimate transaction costs for Blob storage accounts, you will need to break down the transactions into three groups since they are priced differently.

- Write transactions such as '*PutBlob*', '*PutBlock*', '*PutBlockList*', '*AppendBlock*', '*ListBlobs*', '*ListContainers*', '*CreateContainer*', '*SnapshotBlob*', and '*CopyBlob*'.
- Delete transactions such as '*DeleteBlob*' and '*DeleteContainer*'.
- All other transactions.

In order to estimate transaction costs for general-purpose storage accounts, you need to aggregate all transactions irrespective of the operation/API.

Data access and geo-replication data transfer costs

While storage analytics does not provide the amount of data read from and written to a storage account, it can be roughly estimated by looking at the transaction metrics table. The sum of '*TotalIngress*' across all entries for an API in the transaction metrics table indicates the total amount of ingress data in bytes for that particular API. Similarly the sum of '*TotalEgress*' indicates the total amount of egress data, in bytes.

In order to estimate the data access costs for Blob storage accounts, you will need to break down the transactions into two groups.

- The amount of data retrieved from the storage account can be estimated by looking at the sum of '*TotalEgress*' for primarily the '*GetBlob*' and '*CopyBlob*' operations.
- The amount of data written to the storage account can be estimated by looking at the sum of '*TotalIngress*' for primarily the '*PutBlob*', '*PutBlock*', '*CopyBlob*' and '*AppendBlock*' operations.

The cost of geo-replication data transfer for Blob storage accounts can also be calculated by using the estimate for the amount of data written in case of a GRS or RA-GRS storage account.

NOTE

For a more detailed example about calculating the costs for using the hot or cool storage tier, please take a look at the FAQ titled '*What are Hot and Cool access tiers and how should I determine which one to use?*' in the [Azure Storage Pricing Page](#).

Migrating existing data

A Blob storage account is specialized for storing only block and append blobs. Existing general-purpose storage accounts, which allow you to store tables, queues, files and disks, as well as blobs, cannot be converted to Blob storage accounts. To use the storage tiers, you will need to create new Blob storage accounts and migrate your existing data into the newly created accounts. You can use the following methods to migrate existing data into Blob storage accounts from on-premise storage devices, from third-party cloud storage providers, or from your existing general-purpose storage accounts in Azure:

AzCopy

AzCopy is a Windows command-line utility designed for high-performance copying of data to and from Azure Storage. You can use AzCopy to copy data into your Blob storage account from your existing general-purpose storage accounts, or to upload data from your on-premises storage devices into your Blob storage account.

For more details, see [Transfer data with the AzCopy Command-Line Utility](#).

Data Movement Library

Azure Storage data movement library for .NET is based on the core data movement framework that powers AzCopy. The library is designed for high-performance, reliable and easy data transfer operations similar to AzCopy. This allows you to take full benefits of the features provided by AzCopy in your application natively without having to deal with running and monitoring external instances of AzCopy.

For more details, see [Azure Storage Data Movement Library for .Net](#)

REST API or Client Library

You can create a custom application to migrate your data into a Blob storage account using one of the Azure client libraries or the Azure storage services REST API. Azure Storage provides rich client libraries for multiple languages and platforms like .NET, Java, C++, Node.JS, PHP, Ruby, and Python. The client libraries offer advanced capabilities such as retry logic, logging, and parallel uploads. You can also develop directly against the REST API, which can be called by any language that makes HTTP/HTTPS requests.

For more details, see [Get Started with Azure Blob storage](#).

NOTE

Blobs encrypted using client-side encryption store encryption-related metadata stored with the blob. It is absolutely critical that any copy mechanism should ensure that the blob metadata, and especially the encryption-related metadata, is preserved. If you copy the blobs without this metadata, the blob content will not be retrievable again. For more details regarding encryption-related metadata, see [Azure Storage Client-Side Encryption](#).

FAQs

1. Are existing storage accounts still available?

Yes, existing storage accounts are still available and are unchanged in pricing or functionality. They do not have the ability to choose a storage tier and will not have tiering capabilities in the future.

2. Why and when should I start using Blob storage accounts?

Blob storage accounts are specialized for storing blobs and allow us to introduce new blob-centric features. Going forward, Blob storage accounts are the recommended way for storing blobs, as future capabilities such as hierarchical storage and tiering will be introduced based on this account type. However, it is up to you when you would like to migrate based on your business requirements.

3. Can I convert my existing storage account to a Blob storage account?

No. Blob storage account is a different kind of storage account and you will need to create it new and migrate your data as explained above.

4. Can I store objects in both storage tiers in the same account?

The '*Access Tier*' attribute which indicates the storage tier is set at an account level and applies to all objects in that account. You cannot set the access tier attribute at an object level.

5. Can I change the storage tier of my Blob storage account?

Yes. You will be able to change the storage tier by setting the '*Access Tier*' attribute on the storage account. Changing the storage tier will apply to all objects stored in the account. Change the storage tier from hot to cool will not incur any charges, while changing from cool to hot will incur a per GB cost for reading all the data in the account.

6. How frequently can I change the storage tier of my Blob storage account?

While we do not enforce a limitation on how frequently the storage tier can be changed, please be aware that changing the storage tier from cool to hot will incur significant charges. We do not recommend

changing the storage tier frequently.

7. Will the blobs in the cool storage tier behave differently than the ones in the hot storage tier?

Blobs in the hot storage tier have the same latency as blobs in general-purpose storage accounts. Blobs in the cool storage tier have a similar latency (in milliseconds) as blobs in general-purpose storage accounts.

Blobs in the cool storage tier will have a slightly lower availability service level (SLA) than the blobs stored in the hot storage tier. For more details, see [SLA for storage](#).

8. Can I store page blobs and virtual machine disks in Blob storage accounts?

Blob storage accounts support only block and append blobs, and not page blobs. Azure virtual machine disks are backed by page blobs and as a result Blob storage accounts cannot be used to store virtual machine disks. However it is possible to store backups of the virtual machine disks as block blobs in a Blob storage account.

9. Will I need to change my existing applications to use Blob storage accounts?

Blob storage accounts are 100% API consistent with general-purpose storage accounts for block and append blobs. As long as your application is using block blobs or append blobs, and you are using the 2014-02-14 version of the [Storage Services REST API](#) or greater your application should work. If you are using an older version of the protocol, then you will need to update your application to use the new version so as to work seamlessly with both types of storage accounts. In general, we always recommend using the latest version regardless of which storage account type you use.

10. Will there be a change in user experience?

Blob storage accounts are very similar to a general-purpose storage accounts for storing block and append blobs, and support all the key features of Azure Storage, including high durability and availability, scalability, performance, and security. Other than the features and restrictions specific to Blob storage accounts and its storage tiers that have been called out above, everything else remains the same.

Next steps

Evaluate Blob storage accounts

[Check availability of Blob storage accounts by region](#)

[Evaluate usage of your current storage accounts by enabling Azure Storage metrics](#)

[Check Blob storage pricing by region](#)

[Check data transfers pricing](#)

Start using Blob storage accounts

[Get Started with Azure Blob storage](#)

[Moving data to and from Azure Storage](#)

[Transfer data with the AzCopy Command-Line Utility](#)

[Browse and explore your storage accounts](#)

Configure a custom domain name for your Blob storage endpoint

1/17/2017 • 7 min to read • [Edit on GitHub](#)

Overview

You can configure a custom domain for accessing blob data in your Azure storage account. The default endpoint for Blob storage is `<storage-account-name>.blob.core.windows.net`. If you map a custom domain and subdomain such as **www.contoso.com** to the blob endpoint for your storage account, then your users can also access blob data in your storage account using that domain.

IMPORTANT

Azure Storage does not yet support HTTPS with custom domains. We are aware that customers are interested in this feature, and it will be available in a future release.

There are two ways to point your custom domain to the blob endpoint for your storage account. The simplest way is to create a CNAME record mapping your custom domain and subdomain to the blob endpoint. A CNAME record is a DNS feature that maps a source domain to a destination domain. In this case, the source domain is your custom domain and subdomain--note that the subdomain is always required. The destination domain is your Blob service endpoint.

The process of mapping your custom domain to your blob endpoint can, however, result in a brief period of downtime for the domain while you are registering the domain in the [Azure Classic Portal](#). If your custom domain is currently supporting an application with a service-level agreement (SLA) that requires that there be no downtime, then you can use the Azure **asverify** subdomain to provide an intermediate registration step so that users will be able to access your domain while the DNS mapping takes place.

The following table shows sample URLs for accessing blob data in a storage account named **mystorageaccount**. The custom domain registered for the storage account is **www.contoso.com**:

RESOURCE TYPE	URL FORMATS
Storage account	Default URL: http://mystorageaccount.blob.core.windows.net Custom domain URL: http://www.contoso.com
Blob	Default URL: http://mystorageaccount.blob.core.windows.net/mycontainer/myblob Custom domain URL: http://www.contoso.com/mycontainer/myblob
Root container	Default URL: http://mystorageaccount.blob.core.windows.net/myblob or http://mystorageaccount.blob.core.windows.net/\$root/myblob Custom domain URL: http://www.contoso.com/myblob or http://www.contoso.com/\$root/myblob

Register a custom domain for your storage account

Use this procedure to register your custom domain if you do not have concerns about having the domain be briefly unavailable to users, or if your custom domain is not currently hosting an application.

If your custom domain is currently supporting an application that cannot have any downtime, then use the procedure outlined in [Register a custom domain for your storage account using the intermediary asverify subdomain](#).

To configure a custom domain name, you must create a new CNAME record with your domain registrar. The CNAME record specifies an alias for a domain name; in this case it maps the address of your custom domain to the Blob storage endpoint for your storage account.

Each registrar has a similar but slightly different method of specifying a CNAME record, but the concept is the same. Note that many basic domain registration packages do not offer DNS configuration, so you may need to upgrade your domain registration package before you can create the CNAME record.

1. In the [Azure Classic Portal](#), navigate to the **Storage** tab.
2. In the **Storage** tab, click the name of the storage account for which you want to map the custom domain.
3. Click the **Configure** tab.
4. At the bottom of the screen click **Manage Domain** to display the **Manage Custom Domain** dialog. In the text at the top of the dialog, you'll see information on how to create the CNAME record. For this procedure, ignore the text that refers to the **asverify** subdomain.
5. Log on to your DNS registrar's website, and go to the page for managing DNS. You might find this in a section such as **Domain Name, DNS, or Name Server Management**.
6. Find the section for managing CNAMEs. You may have to go to an advanced settings page and look for the words **CNAME, Alias, or Subdomains**.
7. Create a new CNAME record, and provide a subdomain alias, such as **www** or **photos**. Then provide a host name, which is your Blob service endpoint, in the format **mystorageaccount.blob.core.windows.net** (where **mystorageaccount** is the name of your storage account). The host name to use is provided for you in the text of the **Manage Custom Domain** dialog.
8. After you have created the CNAME record, return to the **Manage Custom Domain** dialog, and enter the name of your custom domain, including the subdomain, in the **Custom Domain Name** field. For example, if your domain is **contoso.com** and your subdomain is **www**, enter **www.contoso.com**; if your subdomain is **photos**, enter **photos.contoso.com**. Note that the subdomain is required.
9. Click the **Register** button to register your custom domain.

If the registration is successful, you will see the message **Your custom domain is active**. Users can now view blob data on your custom domain, so long as they have the appropriate permissions.

Register a custom domain for your storage account using the intermediary asverify subdomain

Use this procedure to register your custom domain if your custom domain is currently supporting an application with an SLA that requires that there be no downtime. By creating a CNAME that points from asverify.

<subdomain>.<customdomain> to asverify.<storageaccount>.blob.core.windows.net, you can pre-register your domain with Azure. You can then create a second CNAME that points from <subdomain>.<customdomain> to <storageaccount>.blob.core.windows.net, at which point traffic to your custom domain will be directed to your blob endpoint.

The asverify subdomain is a special subdomain recognized by Azure. By prepending **asverify** to your own subdomain, you permit Azure to recognize your custom domain without modifying the DNS record for the domain. Once you do modify the DNS record for the domain, it will be mapped to the blob endpoint with no downtime.

1. In the [Azure Classic Portal](#), navigate to the **Storage** tab.
2. In the **Storage** tab, click the name of the storage account for which you want to map the custom domain.
3. Click the **Configure** tab.
4. At the bottom of the screen click **Manage Domain** to display the **Manage Custom Domain** dialog. In the text at the top of the dialog, you'll see information on how to create the CNAME record using the **asverify** subdomain.
5. Log on to your DNS registrar's website, and go to the page for managing DNS. You might find this in a section such as **Domain Name, DNS, or Name Server Management**.
6. Find the section for managing CNAMEs. You may have to go to an advanced settings page and look for the words **CNAME, Alias, or Subdomains**.
7. Create a new CNAME record, and provide a subdomain alias that includes the asverify subdomain. For example, the subdomain you specify will be in the format **asverify.www** or **asverify.photos**. Then provide a host name, which is your Blob service endpoint, in the format **asverify.mystorageaccount.blob.core.windows.net** (where **mystorageaccount** is the name of your storage account). The host name to use is provided for you in the text of the **Manage Custom Domain** dialog.
8. After you have created the CNAME record, return to the **Manage Custom Domain** dialog, and enter the name of your custom domain in the **Custom Domain Name** field. For example, if your domain is **contoso.com** and your subdomain is **www**, enter **www.contoso.com**; if your subdomain is **photos**, enter **photos.contoso.com**. Note that the subdomain is required.
9. Click the checkbox that says **Advanced: Use the 'asverify' subdomain to preregister my custom domain**.
10. Click the **Register** button to preregister your custom domain.

If the preregistration is successful, you will see the message **Your custom domain is active**.

11. At this point, your custom domain has been verified by Azure, but traffic to your domain is not yet being routed to your storage account. To complete the process, return to your DNS registrar's website, and create another CNAME record that maps your subdomain to your Blob service endpoint. For example, specify the subdomain as **www** or **photos**, and the hostname as **mystorageaccount.blob.core.windows.net** (where **mystorageaccount** is the name of your storage account). With this step, the registration of your custom domain is complete.
12. Finally, you can delete the CNAME record you created using **asverify**, as it was necessary only as an intermediary step.

Users can now view blob data on your custom domain, so long as they have the appropriate permissions.

Verify that the custom domain references your Blob service endpoint

To verify that your custom domain is indeed mapped to your Blob service endpoint, create a blob in a public container within your storage account. Then, in a web browser, use a URI in the following format to access the blob:

```
http://<*subdomain.customdomain*>/<*mycontainer*>/<*myblob*>
```

For example, you might use the following URI to access a web form via a **photos.contoso.com** custom subdomain that maps to a blob in your **myforms** container:

```
http://photos.contoso.com/myforms/applicationform.htm
```

Unregister a custom domain from your storage account

To unregister a custom domain, follow these steps:

1. Sign in to the [Azure Classic Portal](#).
2. In the navigation pane, click **Storage**.
3. On the **Storage** page, click the name of the storage account to display the dashboard.
4. On the ribbon, click **Manage Domain**.
5. In the **Manage Custom Domain** dialog box, click **Unregister**.

Additional Resources

- [How to map Custom Domain to Content Delivery Network \(CDN\) endpoint](#)

Manage anonymous read access to containers and blobs

1/17/2017 • 4 min to read • [Edit on GitHub](#)

Overview

By default, only the owner of the storage account may access storage resources within that account. For Blob storage only, you can set a container's permissions to permit anonymous read access to the container and its blobs, so that you can grant access to those resources without sharing your account key.

Anonymous access is best for scenarios where you want certain blobs to always be available for anonymous read access. For finer-grained control, you can create a shared access signature, which enables you to delegate restricted access using different permissions and over a specified time interval. For more information about creating shared access signatures, see [Using Shared Access Signatures \(SAS\)](#).

Grant anonymous users permissions to containers and blobs

By default, a container and any blobs within it may be accessed only by the owner of the storage account. To give anonymous users read permissions to a container and its blobs, you can set the container permissions to allow public access. Anonymous users can read blobs within a publicly accessible container without authenticating the request.

Containers provide the following options for managing container access:

- **Full public read access:** Container and blob data can be read via anonymous request. Clients can enumerate blobs within the container via anonymous request, but cannot enumerate containers within the storage account.
- **Public read access for blobs only:** Blob data within this container can be read via anonymous request, but container data is not available. Clients cannot enumerate blobs within the container via anonymous request.
- **No public read access:** Container and blob data can be read by the account owner only.

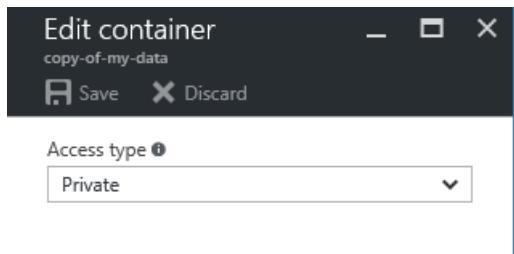
You can set container permissions in the following ways:

- From the [Azure portal](#).
- Programmatically, by using the storage client library or the REST API.
- By using PowerShell. To learn about setting container permissions from Azure PowerShell, see [Using Azure PowerShell with Azure Storage](#).

Setting container permissions from the Azure portal

To set container permissions from the [Azure portal](#), follow these steps:

1. Navigate to the dashboard for your storage account.
2. Select the container name from the list. Clicking the name exposes the blobs in the chosen container.
3. Select **Access policy** from the toolbar.
4. In the **Access type** field, select your desired level of permissions as shown in the screenshot below.



Setting container permissions programmatically using .NET

To set permissions for a container using the .NET client library, first retrieve the container's existing permissions by calling the **GetPermissions** method. Then set the **PublicAccess** property for the **BlobContainerPermissions** object that is returned by the **GetPermissions** method. Finally, call the **SetPermissions** method with the updated permissions.

The following example sets the container's permissions to full public read access. To set permissions to public read access for blobs only, set the **PublicAccess** property to **BlobContainerPublicAccessType.Blob**. To remove all permissions for anonymous users, set the property to **BlobContainerPublicAccessType.Off**.

```
public static void SetPublicContainerPermissions(CloudBlobContainer container)
{
    BlobContainerPermissions permissions = container.GetPermissions();
    permissions.PublicAccess = BlobContainerPublicAccessType.Container;
    container.SetPermissions(permissions);
}
```

Access containers and blobs anonymously

A client that accesses containers and blobs anonymously can use constructors that do not require credentials. The following examples show a few different ways to reference Blob service resources anonymously.

Create an anonymous client object

You can create a new service client object for anonymous access by providing the Blob service endpoint for the account. However, you must also know the name of a container in that account that's available for anonymous access.

```
public static void CreateAnonymousBlobClient()
{
    // Create the client object using the Blob service endpoint.
    CloudBlobClient blobClient = new CloudBlobClient(new Uri(@"https://storagesample.blob.core.windows.net"));

    // Get a reference to a container that's available for anonymous access.
    CloudBlobContainer container = blobClient.GetContainerReference("sample-container");

    // Read the container's properties. Note this is only possible when the container supports full public read
    // access.
    container.FetchAttributes();
    Console.WriteLine(container.Properties.LastModified);
    Console.WriteLine(container.Properties.ETag);
}
```

Reference a container anonymously

If you have the URL to a container that is anonymously available, you can use it to reference the container directly.

```

public static void ListBlobsAnonymously()
{
    // Get a reference to a container that's available for anonymous access.
    CloudBlobContainer container = new CloudBlobContainer(new
Uri(@"https://storagesample.blob.core.windows.net/sample-container"));

    // List blobs in the container.
    foreach (IListBlobItem blobItem in container.ListBlobs())
    {
        Console.WriteLine(blobItem.Uri);
    }
}

```

Reference a blob anonymously

If you have the URL to a blob that is available for anonymous access, you can reference the blob directly using that URL:

```

public static void DownloadBlobAnonymously()
{
    CloudBlockBlob blob = new CloudBlockBlob(new Uri(@"https://storagesample.blob.core.windows.net/sample-
container/logfile.txt"));
    blob.DownloadToFile(@"C:\Temp\logfile.txt", System.IO.FileMode.Create);
}

```

Features available to anonymous users

The following table shows which operations may be called by anonymous users when a container's ACL is set to allow public access.

REST OPERATION	PERMISSION WITH FULL PUBLIC READ ACCESS	PERMISSION WITH PUBLIC READ ACCESS FOR BLOBS ONLY
List Containers	Owner only	Owner only
Create Container	Owner only	Owner only
Get Container Properties	All	Owner only
Get Container Metadata	All	Owner only
Set Container Metadata	Owner only	Owner only
Get Container ACL	Owner only	Owner only
Set Container ACL	Owner only	Owner only
Delete Container	Owner only	Owner only
List Blobs	All	Owner only
Put Blob	Owner only	Owner only
Get Blob	All	All
Get Blob Properties	All	All

REST OPERATION	PERMISSION WITH FULL PUBLIC READ ACCESS	PERMISSION WITH PUBLIC READ ACCESS FOR BLOBS ONLY
Set Blob Properties	Owner only	Owner only
Get Blob Metadata	All	All
Set Blob Metadata	Owner only	Owner only
Put Block	Owner only	Owner only
Get Block List (committed blocks only)	All	All
Get Block List (uncommitted blocks only or all blocks)	Owner only	Owner only
Put Block List	Owner only	Owner only
Delete Blob	Owner only	Owner only
Copy Blob	Owner only	Owner only
Snapshot Blob	Owner only	Owner only
Lease Blob	Owner only	Owner only
Put Page	Owner only	Owner only
Get Page Ranges	All	All
Append Blob	Owner only	Owner only

See Also

- [Authentication for the Azure Storage Services](#)
- [Using Shared Access Signatures \(SAS\)](#)
- [Delegating Access with a Shared Access Signature](#)

Azure Storage Table Design Guide: Designing Scalable and Performant Tables

1/17/2017 • 75 min to read • [Edit on GitHub](#)

Overview

To design scalable and performant tables you must consider a number of factors such as performance, scalability, and cost. If you have previously designed schemas for relational databases, these considerations will be familiar to you, but while there are some similarities between the Azure Table service storage model and relational models, there are also many important differences. These differences typically lead to very different designs that may look counter-intuitive or wrong to someone familiar with relational databases, but which do make good sense if you are designing for a NoSQL key/value store such as the Azure Table service. Many of your design differences will reflect the fact that the Table service is designed to support cloud-scale applications that can contain billions of entities (rows in relational database terminology) of data or for datasets that must support very high transaction volumes; therefore, you need to think differently about how you store your data and understand how the Table service works. A well designed NoSQL data store can enable your solution to scale much further (and at a lower cost) than a solution that uses a relational database. This guide helps you with these topics.

About the Azure Table service

This section highlights some of the key features of the Table service that are especially relevant to designing for performance and scalability. If you are new to Azure Storage and the Table service, first read [Introduction to Microsoft Azure Storage](#) and [Get started with Azure Table Storage using .NET](#) before reading the remainder of this article. Although the focus of this guide is on the Table service, it will include some discussion of the Azure Queue and Blob services, and how you might use them along with the Table service in a solution.

What is the Table service? As you might expect from the name, the Table service uses a tabular format to store data. In the standard terminology, each row of the table represents an entity, and the columns store the various properties of that entity. Every entity has a pair of keys to uniquely identify it, and a timestamp column that the Table service uses to track when the entity was last updated (this happens automatically and you cannot manually overwrite the timestamp with an arbitrary value). The Table service uses this last-modified timestamp (LMT) to manage optimistic concurrency.

NOTE

The Table service REST API operations also return an **ETag** value that it derives from the last-modified timestamp (LMT). In this document we will use the terms ETag and LMT interchangeably because they refer to the same underlying data.

The following example shows a simple table design to store employee and department entities. Many of the examples shown later in this guide are based on this simple design.

PARTITIONKEY	ROWKEY	TIMESTAMP	

Marketing	00001	2014-08-22T00:50:32Z	<table border="1"> <thead> <tr> <th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr> </thead> <tbody> <tr> <td>Don</td><td>Hall</td><td>34</td><td>donh@contoso.com</td></tr> </tbody> </table>	FIRSTNAME	LASTNAME	AGE	EMAIL	Don	Hall	34	donh@contoso.com
FIRSTNAME	LASTNAME	AGE	EMAIL								
Don	Hall	34	donh@contoso.com								
Marketing	00002	2014-08-22T00:50:34Z	<table border="1"> <thead> <tr> <th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr> </thead> <tbody> <tr> <td>Jun</td><td>Caio</td><td>47</td><td>junc@contoso.com</td></tr> </tbody> </table>	FIRSTNAME	LASTNAME	AGE	EMAIL	Jun	Caio	47	junc@contoso.com
FIRSTNAME	LASTNAME	AGE	EMAIL								
Jun	Caio	47	junc@contoso.com								
Marketing	Department	2014-08-22T00:50:30Z	<table border="1"> <thead> <tr> <th>DEPARTMENTNAME</th><th>EMPLOYEECOUNT</th></tr> </thead> <tbody> <tr> <td>Marketing</td><td>153</td></tr> </tbody> </table>	DEPARTMENTNAME	EMPLOYEECOUNT	Marketing	153				
DEPARTMENTNAME	EMPLOYEECOUNT										
Marketing	153										
Sales	00010	2014-08-22T00:50:44Z	<table border="1"> <thead> <tr> <th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr> </thead> <tbody> <tr> <td>Ken</td><td>Kwok</td><td>23</td><td>kenk@contoso.com</td></tr> </tbody> </table>	FIRSTNAME	LASTNAME	AGE	EMAIL	Ken	Kwok	23	kenk@contoso.com
FIRSTNAME	LASTNAME	AGE	EMAIL								
Ken	Kwok	23	kenk@contoso.com								

So far, this looks very similar to a table in a relational database with the key differences being the mandatory columns, and the ability to store multiple entity types in the same table. In addition, each of the user-defined properties such as **FirstName** or **Age** has a data type, such as integer or string, just like a column in a relational database. Although unlike in a relational database, the schema-less nature of the Table service means that a property need not have the same data type on each entity. To store complex data types in a single property, you must use a serialized format such as JSON or XML. For more information about the table service such as supported data types, supported date ranges, naming rules, and size constraints, see [Understanding the Table Service Data Model](#).

As you will see, your choice of **PartitionKey** and **RowKey** is fundamental to good table design. Every entity stored in a table must have a unique combination of **PartitionKey** and **RowKey**. As with keys in a relational database table, the **PartitionKey** and **RowKey** values are indexed to create a clustered index that enables fast look-ups;

however, the Table service does not create any secondary indexes so these are the only two indexed properties (some of the patterns described later show how you can work around this apparent limitation).

A table is made up of one or more partitions, and as you will see, many of the design decisions you make will be around choosing a suitable **PartitionKey** and **RowKey** to optimize your solution. A solution could consist of just a single table that contains all your entities organized into partitions, but typically a solution will have multiple tables. Tables help you to logically organize your entities, help you manage access to the data using access control lists, and you can drop an entire table using a single storage operation.

Table partitions

The account name, table name and **PartitionKey** together identify the partition within the storage service where the table service stores the entity. As well as being part of the addressing scheme for entities, partitions define a scope for transactions (see [Entity Group Transactions](#) below), and form the basis of how the table service scales. For more information on partitions see [Azure Storage Scalability and Performance Targets](#).

In the Table service, an individual node services one or more complete partitions and the service scales by dynamically load-balancing partitions across nodes. If a node is under load, the table service can *split* the range of partitions serviced by that node onto different nodes; when traffic subsides, the service can *merge* the partition ranges from quiet nodes back onto a single node.

For more information about the internal details of the Table service, and in particular how the service manages partitions, see the paper [Microsoft Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency](#).

Entity Group Transactions

In the Table service, Entity Group Transactions (EGTs) are the only built-in mechanism for performing atomic updates across multiple entities. EGTs are also referred to as *batch transactions* in some documentation. EGTs can only operate on entities stored in the same partition (share the same partition key in a given table), so anytime you need atomic transactional behavior across multiple entities you need to ensure that those entities are in the same partition. This is often a reason for keeping multiple entity types in the same table (and partition) and not using multiple tables for different entity types. A single EGT can operate on at most 100 entities. If you submit multiple concurrent EGTs for processing it is important to ensure those EGTs do not operate on entities that are common across EGTs as otherwise processing can be delayed.

EGTs also introduce a potential trade-off for you to evaluate in your design: using more partitions will increase the scalability of your application because Azure has more opportunities for load balancing requests across nodes, but this might limit the ability of your application to perform atomic transactions and maintain strong consistency for your data. Furthermore, there are specific scalability targets at the level of a partition that might limit the throughput of transactions you can expect for a single node: for more information about the scalability targets for Azure storage accounts and the table service, see [Azure Storage Scalability and Performance Targets](#). Later sections of this guide discuss various design strategies that help you manage trade-offs such as this one, and discuss how best to choose your partition key based on the specific requirements of your client application.

Capacity considerations

The following table includes some of the key values to be aware of when you are designing a Table service solution:

TOTAL CAPACITY OF AN AZURE STORAGE ACCOUNT	500 TB
Number of tables in an Azure storage account	Limited only by the capacity of the storage account
Number of partitions in a table	Limited only by the capacity of the storage account
Number of entities in a partition	Limited only by the capacity of the storage account

TOTAL CAPACITY OF AN AZURE STORAGE ACCOUNT	500 TB
Size of an individual entity	Up to 1 MB with a maximum of 255 properties (including the PartitionKey , RowKey , and Timestamp)
Size of the PartitionKey	A string up to 1 KB in size
Size of the RowKey	A string up to 1 KB in size
Size of an Entity Group Transaction	A transaction can include at most 100 entities and the payload must be less than 4 MB in size. An EGT can only update an entity once.

For more information, see [Understanding the Table Service Data Model](#).

Cost considerations

Table storage is relatively inexpensive, but you should include cost estimates for both capacity usage and the quantity of transactions as part of your evaluation of any solution that uses the Table service. However, in many scenarios storing denormalized or duplicate data in order to improve the performance or scalability of your solution is a valid approach to take. For more information about pricing, see [Azure Storage Pricing](#).

Guidelines for table design

These lists summarize some of the key guidelines you should keep in mind when you are designing your tables, and this guide will address them all in more detail later in. These guidelines are very different from the guidelines you would typically follow for relational database design.

Designing your Table service solution to be *read* efficient:

- **Design for querying in read-heavy applications.** When you are designing your tables, think about the queries (especially the latency sensitive ones) that you will execute before you think about how you will update your entities. This typically results in an efficient and performant solution.
- **Specify both PartitionKey and RowKey in your queries.** Point queries such as these are the most efficient table service queries.
- **Consider storing duplicate copies of entities.** Table storage is cheap so consider storing the same entity multiple times (with different keys) to enable more efficient queries.
- **Consider denormalizing your data.** Table storage is cheap so consider denormalizing your data. For example, store summary entities so that queries for aggregate data only need to access a single entity.
- **Use compound key values.** The only keys you have are **PartitionKey** and **RowKey**. For example, use compound key values to enable alternate keyed access paths to entities.
- **Use query projection.** You can reduce the amount of data that you transfer over the network by using queries that select just the fields you need.

Designing your Table service solution to be *write* efficient:

- **Do not create hot partitions.** Choose keys that enable you to spread your requests across multiple partitions at any point of time.
- **Avoid spikes in traffic.** Smooth the traffic over a reasonable period of time and avoid spikes in traffic.
- **Don't necessarily create a separate table for each type of entity.** When you require atomic transactions across entity types, you can store these multiple entity types in the same partition in the same table.
- **Consider the maximum throughput you must achieve.** You must be aware of the scalability targets for the Table service and ensure that your design will not cause you to exceed them.

As you read this guide, you will see examples that put all of these principles into practice.

Design for querying

Table service solutions may be read intensive, write intensive, or a mix of the two. This section focuses on the things to bear in mind when you are designing your Table service to support read operations efficiently. Typically, a design that supports read operations efficiently is also efficient for write operations. However, there are additional considerations to bear in mind when designing to support write operations, discussed in the next section, [Design for data modification](#).

A good starting point for designing your Table service solution to enable you to read data efficiently is to ask "What queries will my application need to execute to retrieve the data it needs from the Table service?"

NOTE

With the Table service, it's important to get the design correct up front because it's difficult and expensive to change it later. For example, in a relational database it's often possible to address performance issues simply by adding indexes to an existing database: this is not an option with the Table service.

This section focuses on the key issues you must address when you design your tables for querying. The topics covered in this section include:

- [How your choice of PartitionKey and RowKey impacts query performance](#)
- [Choosing an appropriate PartitionKey](#)
- [Optimizing queries for the Table service](#)
- [Sorting data in the Table service](#)

How your choice of PartitionKey and RowKey impacts query performance

The following examples assume the table service is storing employee entities with the following structure (most of the examples omit the **Timestamp** property for clarity):

COLUMN NAME	DATA TYPE
PartitionKey (Department Name)	String
RowKey (Employee Id)	String
FirstName	String
LastName	String
Age	Integer
EmailAddress	String

The earlier section [Azure Table service overview](#) describes some of the key features of the Azure Table service that have a direct influence on designing for query. These result in the following general guidelines for designing Table service queries. Note that the filter syntax used in the examples below is from the Table service REST API, for more information see [Query Entities](#).

- A **Point Query** is the most efficient lookup to use and is recommended to be used for high-volume lookups or lookups requiring lowest latency. Such a query can use the indexes to locate an individual entity very efficiently by specifying both the **PartitionKey** and **RowKey** values. For example: \$filter=(PartitionKey eq 'Sales') and (RowKey eq '2')
- Second best is a **Range Query** that uses the **PartitionKey** and filters on a range of **RowKey** values to return more than one entity. The **PartitionKey** value identifies a specific partition, and the **RowKey** values identify a

subset of the entities in that partition. For example: \$filter=PartitionKey eq 'Sales' and RowKey ge 'S' and RowKey lt 'T'

- Third best is a **Partition Scan** that uses the **PartitionKey** and filters on another non-key property and that may return more than one entity. The **PartitionKey** value identifies a specific partition, and the property values select for a subset of the entities in that partition. For example: \$filter=PartitionKey eq 'Sales' and LastName eq 'Smith'
- A **Table Scan** does not include the **PartitionKey** and is very inefficient because it searches all of the partitions that make up your table in turn for any matching entities. It will perform a table scan regardless of whether or not your filter uses the **RowKey**. For example: \$filter=LastName eq 'Jones'
- Queries that return multiple entities return them sorted in **PartitionKey** and **RowKey** order. To avoid resorting the entities in the client, choose a **RowKey** that defines the most common sort order.

Note that using an "or" to specify a filter based on **RowKey** values results in a partition scan and is not treated as a range query. Therefore, you should avoid queries that use filters such as: \$filter=PartitionKey eq 'Sales' and (RowKey eq '121' or RowKey eq '322')

For examples of client-side code that use the Storage Client Library to execute efficient queries, see:

- [Executing a point query using the Storage Client Library](#)
- [Retrieving multiple entities using LINQ](#)
- [Server-side projection](#)

For examples of client-side code that can handle multiple entity types stored in the same table, see:

- [Working with heterogeneous entity types](#)

Choosing an appropriate PartitionKey

Your choice of **PartitionKey** should balance the need to enable the use of EGTs (to ensure consistency) against the requirement to distribute your entities across multiple partitions (to ensure a scalable solution).

At one extreme, you could store all your entities in a single partition, but this may limit the scalability of your solution and would prevent the table service from being able to load-balance requests. At the other extreme, you could store one entity per partition, which would be highly scalable and which enables the table service to load-balance requests, but which would prevent you from using entity group transactions.

An ideal **PartitionKey** is one that enables you to use efficient queries and that has sufficient partitions to ensure your solution is scalable. Typically, you will find that your entities will have a suitable property that distributes your entities across sufficient partitions.

NOTE

For example, in a system that stores information about users or employees, UserID may be a good PartitionKey. You may have several entities that use a given UserID as the partition key. Each entity that stores data about a user is grouped into a single partition, and so these entities are accessible via entity group transactions, while still being highly scalable.

There are additional considerations in your choice of **PartitionKey** that relate to how you will insert, update, and delete entities: see the section [Design for data modification](#) below.

Optimizing queries for the Table service

The Table service automatically indexes your entities using the **PartitionKey** and **RowKey** values in a single clustered index, hence the reason that point queries are the most efficient to use. However, there are no indexes other than that on the clustered index on the **PartitionKey** and **RowKey**.

Many designs must meet requirements to enable lookup of entities based on multiple criteria. For example, locating employee entities based on email, employee id, or last name. The following patterns in the section [Table Design Patterns](#) address these types of requirement and describe ways of working around the fact that the Table service

does not provide secondary indexes:

- [Intra-partition secondary index pattern](#) - Store multiple copies of each entity using different **RowKey** values (in the same partition) to enable fast and efficient lookups and alternate sort orders by using different **RowKey** values.
- [Inter-partition secondary index pattern](#) - Store multiple copies of each entity using different RowKey values in separate partitions or in separate tables to enable fast and efficient lookups and alternate sort orders by using different **RowKey** values.
- [Index Entities Pattern](#) - Maintain index entities to enable efficient searches that return lists of entities.

Sorting data in the Table service

The Table service returns entities sorted in ascending order based on **PartitionKey** and then by **RowKey**. These keys are string values and to ensure that numeric values sort correctly, you should convert them to a fixed length and pad them with zeroes. For example, if the employee id value you use as the **RowKey** is an integer value, you should convert employee id **123** to **00000123**.

Many applications have requirements to use data sorted in different orders: for example, sorting employees by name, or by joining date. The following patterns in the section [Table Design Patterns](#) address how to alternate sort orders for your entities:

- [Intra-partition secondary index pattern](#) - Store multiple copies of each entity using different RowKey values (in the same partition) to enable fast and efficient lookups and alternate sort orders by using different RowKey values.
- [Inter-partition secondary index pattern](#) - Store multiple copies of each entity using different RowKey values in separate partitions in separate tables to enable fast and efficient lookups and alternate sort orders by using different RowKey values.
- [Log tail pattern](#) - Retrieve the n entities most recently added to a partition by using a **RowKey** value that sorts in reverse date and time order.

Design for data modification

This section focuses on the design considerations for optimizing inserts, updates, and deletes. In some cases, you will need to evaluate the trade-off between designs that optimize for querying against designs that optimize for data modification just as you do in designs for relational databases (although the techniques for managing the design trade-offs are different in a relational database). The section [Table Design Patterns](#) describes some detailed design patterns for the Table service and highlights some of these trade-offs. In practice, you will find that many designs optimized for querying entities also work well for modifying entities.

Optimizing the performance of insert, update, and delete operations

To update or delete an entity, you must be able to identify it by using the **PartitionKey** and **RowKey** values. In this respect, your choice of **PartitionKey** and **RowKey** for modifying entities should follow similar criteria to your choice to support point queries because you want to identify entities as efficiently as possible. You do not want to use an inefficient partition or table scan to locate an entity in order to discover the **PartitionKey** and **RowKey** values you need to update or delete it.

The following patterns in the section [Table Design Patterns](#) address optimizing the performance of your insert, update, and delete operations:

- [High volume delete pattern](#) - Enable the deletion of a high volume of entities by storing all the entities for simultaneous deletion in their own separate table; you delete the entities by deleting the table.
- [Data series pattern](#) - Store complete data series in a single entity to minimize the number of requests you make.
- [Wide entities pattern](#) - Use multiple physical entities to store logical entities with more than 252 properties.
- [Large entities pattern](#) - Use blob storage to store large property values.

Ensuring consistency in your stored entities

The other key factor that influences your choice of keys for optimizing data modifications is how to ensure consistency by using atomic transactions. You can only use an EGT to operate on entities stored in the same partition.

The following patterns in the section [Table Design Patterns](#) address managing consistency:

- [Intra-partition secondary index pattern](#) - Store multiple copies of each entity using different **RowKey** values (in the same partition) to enable fast and efficient lookups and alternate sort orders by using different **RowKey** values.
- [Inter-partition secondary index pattern](#) - Store multiple copies of each entity using different RowKey values in separate partitions or in separate tables to enable fast and efficient lookups and alternate sort orders by using different **RowKey** values.
- [Eventually consistent transactions pattern](#) - Enable eventually consistent behavior across partition boundaries or storage system boundaries by using Azure queues.
- [Index Entities Pattern](#) - Maintain index entities to enable efficient searches that return lists of entities.
- [Denormalization pattern](#) - Combine related data together in a single entity to enable you to retrieve all the data you need with a single point query.
- [Data series pattern](#) - Store complete data series in a single entity to minimize the number of requests you make.

For information about entity group transactions, see the section [Entity Group Transactions](#).

Ensuring your design for efficient modifications facilitates efficient queries

In many cases, a design for efficient querying results in efficient modifications, but you should always evaluate whether this is the case for your specific scenario. Some of the patterns in the section [Table Design Patterns](#) explicitly evaluate trade-offs between querying and modifying entities, and you should always take into account the number of each type of operation.

The following patterns in the section [Table Design Patterns](#) address trade-offs between designing for efficient queries and designing for efficient data modification:

- [Compound key pattern](#) - Use compound **RowKey** values to enable a client to lookup related data with a single point query.
- [Log tail pattern](#) - Retrieve the n entities most recently added to a partition by using a **RowKey** value that sorts in reverse date and time order.

Encrypting Table Data

The .NET Azure Storage Client Library supports encryption of string entity properties for insert and replace operations. The encrypted strings are stored on the service as binary properties, and they are converted back to strings after decryption.

For tables, in addition to the encryption policy, users must specify the properties to be encrypted. This can be done by either specifying an [EncryptProperty] attribute (for POCO entities that derive from TableEntity) or an encryption resolver in request options. An encryption resolver is a delegate that takes a partition key, row key, and property name and returns a Boolean that indicates whether that property should be encrypted. During encryption, the client library will use this information to decide whether a property should be encrypted while writing to the wire. The delegate also provides for the possibility of logic around how properties are encrypted. (For example, if X, then encrypt property A; otherwise encrypt properties A and B.) Note that it is not necessary to provide this information while reading or querying entities.

Note that merge is not currently supported. Since a subset of properties may have been encrypted previously using a different key, simply merging the new properties and updating the metadata will result in data loss. Merging either requires making extra service calls to read the pre-existing entity from the service, or using a new key per property, both of which are not suitable for performance reasons.

For information about encrypting table data, see [Client-Side Encryption and Azure Key Vault for Microsoft Azure Storage](#).

Modelling relationships

Building domain models is a key step in the design of complex systems. Typically, you use the modelling process to identify entities and the relationships between them as a way to understand the business domain and inform the design of your system. This section focuses on how you can translate some of the common relationship types found in domain models to designs for the Table service. The process of mapping from a logical data-model to a physical NoSQL based data-model is very different from that used when designing a relational database. Relational databases design typically assumes a data normalization process optimized for minimizing redundancy – and a declarative querying capability that abstracts how the implementation of how the database works.

One-to-many relationships

One-to-many relationships between business domain objects occur very frequently: for example, one department has many employees. There are several ways to implement one-to-many relationships in the Table service each with pros and cons that may be relevant to the particular scenario.

Consider the example of a large multi-national corporation with tens of thousands of departments and employee entities where every department has many employees and each employee is associated with one specific department. One approach is to store separate department and employee entities such as these:

Department entity	Employee entity
<code>PartitionKey (Department name)</code> <code>RowKey ("Department")</code> ----- <code>ManagerName (string)</code> <code>ManagerEmailAddress (string)</code>	<code>PartitionKey (Department name)</code> <code>RowKey (Email address)</code> ----- <code>FirstName (string)</code> <code>LastName (string)</code> <code>Age (integer)</code> <code>Id (string)</code>

This example shows an implicit one-to-many relationship between the types based on the **PartitionKey** value. Each department can have many employees.

This example also shows a department entity and its related employee entities in the same partition. You could choose to use different partitions, tables, or even storage accounts for the different entity types.

An alternative approach is to denormalize your data and store only employee entities with denormalized department data as shown in the following example. In this particular scenario, this denormalized approach may not be the best if you have a requirement to be able to change the details of a department manager because to do this you need to update every employee in the department.

Employee entity
<code>PartitionKey (Department name)</code> <code>RowKey (Email address)</code> ----- <code>FirstName (string)</code> <code>LastName (string)</code> <code>Age (integer)</code> <code>Id (string)</code> <code>ManagerName (string)</code> <code>ManagerEmailAddress (string)</code>

For more information, see the [Denormalization pattern](#) later in this guide.

The following table summarizes the pros and cons of each of the approaches outlined above for storing employee and department entities that have a one-to-many relationship. You should also consider how often you expect to perform various operations: it may be acceptable to have a design that includes an expensive operation if that operation only happens infrequently.

APPROACH	PROS	CONS
Separate entity types, same partition, same table	<ul style="list-style-type: none"> You can update a department entity with a single operation. You can use an EGT to maintain consistency if you have a requirement to modify a department entity whenever you update/insert/delete an employee entity. For example if you maintain a departmental employee count for each department. 	<ul style="list-style-type: none"> You may need to retrieve both an employee and a department entity for some client activities. Storage operations happen in the same partition. At high transaction volumes, this may result in a hotspot. You cannot move an employee to a new department using an EGT.
Separate entity types, different partitions or tables or storage accounts	<ul style="list-style-type: none"> You can update a department entity or employee entity with a single operation. At high transaction volumes, this may help spread the load across more partitions. 	<ul style="list-style-type: none"> You may need to retrieve both an employee and a department entity for some client activities. You cannot use EGTs to maintain consistency when you update/insert/delete an employee and update a department. For example, updating an employee count in a department entity. You cannot move an employee to a new department using an EGT.
Denormalize into single entity type	<ul style="list-style-type: none"> You can retrieve all the information you need with a single request. 	<ul style="list-style-type: none"> It may be expensive to maintain consistency if you need to update department information (this would require you to update all the employees in a department).

How you choose between these options, and which of the pros and cons are most significant, depends on your specific application scenarios. For example, how often do you modify department entities; do all your employee queries need the additional departmental information; how close are you to the scalability limits on your partitions or your storage account?

One-to-one relationships

Domain models may include one-to-one relationships between entities. If you need to implement a one-to-one relationship in the Table service, you must also choose how to link the two related entities when you need to retrieve them both. This link can be either implicit, based on a convention in the key values, or explicit by storing a link in the form of **PartitionKey** and **RowKey** values in each entity to its related entity. For a discussion of whether you should store the related entities in the same partition, see the section [One-to-many relationships](#).

Note that there are also implementation considerations that might lead you to implement one-to-one relationships in the Table service:

- Handling large entities (for more information, see [Large Entities Pattern](#)).
- Implementing access controls (for more information, see [Controlling access with Shared Access Signatures](#)).

Join in the client

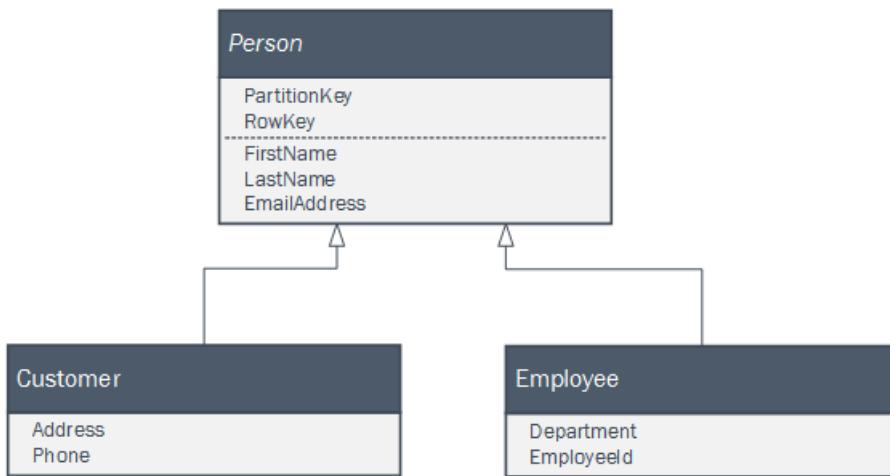
Although there are ways to model relationships in the Table service, you should not forget that the two prime reasons for using the Table service are scalability and performance. If you find you are modelling many

relationships that compromise the performance and scalability of your solution, you should ask yourself if it is necessary to build all the data relationships into your table design. You may be able to simplify the design and improve the scalability and performance of your solution if you let your client application perform any necessary joins.

For example, if you have small tables that contain data that does not change very often, then you can retrieve this data once and cache it on the client. This can avoid repeated roundtrips to retrieve the same data. In the examples we have looked at in this guide, the set of departments in a small organization is likely to be small and change infrequently making it a good candidate for data that client application can download once and cache as look up data.

Inheritance relationships

If your client application uses a set of classes that form part of an inheritance relationship to represent business entities, you can easily persist those entities in the Table service. For example, you might have the following set of classes defined in your client application where **Person** is an abstract class.



You can persist instances of the two concrete classes in the Table service using a single **Person** table using entities in that look like this:

Customer entity	Employee entity
<pre>PartitionKey RowKey PersonType ("Customer") FirstName (string) LastName (string) EmailAddress (string) Address (string) Phone (string)</pre>	<pre>PartitionKey RowKey PersonType ("Employee") FirstName (string) LastName (string) EmailAddress (string) Department (string) EmployeeID (string)</pre>

For more information about working with multiple entity types in the same table in client code, see the section [Working with heterogeneous entity types](#) later in this guide. This provides examples of how to recognize the entity type in client code.

Table Design Patterns

In previous sections, you have seen some detailed discussions about how to optimize your table design for both retrieving entity data using queries and for inserting, updating, and deleting entity data. This section describes some patterns appropriate for use with Table service solutions. In addition, you will see how you can practically address some of the issues and trade-offs raised previously in this guide. The following diagram summarizes the

relationships between the different patterns:



The pattern map above highlights some relationships between patterns (blue) and anti-patterns (orange) that are documented in this guide. There are of course many other patterns that are worth considering. For example, one of the key scenarios for Table Service is to use the [Materialized View Pattern](#) from the [Command Query Responsibility Segregation \(CQRS\)](#) pattern.

Intra-partition secondary index pattern

Store multiple copies of each entity using different **RowKey** values (in the same partition) to enable fast and efficient lookups and alternate sort orders by using different **RowKey** values. Updates between copies can be kept consistent using EGT's.

Context and problem

The Table service automatically indexes entities using the **PartitionKey** and **RowKey** values. This enables a client application to retrieve an entity efficiently using these values. For example, using the table structure shown below, a client application can use a point query to retrieve an individual employee entity by using the department name and the employee id (the **PartitionKey** and **RowKey** values). A client can also retrieve entities sorted by employee id within each department.

Employee entity

PartitionKey (Department name)
RowKey (Employee Id)

FirstName (string)
LastName (string)
Age (integer)
EmailAddress (string)

If you also want to be able to find an employee entity based on the value of another property, such as email address, you must use a less efficient partition scan to find a match. This is because the table service does not provide secondary indexes. In addition, there is no option to request a list of employees sorted in a different order than **RowKey** order.

Solution

To work around the lack of secondary indexes, you can store multiple copies of each entity with each copy using a different **RowKey** value. If you store an entity with the structures shown below, you can efficiently retrieve employee entities based on email address or employee id. The prefix values for the **RowKey**, "empid_" and "email_" enable you to query for a single employee or a range of employees by using a range of email addresses or employee ids.

Employee entity	Employee entity
PartitionKey (Department name) RowKey ("empid_" + Employee Id) ----- FirstName (string) LastName (string) Age (integer) EmailAddress (string)	PartitionKey (Department name) RowKey ("email_" + Email address) ----- FirstName (string) LastName (string) Age (integer) EmployeeId (string)

The following two filter criteria (one looking up by employee id and one looking up by email address) both specify point queries:

- \$filter=(PartitionKey eq 'Sales') and (RowKey eq 'empid_000223')
- \$filter=(PartitionKey eq 'Sales') and (RowKey eq 'email_jonesj@contoso.com')

If you query for a range of employee entities, you can specify a range sorted in employee id order, or a range sorted in email address order by querying for entities with the appropriate prefix in the **RowKey**.

- To find all the employees in the Sales department with an employee id in the range 000100 to 000199 use:
\$filter=(PartitionKey eq 'Sales') and (RowKey ge 'empid_000100') and (RowKey le 'empid_000199')
- To find all the employees in the Sales department with an email address starting with the letter 'a' use:
\$filter=(PartitionKey eq 'Sales') and (RowKey ge 'email_a') and (RowKey lt 'email_b')

Note that the filter syntax used in the examples above is from the Table service REST API, for more information see [Query Entities](#).

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Table storage is relatively cheap to use so the cost overhead of storing duplicate data should not be a major concern. However, you should always evaluate the cost of your design based on your anticipated storage requirements and only add duplicate entities to support the queries your client application will execute.
- Because the secondary index entities are stored in the same partition as the original entities, you should ensure that you do not exceed the scalability targets for an individual partition.
- You can keep your duplicate entities consistent with each other by using EGTs to update the two copies of the entity atomically. This implies that you should store all copies of an entity in the same partition. For more information, see the section [Using Entity Group Transactions](#).
- The value used for the **RowKey** must be unique for each entity. Consider using compound key values.
- Padding numeric values in the **RowKey** (for example, the employee id 000223), enables correct sorting and filtering based on upper and lower bounds.
- You do not necessarily need to duplicate all the properties of your entity. For example, if the queries that lookup the entities using the email address in the **RowKey** never need the employee's age, these entities could have the following structure:

Employee entity
PartitionKey (Department name) RowKey ("email_" + Email address) ----- FirstName (string) LastName (string) EmployeeId (string)

- It is typically better to store duplicate data and ensure that you can retrieve all the data you need with a single query, than to use one query to locate an entity and another to lookup the required data.

When to use this pattern

Use this pattern when your client application needs to retrieve entities using a variety of different keys, when your client needs to retrieve entities in different sort orders, and where you can identify each entity using a variety of unique values. However, you should be sure that you do not exceed the partition scalability limits when you are performing entity lookups using the different **RowKey** values.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Inter-partition secondary index pattern](#)
- [Compound key pattern](#)
- [Entity Group Transactions](#)
- [Working with heterogeneous entity types](#)

Inter-partition secondary index pattern

Store multiple copies of each entity using different **RowKey** values in separate partitions or in separate tables to enable fast and efficient lookups and alternate sort orders by using different **RowKey** values.

Context and problem

The Table service automatically indexes entities using the **PartitionKey** and **RowKey** values. This enables a client application to retrieve an entity efficiently using these values. For example, using the table structure shown below, a client application can use a point query to retrieve an individual employee entity by using the department name and the employee id (the **PartitionKey** and **RowKey** values). A client can also retrieve entities sorted by employee id within each department.

Employee entity
PartitionKey (Department name)
RowKey (Employee Id)
FirstName (string)
LastName (string)
Age (integer)
EmailAddress (string)

If you also want to be able to find an employee entity based on the value of another property, such as email address, you must use a less efficient partition scan to find a match. This is because the table service does not provide secondary indexes. In addition, there is no option to request a list of employees sorted in a different order than **RowKey** order.

You are anticipating a very high volume of transactions against these entities and want to minimize the risk of the Table service throttling your client.

Solution

To work around the lack of secondary indexes, you can store multiple copies of each entity with each copy using different **PartitionKey** and **RowKey** values. If you store an entity with the structures shown below, you can efficiently retrieve employee entities based on email address or employee id. The prefix values for the **PartitionKey**, "empid_" and "email_" enable you to identify which index you want to use for a query.

Employee entity (primary index)	Employee entity (secondary index)
PartitionKey ("empid_" + Department name)	PartitionKey ("email_" + Department name)
RowKey (Employee Id)	RowKey (Email address)
FirstName (string)	FirstName (string)
LastName (string)	LastName (string)
Age (integer)	Age (integer)
EmailAddress (string)	EmailId (string)

The following two filter criteria (one looking up by employee id and one looking up by email address) both specify point queries:

- `$filter=(PartitionKey eq 'empid_Sales') and (RowKey eq '000223')`
- `$filter=(PartitionKey eq 'email_Sales') and (RowKey eq 'jonesj@contoso.com')`

If you query for a range of employee entities, you can specify a range sorted in employee id order, or a range sorted in email address order by querying for entities with the appropriate prefix in the **RowKey**.

- To find all the employees in the Sales department with an employee id in the range **000100** to **000199** sorted in employee id order use: `$filter=(PartitionKey eq 'empid_Sales') and (RowKey ge '000100') and (RowKey le '000199')`
- To find all the employees in the Sales department with an email address that starts with 'a' sorted in email address order use: `$filter=(PartitionKey eq 'email_Sales') and (RowKey ge 'a') and (RowKey lt 'b')`

Note that the filter syntax used in the examples above is from the Table service REST API, for more information see [Query Entities](#).

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- You can keep your duplicate entities eventually consistent with each other by using the [Eventually consistent transactions pattern](#) to maintain the primary and secondary index entities.
- Table storage is relatively cheap to use so the cost overhead of storing duplicate data should not be a major concern. However, you should always evaluate the cost of your design based on your anticipated storage requirements and only add duplicate entities to support the queries your client application will execute.
- The value used for the **RowKey** must be unique for each entity. Consider using compound key values.
- Padding numeric values in the **RowKey** (for example, the employee id 000223), enables correct sorting and filtering based on upper and lower bounds.
- You do not necessarily need to duplicate all the properties of your entity. For example, if the queries that lookup the entities using the email address in the **RowKey** never need the employee's age, these entities could have the following structure:



- It is typically better to store duplicate data and ensure that you can retrieve all the data you need with a single query than to use one query to locate an entity using the secondary index and another to lookup the required data in the primary index.

When to use this pattern

Use this pattern when your client application needs to retrieve entities using a variety of different keys, when your client needs to retrieve entities in different sort orders, and where you can identify each entity using a variety of unique values. Use this pattern when you want to avoid exceeding the partition scalability limits when you are performing entity lookups using the different **RowKey** values.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Eventually consistent transactions pattern](#)
- [Intra-partition secondary index pattern](#)
- [Compound key pattern](#)

- Entity Group Transactions
- Working with heterogeneous entity types

Eventually consistent transactions pattern

Enable eventually consistent behavior across partition boundaries or storage system boundaries by using Azure queues.

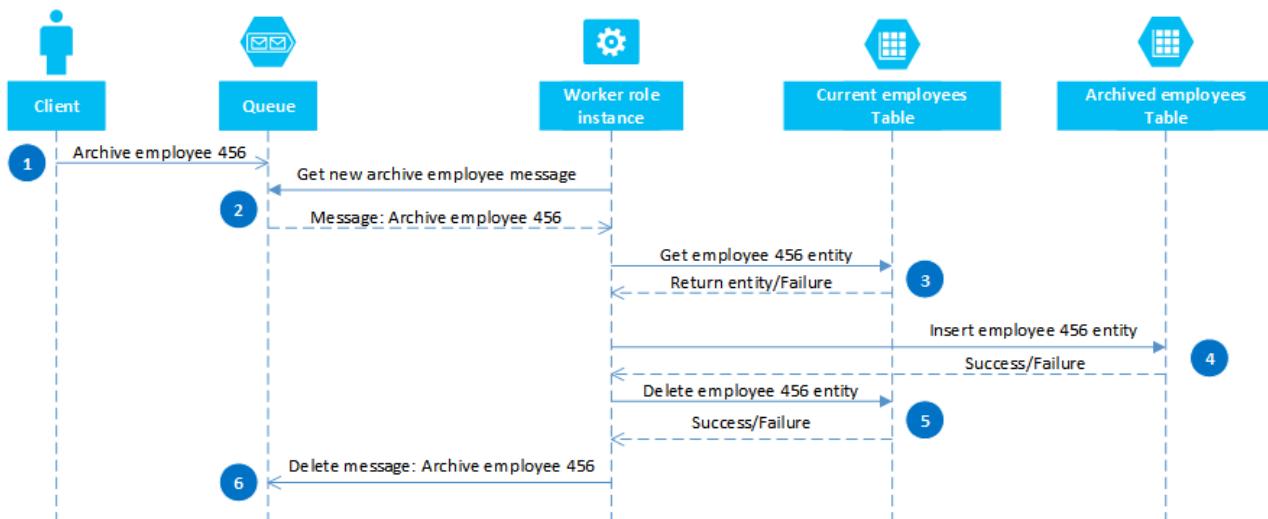
Context and problem

EGTs enable atomic transactions across multiple entities that share the same partition key. For performance and scalability reasons, you might decide to store entities that have consistency requirements in separate partitions or in a separate storage system: in such a scenario, you cannot use EGTs to maintain consistency. For example, you might have a requirement to maintain eventual consistency between:

- Entities stored in two different partitions in the same table, in different tables, in different storage accounts.
- An entity stored in the Table service and a blob stored in the Blob service.
- An entity stored in the Table service and a file in a file system.
- An entity stored in the Table service yet indexed using the Azure Search service.

Solution

By using Azure queues, you can implement a solution that delivers eventual consistency across two or more partitions or storage systems. To illustrate this approach, assume you have a requirement to be able to archive old employee entities. Old employee entities are rarely queried and should be excluded from any activities that deal with current employees. To implement this requirement you store active employees in the **Current** table and old employees in the **Archive** table. Archiving an employee requires you to delete the entity from the **Current** table and add the entity to the **Archive** table, but you cannot use an EGT to perform these two operations. To avoid the risk that a failure causes an entity to appear in both or neither tables, the archive operation must be eventually consistent. The following sequence diagram outlines the steps in this operation. More detail is provided for exception paths in the text following.



A client initiates the archive operation by placing a message on an Azure queue, in this example to archive employee #456. A worker role polls the queue for new messages; when it finds one, it reads the message and leaves a hidden copy on the queue. The worker role next fetches a copy of the entity from the **Current** table, inserts a copy in the **Archive** table, and then deletes the original from the **Current** table. Finally, if there were no errors from the previous steps, the worker role deletes the hidden message from the queue.

In this example, step 4 inserts the employee into the **Archive** table. It could add the employee to a blob in the Blob service or a file in a file system.

Recovering from failures

It is important that the operations in steps **4** and **5** must be *idempotent* in case the worker role needs to restart the archive operation. If you are using the Table service, for step **4** you should use an "insert or replace" operation; for

step **5** you should use a "delete if exists" operation in the client library you are using. If you are using another storage system, you must use an appropriate idempotent operation.

If the worker role never completes step **6**, then after a timeout the message reappears on the queue ready for the worker role to try to reprocess it. The worker role can check how many times a message on the queue has been read and, if necessary, flag it as a "poison" message for investigation by sending it to a separate queue. For more information about reading queue messages and checking the dequeue count, see [Get Messages](#).

Some errors from the Table and Queue services are transient errors, and your client application should include suitable retry logic to handle them.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- This solution does not provide for transaction isolation. For example, a client could read the **Current** and **Archive** tables when the worker role was between steps **4** and **5**, and see an inconsistent view of the data. Note that the data will be consistent eventually.
- You must be sure that steps 4 and 5 are idempotent in order to ensure eventual consistency.
- You can scale the solution by using multiple queues and worker role instances.

When to use this pattern

Use this pattern when you want to guarantee eventual consistency between entities that exist in different partitions or tables. You can extend this pattern to ensure eventual consistency for operations across the Table service and the Blob service and other non-Azure Storage data sources such as database or the file system.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Entity Group Transactions](#)
- [Merge or replace](#)

NOTE

If transaction isolation is important to your solution, you should consider redesigning your tables to enable you to use EGTs.

Index Entities Pattern

Maintain index entities to enable efficient searches that return lists of entities.

Context and problem

The Table service automatically indexes entities using the **PartitionKey** and **RowKey** values. This enables a client application to retrieve an entity efficiently using a point query. For example, using the table structure shown below, a client application can efficiently retrieve an individual employee entity by using the department name and the employee id (the **PartitionKey** and **RowKey**).

Employee entity

PartitionKey (Department name)

RowKey (Employee Id)

FirstName (string)

LastName (string)

Age (integer)

EmailAddress (string)

If you also want to be able to retrieve a list of employee entities based on the value of another non-unique property, such as their last name, you must use a less efficient partition scan to find matches rather than using an

index to look them up directly. This is because the table service does not provide secondary indexes.

Solution

To enable lookup by last name with the entity structure shown above, you must maintain lists of employee ids. If you want to retrieve the employee entities with a particular last name, such as Jones, you must first locate the list of employee ids for employees with Jones as their last name, and then retrieve those employee entities. There are three main options for storing the lists of employee ids:

- Use blob storage.
- Create index entities in the same partition as the employee entities.
- Create index entities in a separate partition or table.

Option #1: Use blob storage

For the first option, you create a blob for every unique last name, and in each blob store a list of the **PartitionKey** (department) and **RowKey** (employee id) values for employees that have that last name. When you add or delete an employee you should ensure that the content of the relevant blob is eventually consistent with the employee entities.

Option #2: Create index entities in the same partition

For the second option, use index entities that store the following data:

Employee index entity
PartitionKey (Department name)
RowKey (LastName)
EmployeeIDs (String containing a list of employee ids with same last name)

The **EmployeeIDs** property contains a list of employee ids for employees with the last name stored in the **RowKey**.

The following steps outline the process you should follow when you are adding a new employee if you are using the second option. In this example, we are adding an employee with Id 000152 and a last name Jones in the Sales department:

1. Retrieve the index entity with a **PartitionKey** value "Sales" and the **RowKey** value "Jones." Save the ETag of this entity to use in step 2.
2. Create an entity group transaction (that is, a batch operation) that inserts the new employee entity (**PartitionKey** value "Sales" and **RowKey** value "000152"), and updates the index entity (**PartitionKey** value "Sales" and **RowKey** value "Jones") by adding the new employee id to the list in the EmployeeIDs field. For more information about entity group transactions, see [Entity Group Transactions](#).
3. If the entity group transaction fails because of an optimistic concurrency error (someone else has just modified the index entity), then you need to start over at step 1 again.

You can use a similar approach to deleting an employee if you are using the second option. Changing an employee's last name is slightly more complex because you will need to execute an entity group transaction that updates three entities: the employee entity, the index entity for the old last name, and the index entity for the new last name. You must retrieve each entity before making any changes in order to retrieve the ETag values that you can then use to perform the updates using optimistic concurrency.

The following steps outline the process you should follow when you need to look up all the employees with a given last name in a department if you are using the second option. In this example, we are looking up all the employees with last name Jones in the Sales department:

1. Retrieve the index entity with a **PartitionKey** value "Sales" and the **RowKey** value "Jones."
2. Parse the list of employee IDs in the EmployeeIDs field.
3. If you need additional information about each of these employees (such as their email addresses), retrieve each of the employee entities using **PartitionKey** value "Sales" and **RowKey** values from the list of employees you obtained in step 2.

Option #3: Create index entities in a separate partition or table

For the third option, use index entities that store the following data:

Employee index entity
PartitionKey ("indexentities")
RowKey (LastName)
EmployeeIDs (String containing a list of employee ids with same last name)

The **EmployeeIDs** property contains a list of employee IDs for employees with the last name stored in the **RowKey**.

With the third option, you cannot use EGTs to maintain consistency because the index entities are in a separate partition from the employee entities. You should ensure that the index entities are eventually consistent with the employee entities.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- This solution requires at least two queries to retrieve matching entities: one to query the index entities to obtain the list of **RowKey** values, and then queries to retrieve each entity in the list.
- Given that an individual entity has a maximum size of 1 MB, option #2 and option #3 in the solution assume that the list of employee IDs for any given last name is never greater than 1 MB. If the list of employee IDs is likely to be greater than 1 MB in size, use option #1 and store the index data in blob storage.
- If you use option #2 (using EGTs to handle adding and deleting employees, and changing an employee's last name) you must evaluate if the volume of transactions will approach the scalability limits in a given partition. If this is the case, you should consider an eventually consistent solution (option #1 or option #3) that uses queues to handle the update requests and enables you to store your index entities in a separate partition from the employee entities.
- Option #2 in this solution assumes that you want to look up by last name within a department: for example, you want to retrieve a list of employees with a last name Jones in the Sales department. If you want to be able to look up all the employees with a last name Jones across the whole organization, use either option #1 or option #3.
- You can implement a queue-based solution that delivers eventual consistency (see the [Eventually consistent transactions pattern](#) for more details).

When to use this pattern

Use this pattern when you want to lookup a set of entities that all share a common property value, such as all employees with the last name Jones.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Compound key pattern](#)
- [Eventually consistent transactions pattern](#)
- [Entity Group Transactions](#)

- [Working with heterogeneous entity types](#)

Denormalization pattern

Combine related data together in a single entity to enable you to retrieve all the data you need with a single point query.

Context and problem

In a relational database, you typically normalize data to remove duplication resulting in queries that retrieve data from multiple tables. If you normalize your data in Azure tables, you must make multiple round trips from the client to the server to retrieve your related data. For example, with the table structure shown below you need two round trips to retrieve the details for a department: one to fetch the department entity that includes the manager's id, and then another request to fetch the manager's details in an employee entity.

Department entity	Employee entity
PartitionKey (Department name) RowKey ("Department") DepartmentName (string) EmployeeCount (integer) ManagerId (string - employee id of manager)	PartitionKey (Department name) RowKey (Employee Id) FirstName (string) LastName (string) Age (integer) EmailAddress (string)

Solution

Instead of storing the data in two separate entities, denormalize the data and keep a copy of the manager's details in the department entity. For example:

Department entity
PartitionKey (Department name) RowKey ("Department") DepartmentName (string) EmployeeCount (integer) ManagerName (string) ManagerEmailAddress (string)

With department entities stored with these properties, you can now retrieve all the details you need about a department using a point query.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- There is some cost overhead associated with storing some data twice. The performance benefit (resulting from fewer requests to the storage service) typically outweighs the marginal increase in storage costs (and this cost is partially offset by a reduction in the number of transactions you require to fetch the details of a department).
- You must maintain the consistency of the two entities that store information about managers. You can handle the consistency issue by using EGTs to update multiple entities in a single atomic transaction: in this case, the department entity, and the employee entity for the department manager are stored in the same partition.

When to use this pattern

Use this pattern when you frequently need to look up related information. This pattern reduces the number of queries your client must make to retrieve the data it requires.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Compound key pattern](#)
- [Entity Group Transactions](#)
- [Working with heterogeneous entity types](#)

Compound key pattern

Use compound **RowKey** values to enable a client to lookup related data with a single point query.

Context and problem

In a relational database, it is quite natural to use joins in queries to return related pieces of data to the client in a single query. For example, you might use the employee id to look up a list of related entities that contain performance and review data for that employee.

Assume you are storing employee entities in the Table service using the following structure:

Employee entity
PartitionKey (Department name)
RowKey (Employee Id)

FirstName (string)
LastName (string)
Age (integer)
EmailAddress (string)

You also need to store historical data relating to reviews and performance for each year the employee has worked for your organization and you need to be able to access this information by year. One option is to create another table that stores entities with the following structure:

Employee review entity
PartitionKey (Employee Id)
RowKey (Year)

FirstName (string)
LastName (string)
ManagerRating (integer)
PeerRating (integer)
Comments (string)

Notice that with this approach you may decide to duplicate some information (such as first name and last name) in the new entity to enable you to retrieve your data with a single request. However, you cannot maintain strong consistency because you cannot use an EGT to update the two entities atomically.

Solution

Store a new entity type in your original table using entities with the following structure:

Employee entity
PartitionKey (Department name)
RowKey (Employee Id + Year)

FirstName (string)
LastName (string)
Age (integer)
EmailAddress (string)
ManagerRating (integer)
PeerRating (integer)
Comments (string)

Notice how the **RowKey** is now a compound key made up of the employee id and the year of the review data that enables you to retrieve the employee's performance and review data with a single request for a single entity.

The following example outlines how you can retrieve all the review data for a particular employee (such as employee 000123 in the Sales department):

```
$filter=(PartitionKey eq 'Sales') and (RowKey ge 'empid_000123') and (RowKey lt 'empid_000124')&$select=RowKey,Manager Rating,Peer Rating,Comments
```

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- You should use a suitable separator character that makes it easy to parse the **RowKey** value: for example,

000123_2012.

- You are also storing this entity in the same partition as other entities that contain related data for the same employee, which means you can use EGTs to maintain strong consistency.
- You should consider how frequently you will query the data to determine whether this pattern is appropriate. For example, if you will access the review data infrequently and the main employee data often you should keep them as separate entities.

When to use this pattern

Use this pattern when you need to store one or more related entities that you query frequently.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Entity Group Transactions](#)
- [Working with heterogeneous entity types](#)
- [Eventually consistent transactions pattern](#)

Log tail pattern

Retrieve the *n* entities most recently added to a partition by using a **RowKey** value that sorts in reverse date and time order.

Context and problem

A common requirement is be able to retrieve the most recently created entities, for example the ten most recent expense claims submitted by an employee. Table queries support a **\$top** query operation to return the first *n* entities from a set: there is no equivalent query operation to return the last *n* entities in a set.

Solution

Store the entities using a **RowKey** that naturally sorts in reverse date/time order by using so the most recent entry is always the first one in the table.

For example, to be able to retrieve the ten most recent expense claims submitted by an employee, you can use a reverse tick value derived from the current date/time. The following C# code sample shows one way to create a suitable "inverted ticks" value for a **RowKey** that sorts from the most recent to the oldest:

```
string invertedTicks = string.Format("{0:D19}", DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks);
```

You can get back to the date time value using the following code:

```
DateTime dt = new DateTime(DateTime.MaxValue.Ticks - Int64.Parse(invertedTicks));
```

The table query looks like this:

```
https://myaccount.table.core.windows.net/EmployeeExpense\(PartitionKey='empid'\)?\$top=10
```

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- You must pad the reverse tick value with leading zeroes to ensure the string value sorts as expected.
- You must be aware of the scalability targets at the level of a partition. Be careful not create hot spot partitions.

When to use this pattern

Use this pattern when you need to access entities in reverse date/time order or when you need to access the most recently added entities.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Prepend / append anti-pattern](#)
- [Retrieving entities](#)

High volume delete pattern

Enable the deletion of a high volume of entities by storing all the entities for simultaneous deletion in their own separate table; you delete the entities by deleting the table.

Context and problem

Many applications delete old data which no longer needs to be available to a client application, or that the application has archived to another storage medium. You typically identify such data by a date: for example, you have a requirement to delete records of all login requests that are more than 60 days old.

One possible design is to use the date and time of the login request in the **RowKey**:

Login attempt entity
PartitionKey (Username)
RowKey (Date and time of login attempt)
SourceIPAddress (string)
SuccessfulLogin (boolean)

This approach avoids partition hotspots because the application can insert and delete login entities for each user in a separate partition. However, this approach may be costly and time consuming if you have a large number of entities because first you need to perform a table scan in order to identify all the entities to delete, and then you must delete each old entity. Note that you can reduce the number of round trips to the server required to delete the old entities by batching multiple delete requests into EGTs.

Solution

Use a separate table for each day of login attempts. You can use the entity design above to avoid hotspots when you are inserting entities, and deleting old entities is now simply a question of deleting one table every day (a single storage operation) instead of finding and deleting hundreds and thousands of individual login entities every day.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Does your design support other ways your application will use the data such as looking up specific entities, linking with other data, or generating aggregate information?
- Does your design avoid hot spots when you are inserting new entities?
- Expect a delay if you want to reuse the same table name after deleting it. It's better to always use unique table names.
- Expect some throttling when you first use a new table while the Table service learns the access patterns and distributes the partitions across nodes. You should consider how frequently you need to create new tables.

When to use this pattern

Use this pattern when you have a high volume of entities that you must delete at the same time.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Entity Group Transactions](#)
- [Modifying entities](#)

Data series pattern

Store complete data series in a single entity to minimize the number of requests you make.

Context and problem

A common scenario is for an application to store a series of data that it typically needs to retrieve all at once. For example, your application might record how many IM messages each employee sends every hour, and then use this information to plot how many messages each user sent over the preceding 24 hours. One design might be to store 24 entities for each employee:

Message stats entity

PartitionKey (Employee Id)
RowKey (Hour ("01", "02", "03", ..., "24"))
MessageCount (integer)

With this design, you can easily locate and update the entity to update for each employee whenever the application needs to update the message count value. However, to retrieve the information to plot a chart of the activity for the preceding 24 hours, you must retrieve 24 entities.

Solution

Use the following design with a separate property to store the message count for each hour:

Message stats entity

PartitionKey (Department)
RowKey (Employee Id)
MessageCountHour01 (integer)
MessageCountHour02 (integer)
MessageCountHour03 (integer)
...
MessageCountHour24 (integer)

With this design, you can use a merge operation to update the message count for an employee for a specific hour. Now, you can retrieve all the information you need to plot the chart using a request for a single entity.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- If your complete data series does not fit into a single entity (an entity can have up to 252 properties), use an alternative data store such as a blob.
- If you have multiple clients updating an entity simultaneously, you will need to use the **ETag** to implement optimistic concurrency. If you have many clients, you may experience high contention.

When to use this pattern

Use this pattern when you need to update and retrieve a data series associated with an individual entity.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Large entities pattern](#)
- [Merge or replace](#)
- [Eventually consistent transactions pattern](#) (if you are storing the data series in a blob)

Wide entities pattern

Use multiple physical entities to store logical entities with more than 252 properties.

Context and problem

An individual entity can have no more than 252 properties (excluding the mandatory system properties) and cannot store more than 1 MB of data in total. In a relational database, you would typically get round any limits on the size of a row by adding a new table and enforcing a 1-to-1 relationship between them.

Solution

Using the Table service, you can store multiple entities to represent a single large business object with more than 252 properties. For example, if you want to store a count of the number of IM messages sent by each employee for the last 365 days, you could use the following design that uses two entities with different schemas:

Message stats entity

```
PartitionKey (Department)
RowKey (Employee Id + "_01")
MessageCountDay01 (integer)
MessageCountDay02 (integer)
MessageCountDay03 (integer)
...
MessageCountDay252 (integer)
```

Message stats entity

```
PartitionKey (Department)
RowKey (Employee Id + "_02")
MessageCountDay253 (integer)
MessageCountDay254 (integer)
MessageCountDay255 (integer)
...
MessageCountDay365 (integer)
```

If you need to make a change that requires updating both entities to keep them synchronized with each other you can use an EGT. Otherwise, you can use a single merge operation to update the message count for a specific day. To retrieve all the data for an individual employee you must retrieve both entities, which you can do with two efficient requests that use both a **PartitionKey** and a **RowKey** value.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Retrieving a complete logical entity involves at least two storage transactions: one to retrieve each physical entity.

When to use this pattern

Use this pattern when need to store entities whose size or number of properties exceeds the limits for an individual entity in the Table service.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Entity Group Transactions](#)
- [Merge or replace](#)

Large entities pattern

Use blob storage to store large property values.

Context and problem

An individual entity cannot store more than 1 MB of data in total. If one or several of your properties store values that cause the total size of your entity to exceed this value, you cannot store the entire entity in the Table service.

Solution

If your entity exceeds 1 MB in size because one or more properties contain a large amount of data, you can store data in the Blob service and then store the address of the blob in a property in the entity. For example, you can store the photo of an employee in blob storage and store a link to the photo in the **Photo** property of your employee entity:

Employee entity

```
PartitionKey (Department name)
RowKey (Email address)
FirstName (string)
LastName (string)
Age (integer)
Photo (string)
```

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- To maintain eventual consistency between the entity in the Table service and the data in the Blob service, use the [Eventually consistent transactions pattern](#) to maintain your entities.
- Retrieving a complete entity involves at least two storage transactions: one to retrieve the entity and one to retrieve the blob data.

When to use this pattern

Use this pattern when you need to store entities whose size exceeds the limits for an individual entity in the Table service.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

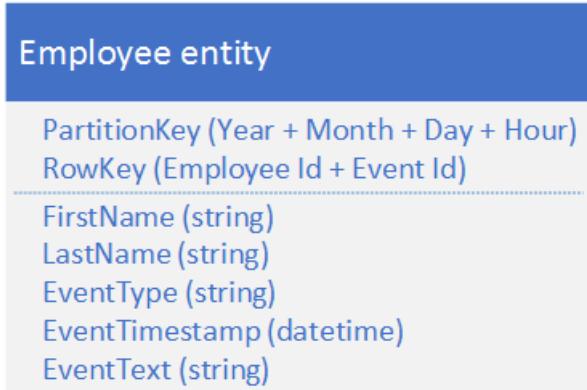
- [Eventually consistent transactions pattern](#)
- [Wide entities pattern](#)

Prepend/append anti-pattern

Increase scalability when you have a high volume of inserts by spreading the inserts across multiple partitions.

Context and problem

Prepending or appending entities to your stored entities typically results in the application adding new entities to the first or last partition of a sequence of partitions. In this case, all of the inserts at any given time are taking place in the same partition, creating a hotspot that prevents the table service from load balancing inserts across multiple nodes, and possibly causing your application to hit the scalability targets for partition. For example, if you have an application that logs network and resource access by employees, then an entity structure as shown below could result in the current hour's partition becoming a hotspot if the volume of transactions reaches the scalability target for an individual partition:



Solution

The following alternative entity structure avoids a hotspot on any particular partition as the application logs events:



Notice with this example how both the **PartitionKey** and **RowKey** are compound keys. The **PartitionKey** uses both the department and employee id to distribute the logging across multiple partitions.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Does the alternative key structure that avoids creating hot partitions on inserts efficiently support the queries your client application makes?
- Does your anticipated volume of transactions mean that you are likely to reach the scalability targets for an individual partition and be throttled by the storage service?

When to use this pattern

Avoid the prepend/append anti-pattern when your volume of transactions is likely to result in throttling by the storage service when you access a hot partition.

Related patterns and guidance

The following patterns and guidance may also be relevant when implementing this pattern:

- [Compound key pattern](#)
- [Log tail pattern](#)
- [Modifying entities](#)

Log data anti-pattern

Typically, you should use the Blob service instead of the Table service to store log data.

Context and problem

A common use case for log data is to retrieve a selection of log entries for a specific date/time range: for example, you want to find all the error and critical messages that your application logged between 15:04 and 15:06 on a specific date. You do not want to use the date and time of the log message to determine the partition you save log entities to: that results in a hot partition because at any given time, all the log entities will share the same

PartitionKey value (see the section [Prepend/append anti-pattern](#)). For example, the following entity schema for a log message results in a hot partition because the application writes all log messages to the partition for the current date and hour:

Log message entity
PartitionKey (Date and hour of log message)
RowKey (Time of log message and unique log message id (GUID))
Severity (integer)
Source (string)
Message (string)
Department (string)
ClientIP (string)

In this example, the **RowKey** includes the date and time of the log message to ensure that log messages are stored sorted in date/time order, and includes a message id in case multiple log messages share the same date and time.

Another approach is to use a **PartitionKey** that ensures that the application writes messages across a range of partitions. For example, if the source of the log message provides a way to distribute messages across many partitions, you could use the following entity schema:

Log message entity
PartitionKey (Source)
RowKey (Time of log message and unique log message id (GUID))
Severity (integer)
Message (string)
Department (string)
ClientIP (string)

However, the problem with this schema is that to retrieve all the log messages for a specific time span you must search every partition in the table.

Solution

The previous section highlighted the problem of trying to use the Table service to store log entries and suggested two, unsatisfactory, designs. One solution led to a hot partition with the risk of poor performance writing log messages; the other solution resulted in poor query performance because of the requirement to scan every partition in the table to retrieve log messages for a specific time span. Blob storage offers a better solution for this type of scenario and this is how Azure Storage Analytics stores the log data it collects.

This section outlines how Storage Analytics stores log data in blob storage as an illustration of this approach to storing data that you typically query by range.

Storage Analytics stores log messages in a delimited format in multiple blobs. The delimited format makes it easy for a client application to parse the data in the log message.

Storage Analytics uses a naming convention for blobs that enables you to locate the blob (or blobs) that contain the log messages for which you are searching. For example, a blob named "queue/2014/07/31/1800/000001.log" contains log messages that relate to the queue service for the hour starting at 18:00 on 31 July 2014. The "000001" indicates that this is the first log file for this period. Storage Analytics also records the timestamps of the first and last log messages stored in the file as part of the blob's metadata. The API for blob storage enables you locate blobs in a container based on a name prefix: to locate all the blobs that contain queue log data for the hour starting at 18:00, you can use the prefix "queue/2014/07/31/1800."

Storage Analytics buffers log messages internally and then periodically updates the appropriate blob or creates a new one with the latest batch of log entries. This reduces the number of writes it must perform to the blob service.

If you are implementing a similar solution in your own application, you must consider how to manage the trade-off between reliability (writing every log entry to blob storage as it happens) and cost and scalability (buffering updates in your application and writing them to blob storage in batches).

Issues and considerations

Consider the following points when deciding how to store log data:

- If you create a table design that avoids potential hot partitions, you may find that you cannot access your log data efficiently.
- To process log data, a client often needs to load many records.
- Although log data is often structured, blob storage may be a better solution.

Implementation considerations

This section discusses some of the considerations to bear in mind when you implement the patterns described in the previous sections. Most of this section uses examples written in C# that use the Storage Client Library (version 4.3.0 at the time of writing).

Retrieving entities

As discussed in the section [Design for querying](#), the most efficient query is a point query. However, in some scenarios you may need to retrieve multiple entities. This section describes some common approaches to retrieving entities using the Storage Client Library.

Executing a point query using the Storage Client Library

The easiest way to execute a point query is to use the **Retrieve** table operation as shown in the following C# code snippet that retrieves an entity with a **PartitionKey** of value "Sales" and a **RowKey** of value "212":

```
TableOperation retrieveOperation = TableOperation.Retrieve<EmployeeEntity>("Sales", "212");
var retrieveResult = employeeTable.Execute(retrieveOperation);
if (retrieveResult.Result != null)
{
    EmployeeEntity employee = (EmployeeEntity)retrieveResult.Result;
    ...
}
```

Notice how this example expects the entity it retrieves to be of type **EmployeeEntity**.

Retrieving multiple entities using LINQ

You can retrieve multiple entities by using LINQ with Storage Client Library and specifying a query with a **where** clause. To avoid a table scan, you should always include the **PartitionKey** value in the where clause, and if possible the **RowKey** value to avoid table and partition scans. The table service supports a limited set of comparison operators (greater than, greater than or equal, less than, less than or equal, equal, and not equal) to use in the where clause. The following C# code snippet finds all the employees whose last name starts with "B" (assuming that

the **RowKey** stores the last name) in the sales department (assuming the **PartitionKey** stores the department name):

```
TableQuery<EmployeeEntity> employeeQuery = employeeTable.CreateQuery<EmployeeEntity>();
var query = (from employee in employeeQuery
            where employee.PartitionKey == "Sales" &&
            employee.RowKey.CompareTo("B") >= 0 &&
            employee.RowKey.CompareTo("C") < 0
            select employee).AsTableQuery();
var employees = query.Execute();
```

Notice how the query specifies both a **RowKey** and a **PartitionKey** to ensure better performance.

The following code sample shows equivalent functionality using the fluent API (for more information about fluent APIs in general, see [Best Practices for Designing a Fluent API](#)):

```
TableQuery<EmployeeEntity> employeeQuery = new TableQuery<EmployeeEntity>().Where(
    TableQuery.CombineFilters(
        TableQuery.CombineFilters(
            TableQuery.GenerateFilterCondition(
                "PartitionKey", QueryComparisons.Equal, "Sales"),
            TableOperators.And,
            TableQuery.GenerateFilterCondition(
                "RowKey", QueryComparisons.GreaterThanOrEqualTo, "B")
        ),
        TableOperators.And,
        TableQuery.GenerateFilterCondition("RowKey", QueryComparisons.LessThan, "C")
    )
);
var employees = employeeTable.ExecuteQuery(employeeQuery);
```

NOTE

The sample nests multiple **CombineFilters** methods to include the three filter conditions.

Retrieving large numbers of entities from a query

An optimal query returns an individual entity based on a **PartitionKey** value and a **RowKey** value. However, in some scenarios you may have a requirement to return many entities from the same partition or even from many partitions.

You should always fully test the performance of your application in such scenarios.

A query against the table service may return a maximum of 1,000 entities at one time and may execute for a maximum of five seconds. If the result set contains more than 1,000 entities, if the query did not complete within five seconds, or if the query crosses the partition boundary, the Table service returns a continuation token to enable the client application to request the next set of entities. For more information about how continuation tokens work, see [Query Timeout and Pagination](#).

If you are using the Storage Client Library, it can automatically handle continuation tokens for you as it returns entities from the Table service. The following C# code sample using the Storage Client Library automatically handles continuation tokens if the table service returns them in a response:

```

string filter = TableQuery.GenerateFilterCondition(
    "PartitionKey", QueryComparisons.Equal, "Sales");
TableQuery<EmployeeEntity> employeeQuery =
    new TableQuery<EmployeeEntity>().Where(filter);

var employees = employeeTable.ExecuteQuery(employeeQuery);
foreach (var emp in employees)
{
    ...
}

```

The following C# code handles continuation tokens explicitly:

```

string filter = TableQuery.GenerateFilterCondition(
    "PartitionKey", QueryComparisons.Equal, "Sales");
TableQuery<EmployeeEntity> employeeQuery =
    new TableQuery<EmployeeEntity>().Where(filter);

TableContinuationToken continuationToken = null;

do
{
    var employees = employeeTable.ExecuteQuerySegmented(
        employeeQuery, continuationToken);
    foreach (var emp in employees)
    {
        ...
    }
    continuationToken = employees.ContinuationToken;
} while (continuationToken != null);

```

By using continuation tokens explicitly, you can control when your application retrieves the next segment of data. For example, if your client application enables users to page through the entities stored in a table, a user may decide not to page through all the entities retrieved by the query so your application would only use a continuation token to retrieve the next segment when the user had finished paging through all the entities in the current segment. This approach has several benefits:

- It enables you to limit the amount of data to retrieve from the Table service and that you move over the network.
- It enables you to perform asynchronous IO in .NET.
- It enables you to serialize the continuation token to persistent storage so you can continue in the event of an application crash.

NOTE

A continuation token typically returns a segment containing 1,000 entities, although it may be fewer. This is also the case if you limit the number of entries a query returns by using **Take** to return the first n entities that match your lookup criteria: the table service may return a segment containing fewer than n entities along with a continuation token to enable you to retrieve the remaining entities.

The following C# code shows how to modify the number of entities returned inside a segment:

```
employeeQuery.TakeCount = 50;
```

Server-side projection

A single entity can have up to 255 properties and be up to 1 MB in size. When you query the table and retrieve entities, you may not need all the properties and can avoid transferring data unnecessarily (to help reduce latency and cost). You can use server-side projection to transfer just the properties you need. The following example is

retrieves just the **Email** property (along with **PartitionKey**, **RowKey**, **Timestamp**, and **ETag**) from the entities selected by the query.

```
string filter = TableQuery.GenerateFilterCondition(
    "PartitionKey", QueryComparisons.Equal, "Sales");
List<string> columns = new List<string>() { "Email" };
TableQuery<EmployeeEntity> employeeQuery =
    new TableQuery<EmployeeEntity>().Where(filter).Select(columns);

var entities = employeeTable.ExecuteQuery(employeeQuery);
foreach (var e in entities)
{
    Console.WriteLine("RowKey: {0}, EmployeeEmail: {1}", e.RowKey, e.Email);
}
```

Notice how the **RowKey** value is available even though it was not included in the list of properties to retrieve.

Modifying entities

The Storage Client Library enables you to modify your entities stored in the table service by inserting, deleting, and updating entities. You can use EGTs to batch multiple insert, update, and delete operations together to reduce the number of round trips required and improve the performance of your solution.

Note that exceptions thrown when the Storage Client Library executes an EGT typically include the index of the entity that caused the batch to fail. This is helpful when you are debugging code that uses EGTs.

You should also consider how your design affects how your client application handles concurrency and update operations.

Managing concurrency

By default, the table service implements optimistic concurrency checks at the level of individual entities for **Insert**, **Merge**, and **Delete** operations, although it is possible for a client to force the table service to bypass these checks. For more information about how the table service manages concurrency, see [Managing Concurrency in Microsoft Azure Storage](#).

Merge or replace

The **Replace** method of the **TableOperation** class always replaces the complete entity in the Table service. If you do not include a property in the request when that property exists in the stored entity, the request removes that property from the stored entity. Unless you want to remove a property explicitly from a stored entity, you must include every property in the request.

You can use the **Merge** method of the **TableOperation** class to reduce the amount of data that you send to the Table service when you want to update an entity. The **Merge** method replaces any properties in the stored entity with property values from the entity included in the request, but leaves intact any properties in the stored entity that are not included in the request. This is useful if you have large entities and only need to update a small number of properties in a request.

NOTE

The **Replace** and **Merge** methods fail if the entity does not exist. As an alternative, you can use the **InsertOrReplace** and **InsertOrMerge** methods that create a new entity if it doesn't exist.

Working with heterogeneous entity types

The Table service is a *schema-less* table store that means that a single table can store entities of multiple types providing great flexibility in your design. The following example illustrates a table storing both employee and department entities:

PARTITIONKEY	ROWKEY	TIMESTAMP
--------------	--------	-----------

			<table border="1"> <tr><td>FIRSTNAME</td><td>LASTNAME</td><td>AGE</td><td>EMAIL</td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>	FIRSTNAME	LASTNAME	AGE	EMAIL				
FIRSTNAME	LASTNAME	AGE	EMAIL								
			<table border="1"> <tr><td>FIRSTNAME</td><td>LASTNAME</td><td>AGE</td><td>EMAIL</td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>	FIRSTNAME	LASTNAME	AGE	EMAIL				
FIRSTNAME	LASTNAME	AGE	EMAIL								
			<table border="1"> <tr><td>DEPARTMENTNAME</td><td>EMPLOYEECOUNT</td></tr> <tr><td></td><td></td></tr> </table>	DEPARTMENTNAME	EMPLOYEECOUNT						
DEPARTMENTNAME	EMPLOYEECOUNT										
			<table border="1"> <tr><td>FIRSTNAME</td><td>LASTNAME</td><td>AGE</td><td>EMAIL</td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>	FIRSTNAME	LASTNAME	AGE	EMAIL				
FIRSTNAME	LASTNAME	AGE	EMAIL								

Note that each entity must still have **PartitionKey**, **RowKey**, and **Timestamp** values, but may have any set of properties. Furthermore, there is nothing to indicate the type of an entity unless you choose to store that information somewhere. There are two options for identifying the entity type:

- Prepend the entity type to the **RowKey** (or possibly the **PartitionKey**). For example, **EMPLOYEE_000123** or **DEPARTMENT_SALES** as **RowKey** values.
- Use a separate property to record the entity type as shown in the table below.

PARTITIONKEY	ROWKEY	TIMESTAMP					
			ENTITYPE	FIRSTNAME	LASTNAME	AGE	EMAIL
			Employee				

The first option, prepending the entity type to the **RowKey**, is useful if there is a possibility that two entities of different types might have the same key value. It also groups entities of the same type together in the partition.

The techniques discussed in this section are especially relevant to the discussion [Inheritance relationships](#) earlier in this guide in the section [Modelling relationships](#).

NOTE

You should consider including a version number in the entity type value to enable client applications to evolve POCO objects and work with different versions.

The remainder of this section describes some of the features in the Storage Client Library that facilitate working with multiple entity types in the same table.

Retrieving heterogeneous entity types

If you are using the Storage Client Library, you have three options for working with multiple entity types.

If you know the type of the entity stored with a specific **RowKey** and **PartitionKey** values, then you can specify the entity type when you retrieve the entity as shown in the previous two examples that retrieve entities of type

[EmployeeEntity: Executing a point query using the Storage Client Library](#) and [Retrieving multiple entities using LINQ](#).

The second option is to use the **DynamicTableEntity** type (a property bag) instead of a concrete POCO entity type (this option may also improve performance because there is no need to serialize and deserialize the entity to .NET types). The following C# code potentially retrieves multiple entities of different types from the table, but returns all entities as **DynamicTableEntity** instances. It then uses the **EntityType** property to determine the type of each entity:

```
string filter = TableQuery.CombineFilters(
    TableQuery.GenerateFilterCondition("PartitionKey",
        QueryComparisons.Equal, "Sales"),
    TableOperators.And,
    TableQuery.CombineFilters(
        TableQuery.GenerateFilterCondition("RowKey",
            QueryComparisons.GreaterThanOrEqualTo, "B"),
        TableOperators.And,
        TableQuery.GenerateFilterCondition("RowKey",
            QueryComparisons.LessThan, "F")
    )
);
TableQuery<DynamicTableEntity> entityQuery =
    new TableQuery<DynamicTableEntity>().Where(filter);
var employees = employeeTable.ExecuteQuery(entityQuery);

IEnumerable<DynamicTableEntity> entities = employeeTable.ExecuteQuery(entityQuery);
foreach (var e in entities)
{
    EntityProperty entityTypeProperty;
    if (e.Properties.TryGetValue("EntityType", out entityTypeProperty))
    {
        if (entityTypeProperty.StringValue == "Employee")
        {
            // Use entityTypeProperty, RowKey, PartitionKey, Etag, and Timestamp
        }
    }
}
```

Note that to retrieve other properties you must use the **TryGetValue** method on the **Properties** property of the **DynamicTableEntity** class.

A third option is to combine using the **DynamicTableEntity** type and an **EntityResolver** instance. This enables you to resolve to multiple POCO types in the same query. In this example, the **EntityResolver** delegate is using the **EntityType** property to distinguish between the two types of entity that the query returns. The **Resolve** method uses the **resolver** delegate to resolve **DynamicTableEntity** instances to **TableEntity** instances.

```

EntityResolver<TableEntity> resolver = (pk, rk, ts, props, etag) =>
{
    TableEntity resolvedEntity = null;
    if (props["EntityType"].StringValue == "Department")
    {
        resolvedEntity = new DepartmentEntity();
    }
    else if (props["EntityType"].StringValue == "Employee")
    {
        resolvedEntity = new EmployeeEntity();
    }
    else throw new ArgumentException("Unrecognized entity", "props");

    resolvedEntity.PartitionKey = pk;
    resolvedEntity.RowKey = rk;
    resolvedEntity.Timestamp = ts;
    resolvedEntity.ETag = etag;
    resolvedEntity.ReadEntity(props, null);
    return resolvedEntity;
};

string filter = TableQuery.GenerateFilterCondition(
    "PartitionKey", QueryComparisons.Equal, "Sales");
TableQuery<DynamicTableEntity> entityQuery =
    new TableQuery<DynamicTableEntity>().Where(filter);

var entities = employeeTable.ExecuteQuery(entityQuery, resolver);
foreach (var e in entities)
{
    if (e is DepartmentEntity)
    {
        ...
    }
    if (e is EmployeeEntity)
    {
        ...
    }
}

```

Modifying heterogeneous entity types

You do not need to know the type of an entity to delete it, and you always know the type of an entity when you insert it. However, you can use **DynamicTableEntity** type to update an entity without knowing its type and without using a POCO entity class. The following code sample retrieves a single entity, and checks the **EmployeeCount** property exists before updating it.

```

TableResult result =
    employeeTable.Execute(TableOperation.Retrieve(partitionKey, rowKey));
DynamicTableEntity department = (DynamicTableEntity)result.Result;

EntityProperty countProperty;

if (!department.Properties.TryGetValue("EmployeeCount", out countProperty))
{
    throw new
        InvalidOperationException("Invalid entity, EmployeeCount property not found.");
}
countProperty.Int32Value += 1;
employeeTable.Execute(TableOperation.Merge(department));

```

Controlling access with Shared Access Signatures

You can use Shared Access Signature (SAS) tokens to enable client applications to modify (and query) table entities directly without the need to authenticate directly with the table service. Typically, there are three main benefits to

using SAS in your application:

- You do not need to distribute your storage account key to an insecure platform (such as a mobile device) in order to allow that device to access and modify entities in the Table service.
- You can offload some of the work that web and worker roles perform in managing your entities to client devices such as end-user computers and mobile devices.
- You can assign a constrained and time limited set of permissions to a client (such as allowing read-only access to specific resources).

For more information about using SAS tokens with the Table service, see [Using Shared Access Signatures \(SAS\)](#).

However, you must still generate the SAS tokens that grant a client application to the entities in the table service: you should do this in an environment that has secure access to your storage account keys. Typically, you use a web or worker role to generate the SAS tokens and deliver them to the client applications that need access to your entities. Because there is still an overhead involved in generating and delivering SAS tokens to clients, you should consider how best to reduce this overhead, especially in high-volume scenarios.

It is possible to generate a SAS token that grants access to a subset of the entities in a table. By default, you create a SAS token for an entire table, but it is also possible to specify that the SAS token grant access to either a range of **PartitionKey** values, or a range of **PartitionKey** and **RowKey** values. You might choose to generate SAS tokens for individual users of your system such that each user's SAS token only allows them access to their own entities in the table service.

Asynchronous and parallel operations

Provided you are spreading your requests across multiple partitions, you can improve throughput and client responsiveness by using asynchronous or parallel queries. For example, you might have two or more worker role instances accessing your tables in parallel. You could have individual worker roles responsible for particular sets of partitions, or simply have multiple worker role instances, each able to access all the partitions in a table.

Within a client instance, you can improve throughput by executing storage operations asynchronously. The Storage Client Library makes it easy to write asynchronous queries and modifications. For example, you might start with the synchronous method that retrieves all the entities in a partition as shown in the following C# code:

```
private static void ManyEntitiesQuery(CloudTable employeeTable, string department)
{
    string filter = TableQuery.GenerateFilterCondition(
        "PartitionKey", QueryComparisons.Equal, department);
    TableQuery<EmployeeEntity> employeeQuery =
        new TableQuery<EmployeeEntity>().Where(filter);

    TableContinuationToken continuationToken = null;

    do
    {
        var employees = employeeTable.ExecuteQuerySegmented(
            employeeQuery, continuationToken);
        foreach (var emp in employees)
        {
            ...
        }
        continuationToken = employees.ContinuationToken;
    } while (continuationToken != null);
}
```

You can easily modify this code so that the query runs asynchronously as follows:

```

private static async Task ManyEntitiesQueryAsync(CloudTable employeeTable, string department)
{
    string filter = TableQuery.GenerateFilterCondition(
        "PartitionKey", QueryComparisons.Equal, department);
    TableQuery<EmployeeEntity> employeeQuery =
        new TableQuery<EmployeeEntity>().Where(filter);
    TableContinuationToken continuationToken = null;

    do
    {
        var employees = await employeeTable.ExecuteQuerySegmentedAsync(
            employeeQuery, continuationToken);
        foreach (var emp in employees)
        {
            ...
        }
        continuationToken = employees.ContinuationToken;
    } while (continuationToken != null);
}

```

In this asynchronous example, you can see the following changes from the synchronous version:

- The method signature now includes the **async** modifier and returns a **Task** instance.
- Instead of calling the **ExecuteSegmented** method to retrieve results, the method now calls the **ExecuteSegmentedAsync** method and uses the **await** modifier to retrieve results asynchronously.

The client application can call this method multiple times (with different values for the **department** parameter), and each query will run on a separate thread.

Note that there is no asynchronous version of the **Execute** method in the **TableQuery** class because the **IEnumerable** interface does not support asynchronous enumeration.

You can also insert, update, and delete entities asynchronously. The following C# example shows a simple, synchronous method to insert or replace an employee entity:

```

private static void SimpleEmployeeUpsert(CloudTable employeeTable,
    EmployeeEntity employee)
{
    TableResult result = employeeTable
        .Execute(TableOperation.InsertOrReplace(employee));
    Console.WriteLine("HTTP Status: {0}", result.HttpStatusCode);
}

```

You can easily modify this code so that the update runs asynchronously as follows:

```

private static async Task SimpleEmployeeUpsertAsync(CloudTable employeeTable,
    EmployeeEntity employee)
{
    TableResult result = await employeeTable
        .ExecuteAsync(TableOperation.InsertOrReplace(employee));
    Console.WriteLine("HTTP Status: {0}", result.HttpStatusCode);
}

```

In this asynchronous example, you can see the following changes from the synchronous version:

- The method signature now includes the **async** modifier and returns a **Task** instance.
- Instead of calling the **Execute** method to update the entity, the method now calls the **ExecuteAsync** method and uses the **await** modifier to retrieve results asynchronously.

The client application can call multiple asynchronous methods like this one, and each method invocation will run on a separate thread.

Credits

We would like to thank the following members of the Azure team for their contributions: Dominic Betts, Jason Hogg, Jean Ghanem, Jai Haridas, Jeff Irwin, Vamshidhar Kommineni, Vinay Shah and Serdar Ozler as well as Tom Hollander from Microsoft DX.

We would also like to thank the following Microsoft MVP's for their valuable feedback during review cycles: Igor Papirof and Edward Bakker.

Troubleshooting Azure File storage problems

1/17/2017 • 9 min to read • [Edit on GitHub](#)

This article lists common problems that are related to Microsoft Azure File storage when you connect from Windows and Linux clients. It also provides the possible causes of and resolutions for these problems.

General problems (occur in both Windows and Linux clients)

- [Quota error when trying to open a file](#)
- [Slow performance when you access Azure File storage from Windows or from Linux](#)

Windows client problems

- [Slow performance when you access Azure File storage from Windows 8.1 or Windows Server 2012 R2](#)
- [Error 53 attempting to mount an Azure File Share](#)
- [Net use was successful but I don't see the Azure file share mounted in Windows Explorer](#)
- [My storage account contains "/" and the net use command fails](#)
- [My application/service cannot access mounted Azure Files drive.](#)
- [Additional recommendations to optimize performance](#)

Linux client problems

- [Error "You are copying a file to a destination that does not support encryption" when uploading/copying files to Azure Files](#)
- ["Host is down" error on existing file shares, or the shell hangs when doing list commands on the mount point](#)
- [Mount error 115 when attempting to mount Azure Files on the Linux VM](#)
- [Linux VM experiencing random delays in commands like "ls"](#)

Quota error when trying to open a file

In Windows, you receive error messages that resemble the following:

1816 ERROR_NOT_ENOUGH_QUOTA <--> 0xc0000044

STATUS_QUOTA_EXCEEDED

Not enough quota is available to process this command

Invalid handle value GetLastError: 53

On Linux, you receive error messages that resemble the following:

[permission denied]

Disk quota exceeded

Cause

The problem occurs because you have reached the upper limit of concurrent open handles that are allowed for a file.

Solution

Reduce the number of concurrent open handles by closing some handles, and then retry. For more information, see [Microsoft Azure Storage Performance and Scalability Checklist](#).

Slow performance when accessing File storage from Windows or Linux

- If you don't have a specific minimum I/O size requirement, we recommend that you use 1 MB as the I/O size for optimal performance.
- If you know the final size of a file that you are extending with writes, and your software doesn't have compatibility issues when the not yet written tail on the file containing zeros, then set the file size in advance instead of every write being an extending write.

Slow performance when accessing the File storage from Windows 8.1 or Windows Server 2012 R2

For clients who are running Windows 8.1 or Windows Server 2012 R2, make sure that the hotfix [KB3114025](#) is installed. This hotfix improves the create and close handle performance.

You can run the following script to check whether the hotfix has been installed on:

```
reg query HKLM\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\Parameters\Policies
```

If hotfix is installed, the following output is displayed:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\Parameters\Policies

{96c345ef-3cac-477b-8fcd-bea1a564241c} REG_DWORD 0x1

NOTE

Windows Server 2012 R2 images in Azure Marketplace have the hotfix KB3114025 installed by default starting in December 2015.

Additional recommendations to optimize performance

Never create or open a file for cached I/O that is requesting write access but not read access. That is, when you call **CreateFile()**, never specify only **GENERIC_WRITE**, but always specify **GENERIC_READ | GENERIC_WRITE**. A write-only handle cannot cache small writes locally, even when it is the only open handle for the file. This imposes a severe performance penalty on small writes. Note that the "a" mode to CRT **fopen()** opens a write-only handle.

"Error 53" or "Error 67" when you try to mount or unmount an Azure File Share

This problem can be caused by following conditions:

Cause 1

"System error 53 has occurred. Access is denied." For security reasons, connections to Azure Files shares are blocked if the communication channel isn't encrypted and the connection attempt is not made from the same data center on which Azure File shares reside. Communication channel encryption is not provided if the user's client OS doesn't support SMB encryption. This is indicated by a "System error 53 has occurred. Access is denied" Error message when a user tries to mount a file share from on-premises or from a different data center. Windows 8, Windows Server 2012, and later versions of each negotiate request that includes SMB 3.0, which supports encryption.

Solution for Cause 1

Connect from a client that meets the requirements of Windows 8, Windows Server 2012 or later versions, or that connect from a virtual machine that is on the same data center as the Azure Storage account that is used for the Azure File share.

Cause 2

"System Error 53" or "System Error 67" when you mount an Azure file share can occur if Port 445 outbound communication to Azure Files data center is blocked. Click [here](#) to see the summary of ISPs that allow or disallow access from port 445.

Comcast and some IT organizations block this port. To understand whether this is the reason behind the "System Error 53" message, you can use Portqry to query the TCP:445 endpoint. If the TCP:445 endpoint is displayed as filtered, the TCP port is blocked. Here is an example query:

```
g:\DataDump\Tools>Portqry.exe -n [storage account name].file.core.windows.net -p TCP -e 445
```

If the TCP 445 being blocked by a rule along the network path, you will see the following output:

TCP port 445 (microsoft-ds service): FILTERED

For more information on using Portqry, see [Description of the Portqry.exe command-line utility](#).

Solution for Cause 2

Work with your IT organization to open Port 445 outbound to [Azure IP ranges](#).

Cause 3

"System Error 53" can also be received if NTLMv1 communication is enabled on the client. Having NTLMv1 enabled creates a less-secure client. Therefore, communication will be blocked for Azure Files. To verify whether this is the cause of the error, verify that the following registry subkey is set to a value of 3:

HKLM\SYSTEM\CurrentControlSet\Control\Lsa > LmCompatibilityLevel.

For more information, see the [LmCompatibilityLevel](#) topic on TechNet.

Solution for Cause 3

To resolve this issue, revert the LmCompatibilityLevel value in the HKLM\SYSTEM\CurrentControlSet\Control\Lsa registry key to the default value of 3.

Azure Files supports only NTLMv2 authentication. Make sure that Group Policy is applied to the clients. This will prevent this error from occurring. This is also considered to be a security best practice. For more information, see [how to configure clients to use NTLMv2 using Group Policy](#)

The recommended policy setting is **Send NTLMv2 response only**. This corresponds to a registry value of 3. Clients use only NTLMv2 authentication, and they use NTLMv2 session security if the server supports it. Domain controllers accept LM, NTLM, and NTLMv2 authentication.

Net use was successful but don't see the Azure file share mounted in Windows Explorer

Cause

By default, Windows Explorer does not run as Administrator. If you run **net use** from an Administrator command prompt, you map the network drive "As Administrator." Because mapped drives are user-centric, the user account that is logged in does not display the drives if they are mounted under a different user account.

Solution

Mount the share from a non-administrator command line. Alternatively, you can follow [this TechNet topic](#) to configure the **EnableLinkedConnections** registry value.

My storage account contains "/" and the net use command fails

Cause

When the **net use** command is run under Command Prompt (cmd.exe), it's parsed by adding "/" as a command-line option. This causes the drive mapping to fail.

Solution

You can use either of the following steps to work around the issue:

- Use the following PowerShell command:

```
New-SmbMapping -LocalPath y: -RemotePath \\server\share -UserName accountName -Password "password can contain / and \ etc"
```

From a batch file this can be done as

```
Echo new-smbMapping ... | powershell -command -
```

- Put double quotation marks around the key to work around this issue — unless "/" is the first character. If it is, either use the interactive mode and enter your password separately or regenerate your keys to get a key that doesn't start with the forward slash (/) character.

My application/service cannot access mounted Azure Files drive

Cause

Drives are mounted per user. If your application or service is running under a different user account, users won't see the drive.

Solution

Mount drive from the same user account under which the application is. This can be done using tools such as psexec.

Alternatively, you can create a new user that has the same privileges as the network service or system account, and then run **cmdkey** and **net use** under that account. The user name should be the storage account name, and password should be the storage account key. Another option for **net use** is to pass in the storage account name and key in the user name and password parameters of the **net use** command.

After you follow these instructions, you may receive the following error message: "System error 1312 has occurred. A specified logon session does not exist. It may already have been terminated" when you run **net use** for the system/network service account. If this occurs, make sure that the username that is passed to **net use** includes domain information (for example: "[storage account name].file.core.windows.net").

Error "You are copying a file to a destination that does not support encryption"

Cause

Bitlocker-encrypted files can be copied to Azure Files. However, the File storage does not support NTFS EFS. Therefore, you are likely using EFS in this case. If you have files that are encrypted through EFS, a copy operation to the File storage can fail unless the copy command is decrypting a copied file.

Workaround

To copy a file to the File storage, you must first decrypt it. You can do this by using one of the following methods:

- Use **copy /d**.
- Set the following registry key:
 - Path=HKLM\Software\Policies\Microsoft\Windows\System
 - Value type=DWORD
 - Name = CopyFileAllowDecryptedRemoteDestination

- Value = 1

However, note that setting the registry key affects all copy operations to network shares.

"Host is down" error on existing file shares, or the shell hangs when you run list commands on the mount point

Cause

This error occurs on the Linux client when the client has been idle for an extended period of time. When this error occurs, the client disconnects, and the client connection times out.

Solution

This issue is now fixed in the Linux kernel as part of [change set](#), pending backport into Linux distribution.

To work around this issue, sustain the connection and avoid getting into an idle state, keep a file in the Azure File share that you write to periodically. This has to be a write operation, such as rewriting the created/modified date on the file. Otherwise, you might get cached results, and your operation might not trigger the connection.

"Mount error 115" when you try to mount Azure Files on the Linux VM

Cause

Linux distributions do not yet support encryption feature in SMB 3.0. In some distributions, user may receive a "115" error message if they try to mount Azure Files by using SMB 3.0 because of a missing feature.

Solution

If the Linux SMB client that is used does not support encryption, mount Azure Files by using SMB 2.1 from a Linux VM on the same data center as the File storage account.

Linux VM experiencing random delays in commands like "ls"

Cause

This can occur when the mount command does not include the **serverino** option. Without **serverino**, the ls command runs a **stat** on every file.

Solution

Check the **serverino** in your "/etc/fstab" entry:

```
//azureuser.file.core.windows.net/wms/comer on /home/sampledir type cifs  
(rw,nodev,relatime,vers=2.1,sec=ntlmssp,cache=strict,username=xxx, domain=X,  
file_mode=0755,dir_mode=0755,serverino,rsize=65536,wsize=65536,actimeo=1)
```

If the **serverino** option is not present, unmount and mount Azure Files again by having the **serverino** option selected.

Learn more

- [Get started with Azure File storage on Windows](#)
- [Get started with Azure File storage on Linux](#)

Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads

1/17/2017 • 28 min to read • [Edit on GitHub](#)

Overview

Azure Premium Storage delivers high-performance, low-latency disk support for virtual machines running I/O-intensive workloads. Virtual machine (VM) disks that use Premium Storage store data on solid state drives (SSDs). You can migrate your application's VM disks to Azure Premium Storage to take advantage of the speed and performance of these disks.

An Azure VM supports attaching several Premium Storage disks, so that your applications can have up to 64 TB of storage per VM. With Premium Storage, your applications can achieve 80,000 IOPS (input/output operations per second) per VM and 2000 MB per second disk throughput per VM with extremely low latencies for read operations.

With Premium Storage, Azure offers the ability to truly lift-and-shift your demanding enterprise applications like, Dynamics AX, Dynamics CRM, Exchange Server, SharePoint Farms, and SAP Business Suite, to the cloud. You can run a variety of performance intensive database workloads like SQL Server, Oracle, MongoDB, MySQL, Redis, that require consistent high performance and low latency on Premium Storage.

NOTE

We recommend migrating any virtual machine disk requiring high IOPS to Azure Premium Storage for the best performance for your application. If your disk does not require high IOPS, you can limit costs by maintaining it in Standard Storage, which stores virtual machine disk data on Hard Disk Drives (HDDs) instead of SSDs.

To get started with Azure Premium Storage, visit [Get started for free](#) page. For information on migrating your existing virtual machines to Premium Storage, see [Migrating to Azure Premium Storage](#).

NOTE

Premium Storage is currently supported in some regions. You can find the list of available regions in [Azure Services by Region](#).

Premium Storage Features

Premium Storage Disks: Azure Premium Storage supports VM disks that can be attached to Premium Storage supported Azure VMs (DS, DSv2, GS, or Fs series). When using Premium Storage you have a choice of three disk sizes namely, P10 (128GiB), P20 (512GiB) and P30 (1024GiB), each with its own performance specifications. Depending on your application requirement you can attach one or more of these disks to your Premium Storage supported VM. In the following section on [Premium Storage Scalability and Performance Targets](#) we will describe the specifications in more detail.

Premium Page Blob: Premium Storage supports Azure Page Blobs, which are used to hold persistent disks for Azure Virtual Machines (VMs). Currently, Premium Storage does not support Azure Block Blobs, Azure Append Blobs, Azure Files, Azure Tables, or Azure Queues. Any other object placed in a Premium Storage account will be a Page Blob, and it will snap to one of the supported provisioned sizes. Hence Premium Storage account is not meant for storing tiny blobs.

Premium Storage account: To start using Premium Storage, you must create a Premium Storage account. If you prefer to use the [Azure portal](#), you can create a Premium Storage account by specifying the “Premium” performance tier and “Locally-redundant storage (LRS)” as the replication option. You can also create a Premium Storage account by specifying the type as “Premium_LRS” using the [Storage REST API](#) version 2014-02-14 or later; the [Service Management REST API](#) version 2014-10-01 or later (Classic deployments); the [Azure Storage Resource Provider REST API Reference](#) (Resource Manager deployments); and the [Azure PowerShell](#) version 0.8.10 or later. Learn about premium storage account limits in the following section on [Premium Storage Scalability and Performance Targets](#).

Premium Locally Redundant Storage: A Premium Storage account only supports Locally Redundant Storage (LRS) as the replication option and keeps three copies of the data within a single region. For considerations regarding geo replication when using Premium Storage, see the [Snapshots and Copy Blob](#) section in this article.

Azure uses the storage account as a container for your operating system (OS) and data disks. When you create an Azure DS, DSv2, GS, or Fs VM and select an Azure Premium Storage account, your operating system and data disks are stored in that storage account.

You can use Premium Storage for Disks in one of two ways:

- First, create a new premium storage account. Next, when creating a new DS, DSv2, GS, or Fs VM, select the premium storage account in the Storage configuration settings. OR,
- When creating a new DS, DSv2, GS, or Fs VM create a new premium storage account in Storage configuration settings, or let Azure portal create a default premium storage account.

For step-by-step instructions, see the [Quick Start](#) section later in this article.

NOTE

A premium storage account cannot be mapped to a custom domain name.

Premium Storage supported VMs

Premium Storage supports DS-series, DSv2-series, GS-series, and Fs-series Azure Virtual Machines (VMs). You can use both Standard and Premium storage disks with Premium Storage supported of VMs. But you cannot use Premium Storage disks with VM series which are not Premium Storage compatible.

For information on available Azure VM types and sizes for Windows VMs, see [Windows VM sizes](#). For information on VM types and sizes for Linux VMs, see [Linux VM sizes](#).

Following are some of the features of DS, DSv2, GS, and Fs series VMs:

Cloud Service: DS-series VMs can be added to a cloud service that includes only DS-series VMs. Do not add DS-series VMs to an existing cloud service that includes non-DS-series VMs. You can migrate your existing VHDs to a new cloud service running only DS-series VMs. If you want to retain the same virtual IP address (VIP) for the new cloud service that hosts your DS-series VMs, use the [Reserved IP Addresses](#). GS-series VMs can be added to an existing cloud service running only G-series VMs.

Operating System Disk: The Premium Storage supported Azure virtual machines can be configured to use an operating system (OS) disk hosted either on a Standard Storage account or on a Premium Storage account. We recommend using Premium Storage based OS disk for best experience.

Data Disks: You can use both Premium and Standard storage disks in the same Premium Storage supported VM. With Premium Storage, you can provision a Premium Storage supported VM and attach several persistent data disks to the VM. If needed, you can stripe across the disks to increase the capacity and performance of the volume.

NOTE

If you stripe Premium Storage data disks using [Storage Spaces](#), you should configure it with one column for each disk that is used. Otherwise, overall performance of the striped volume may be lower than expected due to uneven distribution of traffic across the disks. By default, the Server Manager user interface (UI) allows you to setup columns up to 8 disks. But if you have more than 8 disks, you need to use PowerShell to create the volume and also specify the number of columns manually. Otherwise, the Server Manager UI continues to use 8 columns even though you have more disks. For example, if you have 32 disks in a single stripe set, you should specify 32 columns. You can use the *NumberOfColumns* parameter of the [New-VirtualDisk](#) PowerShell cmdlet to specify the number of columns used by the virtual disk. For more information, see [Storage Spaces Overview](#) and [Storage Spaces Frequently Asked Questions](#).

Cache: Premium Storage supported VMs have a unique caching capability with which you can get high levels of throughput and latency, which exceeds underlying Premium Storage disk performance. You can configure disk caching policy on the Premium Storage disks as ReadOnly, ReadWrite or None. The default disk caching policy is ReadOnly for all premium data disks and ReadWrite for operating system disks. Use the right configuration setting to achieve optimal performance for your application. For example, for read heavy or read only data disks, such as SQL Server data files, set disk caching policy to "ReadOnly". For write heavy or write only data disks, such as SQL Server log files, set disk caching policy to "None". Learn more about optimizing your design with Premium Storage in [Design for Performance with Premium Storage](#).

Analytics: To analyze the performance of VMs using disks on Premium Storage accounts, you can enable the Azure VM Diagnostics in the Azure portal. Refer to [Microsoft Azure Virtual Machine Monitoring with Azure Diagnostics Extension](#) for details. To see the disk performance, use operating system based tools, such as [Windows Performance Monitor](#) for Windows VMs and [IOSTAT](#) for Linux VMs.

VM scale limits and performance: Each Premium Storage supported VM size has scale limits and performance specification for IOPS, bandwidth and number of disks that can be attached per VM. When using premium storage disks with Premium Storage supported VMs, make sure there is sufficient IOPS and Bandwidth available on your VM to drive the disk traffic. For example, a STANDARD_DS1 VM has 32 MB per second dedicated bandwidth available for Premium Storage disk traffic. A P10 premium storage disk can provide 100 MB per second bandwidth. If a P10 Premium Storage disk were attached to this VM, it can only go up to 32 MB per second but not up to 100 MB per second that the P10 disk can provide.

Currently, the largest VM on DS-series is Standard_DS15_v2 and it can provide up to 960 MB per second across all disks. The largest VM on GS-series is Standard_GS5 and it can give up to 2000 MB per second across all disks. Note that these limits are for disk traffic alone, not including cache-hits and network traffic. There is a separate bandwidth available for VM network traffic, which is different from the dedicated bandwidth for Premium Storage disks.

For the most up-to-date information on maximum IOPS and throughput (bandwidth) for Premium Storage supported VMs, see [Windows VM sizes](#) or [Linux VM sizes](#).

To learn about the Premium storage disks and their IOPs and throughput limits, see the table in the [Premium Storage Scalability and Performance Targets](#) section in this article.

Premium Storage Scalability and Performance Targets

In this section, we will describe all the Scalability and Performance targets you must consider when using Premium Storage.

Premium Storage account limits

Premium Storage accounts have following scalability targets:

TOTAL ACCOUNT CAPACITY	TOTAL BANDWIDTH FOR A LOCALLY REDUNDANT STORAGE ACCOUNT
Disk Capacity: 35 TB Snapshot capacity: 10 TB	Up to 50 gigabits per second for Inbound + Outbound

- Inbound refers to all data (requests) being sent to a storage account.
- Outbound refers to all data (responses) being received from a storage account.

For more information, see [Azure Storage Scalability and Performance Targets](#).

If the needs of your application exceed the scalability targets of a single storage account, build your application to use multiple storage accounts, and partition your data across those storage accounts. For example, if you want to attach 51 terabytes (TB) disks across a number of VMs, spread them across two storage accounts since 35 TB is the limit for a single Premium Storage account. Make sure that a single Premium Storage account has never more than 35 TB of provisioned disks.

Premium Storage Disks Limits

When you provision a disk against a Premium Storage account, how much input/output operations per second (IOPS) and throughput (bandwidth) it can get depends on the size of the disk. Currently, there are three types of Premium Storage disks: P10, P20, and P30. Each one has specific limits for IOPS and throughput as specified in the following table:

PREMIUM STORAGE DISK TYPE	P10	P20	P30
Disk Size	128 GiB	512 GiB	1024 GiB (1 TB)
IOPS per disk	500	2300	5000
Throughput per disk	100 MB per second	150 MB per second	200 MB per second

NOTE

Make sure that there is sufficient bandwidth available on your VM to drive the disk traffic as explained in the [Premium Storage supported VMs](#) section earlier in this article. Otherwise, your disk throughput and IOPS will be constrained to lower values based on the VM limits rather than the disk limits mentioned in the previous table.

Here are some important things you must know regarding Premium Storage scalability and performance targets:

- **Provisioned Capacity and Performance:** When you provision a premium storage disk, unlike standard storage, you are guaranteed the Capacity, IOPS and Throughput for that disk. For example, if you create a P30 disk, Azure provisions 1024 GB storage capacity, 5000 IOPS and 200 MB per second Throughput for that disk. Your application can use all or part of the capacity and performance.
- **Disk Size:** Azure maps the disk size (rounded up) to the nearest Premium Storage Disk option as specified in the table. For example, a disk of size 100 GiB is classified as a P10 option and can perform up to 500 IO units per second, and with up to 100 MB per second throughput. Similarly, a disk of size 400 GiB is classified as a P20 option, and can perform up to 2300 IO units per second and up to 150 MB per second throughput.

NOTE

You can easily increase the size of existing disks. For example, if you wish to increase the size of a 30 GB disk to 128GB or to 1 TB. Or, if you wish to convert your P20 disk to a P30 disk because you need more capacity or more IOPS and throughput. You can expand the disk using "Update-AzureDisk" PowerShell commandlet with "-ResizedSizeInGB" property. For performing this action, disk needs to be detached from the VM or the VM needs to be stopped.

- **IO Size:** The input/output (I/O) unit size is 256 KB. If the data being transferred is less than 256 KB, it is considered a single I/O unit. The larger I/O sizes are counted as multiple I/Os of size 256 KB. For example, 1100 KB I/O is counted as five I/O units.
- **Throughput:** The throughput limit includes writes to the disk as well as reads from that disk that are not served from the cache. For example, a P10 disk has 100 MB per second throughput per disk. Some examples of valid throughput for the P10 disk are,

MAX THROUGHPUT PER P10 DISK	NON-CACHE READS FROM DISK	NON-CACHE WRITES TO DISK
100 MB per sec	100 MB per sec	0
100 MB per sec	0	100 MB per sec
100 MB per sec	60 MB per sec	40 MB per sec

- **Cache hits:** Cache-hits are not limited by the allocated IOPS/Throughput of the disk. For example, when you use a data disk with ReadOnly cache setting on a Premium Storage supported VM, Reads that are served from the cache are not subject to Hence you could get very high throughput from a disk if the workload is predominantly Reads. Note that, cache is subject to separate IOPS / Throughput limits at VM level based on the VM size. DS-series VMs have roughly 4000 IOPS and 33 MB/sec per core for cache and local SSD IOs. GS-series VMs have a limit of 5000 IOPS and 50 MB/sec per core for cache and local SSD IOs.

Throttling

You may see throttling if your application IOPS or throughput exceed the allocated limits for a Premium Storage disk or if your total disk traffic across all disks on the VM exceeds the disk bandwidth limit available for the VM. To avoid throttling, we recommend that you limit the number of pending I/O requests for disk based on the scalability and performance targets for the disk you have provisioned and based on the disk bandwidth available to the VM.

Your application can achieve the lowest latency when it is designed to avoid throttling. On the other hand, if the number of pending I/O requests for the disk is too small, your application cannot take advantage of the maximum IOPS and throughput levels that are available to the disk.

The following examples demonstrate how to calculate the throttling levels. All calculations are based on I/O unit size of 256 KB:

Example 1:

Your application has done 495 I/O units of 16 KB size in one second on a P10 disk. These will be counted as 495 I/O Units per second (IOPS). If you try a 2 MB I/O in the same second, the total of I/O units is equal to 495 + 8. This is because 2 MB I/O results in $2048 \text{ KB} / 256 \text{ KB} = 8$ I/O Units when the I/O unit size is 256 KB. Since the sum of 495 + 8 exceeds the 500 IOPS limit for the disk, throttling occurs.

Example 2:

Your application has done 400 I/O units of 256 KB size on a P10 disk. The total bandwidth consumed is $(400 * 256 \text{ KB}) / 100 \text{ MB/sec} = 1024 \text{ MB/sec}$.

$256 / 1024 = 100$ MB/sec. A P10 disk has throughput limit of 100 MB per second. If your application tries to perform more I/O in that second, it gets throttled because it exceeds the allocated limit.

Example 3:

You have a DS4 VM with two P30 disks attached. Each P30 disk is capable of 200 MB per second throughput. However, a DS4 VM has a total disk bandwidth capacity of 256 MB per second. Therefore, you cannot drive the attached disks to the maximum throughput on this DS4 VM at the same time. To resolve this, you can sustain traffic of 200 MB per second on one disk and 56 MB per second on the other disk. If the sum of your disk traffic goes over 256 MB per second, the disk traffic gets throttled.

NOTE

If the disk traffic mostly consists of small I/O sizes, it is highly likely that your application will hit the IOPS limit before the throughput limit. On the other hand, if the disk traffic mostly consists of large I/O sizes, it is highly likely that your application will hit the throughput limit instead of the IOPS limit. You can maximize your application IOPS and throughput capacity by using optimal I/O sizes and also by limiting the number of pending I/O requests for disk.

To learn about designing for high performance using Premium Storage read the article, [Design for Performance with Premium Storage](#).

Snapshots and Copy Blob

You can create a snapshot for Premium Storage in the same way as you create a snapshot when using Standard Storage. Since Premium Storage only supports Locally Redundant Storage (LRS) as the replication option, we recommend that you create snapshots and then copy those snapshots to a geo-redundant standard storage account. For more information, see [Azure Storage Redundancy Options](#).

If a disk is attached to a VM, certain API operations are not permitted on the page blob backing the disk. For example, you cannot perform a [Copy Blob](#) operation on that blob as long as the disk is attached to a VM. Instead, first create a snapshot of that blob by using the [Snapshot Blob](#) REST API method, and then perform the [Copy Blob](#) of the snapshot to copy the attached disk. Alternatively, you can detach the disk and then perform any necessary operations on the underlying blob.

Following limits apply to Premium Storage blob snapshots:

PREMIUM STORAGE LIMIT	VALUE
Max. number of snapshots per blob	100
Storage account capacity for snapshots (includes data in snapshots only, and does not include data in base blob)	10 TB
Min. time between consecutive snapshots	10 minutes

To maintain geo-redundant copies of your snapshots, you can copy snapshots from a Premium Storage account to a geo-redundant standard storage account by using AzCopy or Copy Blob. For more information, see [Transfer data with the AzCopy Command-Line Utility](#) and [Copy Blob](#).

For detailed information on performing REST operations against page blobs in Premium Storage accounts, see [Using Blob Service Operations with Azure Premium Storage](#) in the MSDN library.

Using Linux VMs with Premium Storage

Please refer to important instructions below for configuring your Linux VMs on Premium Storage:

- For all Premium Storage disks with cache setting as either "ReadOnly" or "None", you must disable "barriers" while mounting the file system in order to achieve the scalability targets for Premium Storage. You do not need barriers for this scenario because the writes to Premium Storage backed disks are durable for these cache settings. When the write request successfully completes, data has been written to the persistent store. Please use the following methods for disabling "barriers" depending on your file system:
 - If you use **reiserFS**, disable barriers using the mount option "barrier=none" (For enabling barriers, use "barrier=flush")
 - If you use **ext3/ext4**, disable barriers using the mount option "barrier=0" (For enabling barriers, use "barrier=1")
 - If you use **XFS**, disable barriers using the mount option "nobarrier" (For enabling barriers, use the option "barrier")
- For Premium Storage disks with cache setting "ReadWrite", barriers should be enabled for durability of writes.
- For the volume labels to persist after VM reboot, you must update /etc/fstab with the UUID references to the disks. Also refer to [How to Attach a Data Disk to a Linux Virtual Machine](#)

Following are the Linux Distributions that we validated with Premium Storage. We recommend that you upgrade your VMs to at least one of these versions (or later) for better performance and stability with Premium Storage. Also, some of the versions require the latest LIS (Linux Integration Services v4.0 for Microsoft Azure). Please follow the link provided below for download and installation. We will continue to add more images to the list as we complete additional validations. Please note, our validations showed that performance varies for these images, and it also depends on workload characteristics and settings on the images. Different images are tuned for different kinds of workload.

DISTRIBUTION	VERSION	SUPPORTED KERNEL	DETAILS
Ubuntu	12.04	3.2.0-75.110+	Ubuntu-12_04_5-LTS-amd64-server-20150119-en-us-30GB
Ubuntu	14.04	3.13.0-44.73+	Ubuntu-14_04_1-LTS-amd64-server-20150123-en-us-30GB
Debian	7.x, 8.x	3.16.7-ckt4-1+	
SUSE	SLES 12	3.12.36-38.1+	suse-sles-12-priority-v20150213 suse-sles-12-v20150213
SUSE	SLES 11 SP4	3.0.101-0.63.1+	
CoreOS	584.0.0+	3.18.4+	CoreOS 584.0.0
CentOS	6.5, 6.6, 6.7, 7.0		LIS4 Required <i>See note below</i>
CentOS	7.1+	3.10.0-229.1.2.el7+	LIS4 Recommended <i>See note below</i>
RHEL	6.8+, 7.2+		
Oracle	6.0+, 7.2+		UEK4 or RHCK

DISTRIBUTION	VERSION	SUPPORTED KERNEL	DETAILS
Oracle	7.0-7.1		UEK4 or RHCK w/ LIS 4.1+
Oracle	6.4-6.7		UEK4 or RHCK w/ LIS 4.1+

LIS Drivers for Openlogic CentOS

Customers running OpenLogic CentOS VMs should run the following command to install the latest drivers:

```
sudo rpm -e hypervkvpd ## (may return error if not installed, that's OK)
sudo yum install microsoft-hyper-v
```

A reboot will then be required to activate the new drivers.

Pricing and Billing

When using Premium Storage, the following billing considerations apply:

- Premium Storage Disk / Blob Size
- Premium Storage Snapshots
- Outbound data transfers

Premium Storage Disk / Blob Size: Billing for a Premium Storage disk/blob depends on the provisioned size of the disk/blob. Azure maps the provisioned size (rounded up) to the nearest Premium Storage Disk option as specified in the table given in the [Scalability and Performance Targets when using Premium Storage](#) section. All objects stored in a Premium Storage account will map to one of the supported provisioned sizes and will be billed accordingly. Hence avoid using Premium Storage account for storing tiny blobs. Billing for any provisioned disk/blob is prorated hourly using the monthly price for the Premium Storage offer. For example, if you provisioned a P10 disk and deleted it after 20 hours, you are billed for the P10 offering prorated to 20 hours. This is regardless of the amount of actual data written to the disk or the IOPS/throughput used.

Premium Storage Snapshots: Snapshots on Premium Storage are billed for the additional capacity used by the snapshots. For information on snapshots, see [Creating a Snapshot of a Blob](#).

Outbound data transfers: [Outbound data transfers](#) (data going out of Azure data centers) incur billing for bandwidth usage.

For detailed information on pricing for Premium Storage, Premium Storage supported VMs, see:

- [Azure Storage Pricing](#)
- [Virtual Machines Pricing](#)

Backup

Virtual machines using premium storage can be backed up using Azure Backup. [More details](#).

Quick Start: Create and use a Premium Storage account for a virtual machine data disk

In this section we will demonstrate the following scenarios using Azure portal, Azure PowerShell and Azure CLI:

- How to create a Premium Storage account.
- How to create a virtual machine and attach a data disk to the virtual machine when using Premium Storage.
- How to change disk caching policy of a data disk attached to a virtual machine.

Create an Azure virtual machine using Premium Storage via the Azure portal

To create a virtual machine in Premium Storage, you first have to create a Premium Storage account.

Create a Premium Storage account in Azure portal

This section shows how to create a Premium Storage account using the Azure portal.

1. Sign in to the [Azure portal](#). Check out the [Free Trial](#) offer if you do not have a subscription yet.
2. On the Hub menu, select **New -> Data + Storage -> Storage account**.
3. Enter a name for your storage account.

NOTE

Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only.

Your storage account name must be unique within Azure. The Azure portal will indicate if the storage account name you select is already in use.

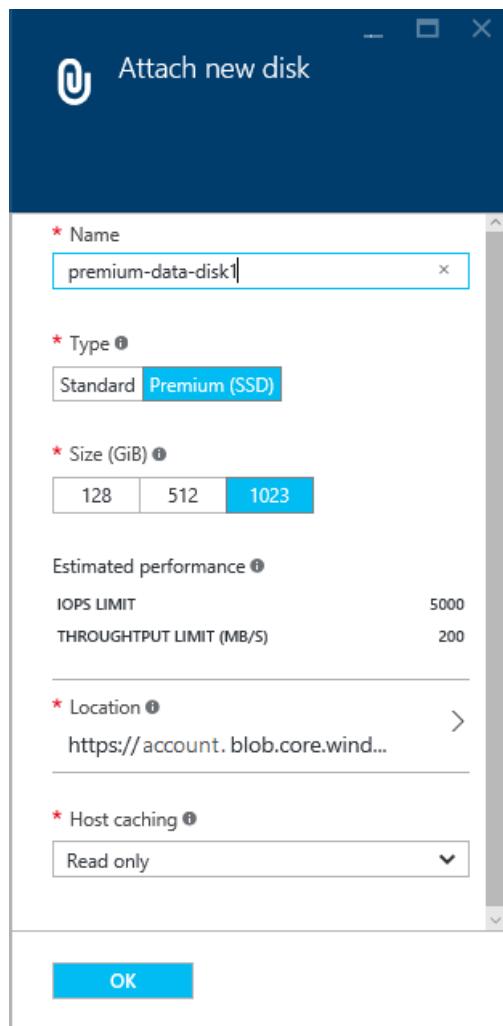
4. Specify the deployment model to be used: **Resource Manager** or **Classic**. **Resource Manager** is the recommended deployment model. For more information, see [Understanding Resource Manager deployment and classic deployment](#).
5. Specify the performance tier for the storage account as **Premium**.
6. **Locally-redundant storage (LRS)** is the only available replication option with Premium Storage. For more details on Azure Storage replication options, see [Azure Storage replication](#).
7. Select the subscription in which you want to create the new storage account.
8. Specify a new resource group or select an existing resource group. For more information on resource groups, see [Azure Resource Manager overview](#).
9. Select the geographic location for your storage account. You can confirm whether Premium Storage is available in the selected Location by referring to [Azure Services by Region](#).
10. Click **Create** to create the storage account.

Create an Azure virtual machine via Azure portal

You must create a Premium Storage supported VM to be able to use Premium Storage. Follow the steps in [Create a Windows virtual machine in the Azure portal](#) to create a new DS, DSv2, GS, or Fs virtual machine.

Attach a premium storage data disk via Azure portal

1. Find the new or existing DS, DSv2, GS, or Fs VM in Azure portal.
2. In the VM **All Settings**, go to **Disks** and click on **Attach New**.
3. Enter the name of your data disk and select the **Type** as **Premium**. Select the desired **Size** and **Host caching** setting.



See more detailed steps in [How to attach a data disk in Azure portal](#).

Change disk caching policy via Azure portal

1. Find the new or existing DS, DSv2, GS, or Fs VM in Azure portal.
2. In the VM All Settings, go to Disks and click on the disk you wish to change.
3. Change the Host caching option to the desired value, None or ReadOnly or ReadWrite

WARNING

Changing the cache setting of an Azure disk detaches and re-attaches the target disk. If it is the operating system disk, the VM is restarted. Stop all applications/services that might be affected by this disruption before changing the disk cache setting.

Create an Azure virtual machine using Premium Storage via Azure PowerShell

To create a virtual machine in Premium Storage, you first have to create a Premium Storage account.

Create a Premium Storage account in Azure PowerShell

This PowerShell example shows how to create a new Premium Storage account and attach a data disk that uses that account to a new Azure virtual machine.

Setup your PowerShell environment by following the steps given at [How to install and configure Azure PowerShell](#).

Start the PowerShell console, connect to your subscription, and run the following PowerShell cmdlet in the console window. As seen in this PowerShell statement, you need to specify the **Type** parameter as **Premium_LRS** when you create a Premium Storage account.

```
New-AzureStorageAccount -StorageAccountName "yourpremiumaccount" -Location "West US" -Type "Premium_LRS"
```

Create an Azure virtual machine via Azure PowerShell

Next, create a new DS-Series VM and specify that you want Premium Storage by running the following PowerShell cmdlets in the console window. You can create a GS-series VM using the same steps. Specify the appropriate VM size in the commands. For e.g. Standard_GS2:

```
$storageAccount = "yourpremiumaccount"
$adminName = "youradmin"
$adminPassword = "yourpassword"
$vmName ="yourVM"
$location = "West US"
$imageName = "a699494373c04fc0bc8f2bb1389d6106__Windows-Server-2012-R2-201409.01-en.us-127GB.vhd"
$vmSize ="Standard_DS2"
$OSDiskPath = "https://" + $storageAccount + ".blob.core.windows.net/vhds/" + $vmName + "_OS_PIO.vhd"
$vm = New-AzureVMConfig -Name $vmName -ImageName $imageName -InstanceSize $vmSize -MediaLocation $OSDiskPath
Add-AzureProvisioningConfig -Windows -VM $vm -AdminUsername $adminName -Password $adminPassword
New-AzureVM -ServiceName $vmName -VMs $VM -Location $location
```

Attach a premium storage data disk via Azure PowerShell

If you want more disk space for your VM, attach a new data disk to an existing Premium Storage supported VM after it is created by running the following PowerShell cmdlets in the console window:

```
$storageAccount = "yourpremiumaccount"
$vmName ="yourVM"
$vm = Get-AzureVM -ServiceName $vmName -Name $vmName
$LunNo = 1
$path = "http://" + $storageAccount + ".blob.core.windows.net/vhds/" + "myDataDisk_" + $LunNo + "_PIO.vhd"
$label = "Disk " + $LunNo
Add-AzureDataDisk -CreateNew -MediaLocation $path -DiskSizeInGB 128 -DiskLabel $label -LUN $LunNo -HostCaching
ReadOnly -VM $vm | Update-AzureVm
```

Change disk caching policy via Azure PowerShell

To update the disk caching policy, note the LUN number of the data disk attached. Run the following command to update data disk attached at LUN number 2, to ReadOnly.

```
Get-AzureVM "myservice" -name "MyVM" | Set-AzureDataDisk -LUN 2 -HostCaching ReadOnly | Update-AzureVm
```

WARNING

Changing the cache setting of an Azure disk detaches and re-attaches the target disk. If it is the operating system disk, the VM is restarted. Stop all applications/services that might be affected by this disruption before changing the disk cache setting.

Create an Azure virtual machine using Premium Storage via the Azure Command-Line Interface

The [Azure Command-Line Interface](#)(Azure CLI) provides a provides a set of open source, cross-platform commands for working with the Azure Platform. The following examples show how to use Azure CLI (version 0.8.14 and later) to create a Premium Storage account, a new virtual machine, and attach a new data disk from a Premium Storage account.

Create a Premium Storage account via Azure CLI

```
azure storage account create "premiumtestaccount" -l "west us" --type PLRS
```

Create a DS-series virtual machine via Azure CLI

```
azure vm create -z "Standard_DS2" -l "west us" -e 22 "premium-test-vm"
  "b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-14_10-amd64-server-20150202-en-us-30GB" -u "myusername" -p
  "passwd@123"
```

Display information about the virtual machine

```
azure vm show premium-test-vm
```

Attach a new premium data disk via Azure CLI

```
azure vm disk attach-new premium-test-vm 20 https://premiumstorageaccount.blob.core.windows.net/vhd-
store/data1.vhd
```

Display information about the new data disk

```
azure vm disk show premium-test-vm-premium-test-vm-0-201502210429470316
```

Change disk caching policy

To change the cache policy on one of your disks using Azure CLI, run the following command:

```
$ azure vm disk attach -h ReadOnly <VM-Name> <Disk-Name>
```

Note that the caching policy options can be `ReadOnly`, `None`, or `ReadWrite`. For more options, see the help by running the following command:

```
azure vm disk attach --help
```

WARNING

Changing the cache setting of an Azure disk detaches and re-attaches the target disk. If it is the operating system disk, the VM is restarted. Stop all applications/services that might be affected by this disruption before changing the disk cache setting.

FAQs

1. Can I attach both premium and standard data disks to a Premium Storage supported VM?

Yes. You can attach both premium and standard data disks to a Premium Storage supported series VM.

2. Can I attach both premium and standard data disks to a D, Dv2, G or F series VM?

No. You can only attach a standard data disk to all VMs that are not Premium Storage supported series.

3. If I create a premium data disk from an existing VHD that was 80 GB in size, how much will that cost me?

A premium data disk created from 80 GB VHD will be treated as the next available premium disk size, a P10 disk. You will be charged as per the P10 disk pricing.

4. Are there any transaction costs when using Premium Storage?

There is a fixed cost for each disk size which comes provisioned with certain number of IOPS and Throughput. The only other costs are outbound bandwidth and snapshots capacity, if applicable. See [Azure Storage Pricing](#) for more details.

5. Where can I store boot diagnostics for my Premium Storage supported series VM?

Create a standard storage account to store the boot diagnostics of your Premium Storage supported series VM.

6. How many IOPS and Throughput can I get from the disk cache?

The combined limits for cache and local SSD for a DS series are 4000 IOPS per core and 33 MB per second per core. GS series offers 5000 IOPS per core and 50 MB per second per core.

7. What is the local SSD in a Premium Storage supported series VM?

The local SSD is a temporary storage that is included with a Premium Storage supported series VM. There is no extra cost for this temporary storage. It is recommended that you do not use this temporary storage or local SSD for storing your application data as it is not persisted in Azure Blob Storage.

8. Can I convert my standard storage account to a Premium Storage account?

No. It is not possible to convert standard storage account to Premium Storage account or vice versa. You must create a new storage account with the desired type and copy data to new storage account, if applicable.

9. How can I convert my D series VM to a DS series VM?

Please refer to the migration guide, [Migrating to Azure Premium Storage](#) to move your workload from a D series VM using standard storage account to a DS series VM using Premium Storage account.

Next steps

For more information about Azure Premium Storage refer to the following articles.

Design and implement with Azure Premium Storage

- [Design for Performance with Premium Storage](#)
- [Using Blob Service Operations with Azure Premium Storage](#)

Operational guidance

- [Migrating to Azure Premium Storage](#)

Blog Posts

- [Azure Premium Storage Generally Available](#)
- [Announcing the GS-Series: Adding Premium Storage Support to the Largest VMs in the Public Cloud](#)

Migrating to Azure Premium Storage

1/17/2017 • 31 min to read • [Edit on GitHub](#)

Overview

Azure Premium Storage delivers high-performance, low-latency disk support for virtual machines running I/O-intensive workloads. You can take advantage of the speed and performance of these disks by migrating your application's VM disks to Azure Premium Storage.

The purpose of this guide is to help new users of Azure Premium Storage better prepare to make a smooth transition from their current system to Premium Storage. The guide addresses three of the key components in this process:

- [Plan for the Migration to Premium Storage](#)
- [Prepare and Copy Virtual Hard Disks \(VHDs\) to Premium Storage](#)
- [Create Azure Virtual Machine using Premium Storage](#)

You can either migrate VMs from other platforms to Azure Premium Storage or migrate existing Azure VMs from Standard Storage to Premium Storage. This guide covers steps for both two scenarios. Follow the steps specified in the relevant section depending on your scenario.

NOTE

You can find a feature overview and pricing of Premium Storage in Premium Storage: [High-Performance Storage for Azure Virtual Machine Workloads](#). We recommend migrating any virtual machine disk requiring high IOPS to Azure Premium Storage for the best performance for your application. If your disk does not require high IOPS, you can limit costs by maintaining it in Standard Storage, which stores virtual machine disk data on Hard Disk Drives (HDDs) instead of SSDs.

Completing the migration process in its entirety may require additional actions both before and after the steps provided in this guide. Examples include configuring virtual networks or endpoints or making code changes within the application itself which may require some downtime in your application. These actions are unique to each application, and you should complete them along with the steps provided in this guide to make the full transition to Premium Storage as seamless as possible.

Plan for the migration to Premium Storage

This section ensures that you are ready to follow the migration steps in this article, and helps you to make the best decision on VM and Disk types.

Prerequisites

- You will need an Azure subscription. If you don't have one, you can create a one-month [free trial](#) subscription or visit [Azure Pricing](#) for more options.
- To execute PowerShell cmdlets, you will need the Microsoft Azure PowerShell module. See [How to install and configure Azure PowerShell](#) for the install point and installation instructions.
- When you plan to use Azure VMs running on Premium Storage, you need to use the Premium Storage capable VMs. You can use both Standard and Premium Storage disks with Premium Storage capable VMs. Premium storage disks will be available with more VM types in the future. For more information on all available Azure VM disk types and sizes, see [Sizes for virtual machines](#) and [Sizes for Cloud Services](#).

Considerations

An Azure VM supports attaching several Premium Storage disks so that your applications can have up to 64 TB of storage per VM. With Premium Storage, your applications can achieve 80,000 IOPS (input/output operations per second) per VM and 2000 MB per second throughput per VM with extremely low latencies for read operations. You have options in a variety of VMs and Disks. This section is to help you to find an option that best suits your workload.

VM sizes

The Azure VM size specifications are listed in [Sizes for virtual machines](#). Review the performance characteristics of virtual machines that work with Premium Storage and choose the most appropriate VM size that best suits your workload. Make sure that there is sufficient bandwidth available on your VM to drive the disk traffic.

Disk sizes

There are three types of disks that can be used with your VM and each has specific IOPs and throughput limits. Take into consideration these limits when choosing the disk type for your VM based on the needs of your application in terms of capacity, performance, scalability, and peak loads.

PREMIUM STORAGE DISK TYPE	P10	P20	P30
Disk size	128 GB	512 GB	1024 GB (1 TB)
IOPS per disk	500	2300	5000
Throughput per disk	100 MB per second	150 MB per second	200 MB per second

Depending on your workload, determine if additional data disks are necessary for your VM. You can attach several persistent data disks to your VM. If needed, you can stripe across the disks to increase the capacity and performance of the volume. (See what is Disk Striping [here](#).) If you stripe Premium Storage data disks using [Storage Spaces](#), you should configure it with one column for each disk that is used. Otherwise, the overall performance of the striped volume may be lower than expected due to uneven distribution of traffic across the disks. For Linux VMs you can use the *mdadm* utility to achieve the same. See article [Configure Software RAID on Linux](#) for details.

Storage account scalability targets

Premium Storage accounts have the following scalability targets in addition to the [Azure Storage Scalability and Performance Targets](#). If your application requirements exceed the scalability targets of a single storage account, build your application to use multiple storage accounts, and partition your data across those storage accounts.

TOTAL ACCOUNT CAPACITY	TOTAL BANDWIDTH FOR A LOCALLY REDUNDANT STORAGE ACCOUNT
Disk capacity: 35TB Snapshot capacity: 10 TB	Up to 50 gigabits per second for Inbound + Outbound

For the more information on Premium Storage specifications, check out [Scalability and Performance Targets when using Premium Storage](#).

Disk caching policy

By default, disk caching policy is *Read-Only* for all the Premium data disks, and *Read-Write* for the Premium operating system disk attached to the VM. This configuration setting is recommended to achieve the optimal performance for your application's IOs. For write-heavy or write-only data disks (such as SQL Server log files), disable disk caching so that you can achieve better application performance. The cache settings for existing data disks can be updated using [Azure Portal](#) or the *-HostCaching* parameter of the *Set-AzureDataDisk* cmdlet.

Location

Pick a location where Azure Premium Storage is available. See [Azure Services by Region](#) for up-to-date information on available locations. VMs located in the same region as the Storage account that stores the disks for

the VM will give much better performance than if they are in separate regions.

Other Azure VM configuration settings

When creating an Azure VM, you will be asked to configure certain VM settings. Remember, few settings are fixed for the lifetime of the VM, while you can modify or add others later. Review these Azure VM configuration settings and make sure that these are configured appropriately to match your workload requirements.

Optimization

[Azure Premium Storage: Design for High Performance](#) provides guidelines for building high-performance applications using Azure Premium Storage. You can follow the guidelines combined with performance best practices applicable to technologies used by your application.

Prepare and copy virtual hard disks (VHDs) to Premium Storage

The following section provides guidelines for preparing VHDs from your VM and copying VHDs to Azure Storage.

- [Scenario 1: "I am migrating existing Azure VMs to Azure Premium Storage."](#)
- [Scenario 2: "I am migrating VMs from other platforms to Azure Premium Storage."](#)

Prerequisites

To prepare the VHDs for migration, you'll need:

- An Azure subscription, a storage account, and a container in that storage account to which you can copy your VHD. Note that the destination storage account can be a Standard or Premium Storage account depending on your requirement.
- A tool to generalize the VHD if you plan to create multiple VM instances from it. For example, sysprep for Windows or virt-sysprep for Ubuntu.
- A tool to upload the VHD file to the Storage account. See [Transfer data with the AzCopy Command-Line Utility](#) or use an [Azure storage explorer](#). This guide describes copying your VHD using the AzCopy tool.

NOTE

If you choose synchronous copy option with AzCopy, for optimal performance, copy your VHD by running one of these tools from an Azure VM that is in the same region as the destination storage account. If you are copying a VHD from an Azure VM in a different region, your performance may be slower.

For copying a large amount of data over limited bandwidth, consider [using the Azure Import/Export service to transfer data to Blob Storage](#); this allows you to transfer your data by shipping hard disk drives to an Azure datacenter. You can use the Azure Import/Export service to copy data to a standard storage account only. Once the data is in your standard storage account, you can use either the [Copy Blob API](#) or AzCopy to transfer the data to your premium storage account.

Note that Microsoft Azure only supports fixed size VHD files. VHDX files or dynamic VHDs are not supported. If you have a dynamic VHD, you can convert it to fixed size using the [Convert-VHD](#) cmdlet.

Scenario 1: "I am migrating existing Azure VMs to Azure Premium Storage."

If you are migrating existing Azure VMs, stop the VM, prepare VHDs per the type of VHD you want, and then copy the VHD with AzCopy or PowerShell.

The VM needs to be completely down to migrate a clean state. There will be a downtime until the migration completes.

Step 1. Prepare VHDs for migration

If you are migrating existing Azure VMs to Premium Storage, your VHD may be:

- A generalized operating system image
- A unique operating system disk

- A data disk

Below we walk through these 3 scenarios for preparing your VHD.

Use a generalized Operating System VHD to create multiple VM instances

If you are uploading a VHD that will be used to create multiple generic Azure VM instances, you must first generalize VHD using a sysprep utility. This applies to a VHD that is on-premises or in the cloud. Sysprep removes any machine-specific information from the VHD.

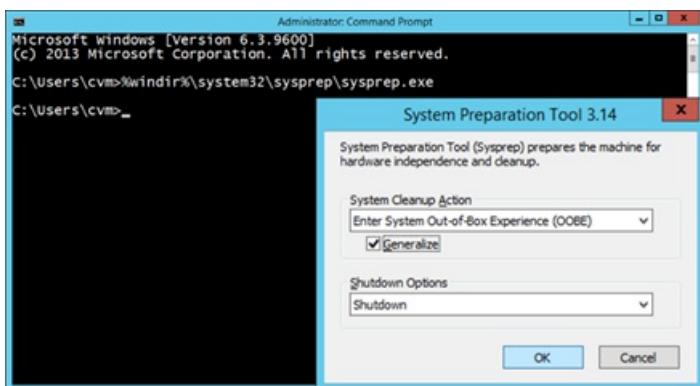
IMPORTANT

Take a snapshot or backup your VM before generalizing it. Running sysprep will stop and deallocate the VM instance. Follow steps below to sysprep a Windows OS VHD. Note that running the Sysprep command will require you to shut down the virtual machine. For more information about Sysprep, see [Sysprep Overview](#) or [Sysprep Technical Reference](#).

1. Open a Command Prompt window as an administrator.
2. Enter the following command to open Sysprep:

```
%windir%\system32\sysprep\sysprep.exe
```

3. In the System Preparation Tool, select Enter System Out-of-Box Experience (OOBE), select the Generalize check box, select **Shutdown**, and then click **OK**, as shown in the image below. Sysprep will generalize the operating system and shut down the system.



For an Ubuntu VM, use `virt-sysprep` to achieve the same. See [virt-sysprep](#) for more details. See also some of the open source [Linux Server Provisioning software](#) for other Linux operating systems.

Use a unique Operating System VHD to create a single VM instance

If you have an application running on the VM which requires the machine specific data, do not generalize the VHD. A non-generalized VHD can be used to create a unique Azure VM instance. For example, if you have Domain Controller on your VHD, executing sysprep will make it ineffective as a Domain Controller. Review the applications running on your VM and the impact of running sysprep on them before generalizing the VHD.

Register data disk VHD

If you have data disks in Azure to be migrated, you must make sure the VMs that use these data disks are shut down.

Follow the steps described below to copy VHD to Azure Premium Storage and register it as a provisioned data disk.

Step 2. Create the destination for your VHD

Create a storage account for maintaining your VHDs. Consider the following points when planning where to store your VHDs:

- The target Premium storage account.
- The storage account location must be same as Premium Storage capable Azure VMs you will create in the final

stage. You could copy to a new storage account, or plan to use the same storage account based on your needs.

- Copy and save the storage account key of the destination storage account for the next stage.

For data disks, you can choose to keep some data disks in a standard storage account (for example, disks that have cooler storage), but we strongly recommend you moving all data for production workload to use premium storage.

Step 3. Copy VHD with AzCopy or PowerShell

You will need to find your container path and storage account key to process either of these two options. Container path and storage account key can be found in **Azure Portal > Storage**. The container URL will be like "<https://myaccount.blob.core.windows.net/mycontainer/>".

Option 1: Copy a VHD with AzCopy (Asynchronous copy)

Using AzCopy, you can easily upload the VHD over the Internet. Depending on the size of the VHDs, this may take time. Remember to check the storage account ingress/egress limits when using this option. See [Azure Storage Scalability and Performance Targets](#) for details.

1. Download and install AzCopy from here: [Latest version of AzCopy](#)
2. Open Azure PowerShell and go to the folder where AzCopy is installed.
3. Use the following command to copy the VHD file from "Source" to "Destination".

```
AzCopy /Source: <source> /SourceKey: <source-account-key> /Dest: <destination> /DestKey: <dest-account-key> /BlobType:page /Pattern: <file-name>
```

Example:

```
AzCopy /Source:https://sourceaccount.blob.core.windows.net/mycontainer1 /SourceKey:key1  
/Dest:https://destaccount.blob.core.windows.net/mycontainer2 /DestKey:key2 /Pattern:abc.vhd
```

Here are descriptions of the parameters used in the AzCopy command:

- **/Source: *<source>:*** Location of the folder or storage container URL that contains the VHD.
- **/SourceKey: *<source-account-key>:*** Storage account key of the source storage account.
- **/Dest: *<destination>:*** Storage container URL to copy the VHD to.
- **/DestKey: *<dest-account-key>:*** Storage account key of the destination storage account.
- **/Pattern: *<file-name>:*** Specify the file name of the VHD to copy.

For details on using AzCopy tool, see [Transfer data with the AzCopy Command-Line Utility](#).

Option 2: Copy a VHD with PowerShell (Synchronized copy)

You can also copy the VHD file using the PowerShell cmdlet Start-AzureStorageBlobCopy. Use the following command on Azure PowerShell to copy VHD. Replace the values in <> with corresponding values from your source and destination storage account. To use this command, you must have a container called vhds in your destination storage account. If the container doesn't exist, create one before running the command.

```
$sourceBlobUri = <source-vhd-uri>  
  
$sourceContext = New-AzureStorageContext -StorageAccountName <source-account> -StorageAccountKey <source-account-key>  
  
$destinationContext = New-AzureStorageContext -StorageAccountName <dest-account> -StorageAccountKey <dest-account-key>  
  
Start-AzureStorageBlobCopy -srcUri $sourceBlobUri -SrcContext $sourceContext -DestContainer <dest-container> -  
DestBlob <dest-disk-name> -DestContext $destinationContext
```

Example:

```
C:\PS> $sourceBlobUri = "https://sourceaccount.blob.core.windows.net/vhds/myvhd.vhd"

C:\PS> $sourceContext = New-AzureStorageContext -StorageAccountName "sourceaccount" -StorageAccountKey "J4zUI9T5b8gvHohkiRg"

C:\PS> $destinationContext = New-AzureStorageContext -StorageAccountName "destaccount" -StorageAccountKey "XZTmqSGKUYFSh7zB5"

C:\PS> Start-AzureStorageBlobCopy -srcUri $sourceBlobUri -SrcContext $sourceContext -DestContainer "vhds" -DestBlob "myvhd.vhd" -DestContext $destinationContext
```

Scenario 2: "I am migrating VMs from other platforms to Azure Premium Storage."

If you are migrating VHD from non-Azure Cloud Storage to Azure, you must first export the VHD to a local directory. Have the complete source path of the local directory where VHD is stored handy, and then using AzCopy to upload it to Azure Storage.

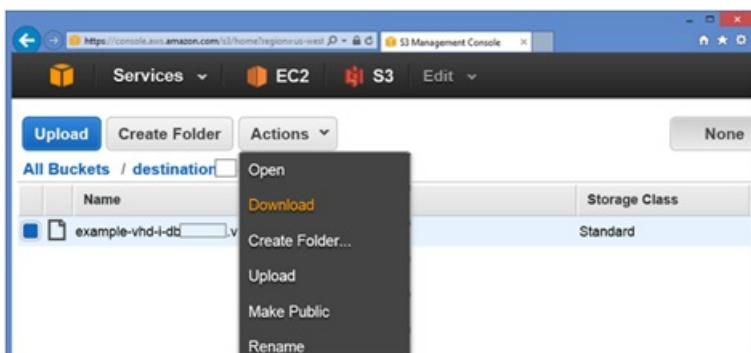
Step 1. Export VHD to a local directory

[Copy a VHD from AWS](#)

1. If you are using AWS, export the EC2 instance to a VHD in an Amazon S3 bucket. Follow the steps described in the Amazon documentation for Exporting Amazon EC2 Instances to install the Amazon EC2 command-line interface (CLI) tool and run the create-instance-export-task command to export the EC2 instance to a VHD file. Be sure to use **VHD** for the DISK_IMAGE_FORMAT variable when running the **create-instance-export-task** command. The exported VHD file is saved in the Amazon S3 bucket you designate during that process.

```
aws ec2 create-instance-export-task --instance-id ID --target-environment TARGET_ENVIRONMENT \
--export-to-s3-task
DiskImageFormat=DISK_IMAGE_FORMAT,ContainerFormat=ova,S3Bucket=BUCKET,S3Prefix=PREFIX
```

2. Download the VHD file from the S3 bucket. Select the VHD file, then **Actions > Download**.



[Copy a VHD from other non-Azure cloud](#)

If you are migrating VHD from non-Azure Cloud Storage to Azure, you must first export the VHD to a local directory. Copy the complete source path of the local directory where VHD is stored.

[Copy a VHD from on-premises](#)

If you are migrating VHD from an on-premises environment, you will need the complete source path where VHD is stored. The source path could be a server location or file share.

Step 2. Create the destination for your VHD

Create a storage account for maintaining your VHDS. Consider the following points when planning where to store your VHDS:

- The target storage account could be standard or premium storage depending on your application requirement.
- The storage account region must be same as Premium Storage capable Azure VMs you will create in the final stage. You could copy to a new storage account, or plan to use the same storage account based on your needs.
- Copy and save the storage account key of the destination storage account for the next stage.

We strongly recommend you moving all data for production workload to use premium storage.

Step 3. Upload the VHD to Azure Storage

Now that you have your VHD in the local directory, you can use AzCopy or AzurePowerShell to upload the .vhf file to Azure Storage. Both options are provided here:

Option 1: Using Azure PowerShell Add-AzureVhd to upload the .vhf file

```
Add-AzureVhd [-Destination] <Uri> [-LocalFilePath] <FileInfo>
```

An example might be "<https://storagesample.blob.core.windows.net/mycontainer/blob1.vhd>". An example might be "**C:\path\to\upload.vhd**".

Option 2: Using AzCopy to upload the .vhf file

Using AzCopy, you can easily upload the VHD over the Internet. Depending on the size of the VHDs, this may take time. Remember to check the storage account ingress/egress limits when using this option. See [Azure Storage Scalability and Performance Targets](#) for details.

1. Download and install AzCopy from here: [Latest version of AzCopy](#)
2. Open Azure PowerShell and go to the folder where AzCopy is installed.
3. Use the following command to copy the VHD file from "Source" to "Destination".

```
AzCopy /Source: <source> /SourceKey: <source-account-key> /Dest: <destination> /DestKey: <dest-account-key> /BlobType:page /Pattern: <file-name>
```

Example:

```
AzCopy /Source:https://sourceaccount.blob.core.windows.net/mycontainer1 /SourceKey:key1  
/Dest:https://destaccount.blob.core.windows.net/mycontainer2 /DestKey:key2 /BlobType:page  
/Pattern:abc.vhd
```

Here are descriptions of the parameters used in the AzCopy command:

- **/Source: *<source>:*** Location of the folder or storage container URL that contains the VHD.
- **/SourceKey: *<source-account-key>:*** Storage account key of the source storage account.
- **/Dest: *<destination>:*** Storage container URL to copy the VHD to.
- **/DestKey: *<dest-account-key>:*** Storage account key of the destination storage account.
- **/BlobType: page:** Specifies that the destination is a page blob.
- **/Pattern: *<file-name>:*** Specify the file name of the VHD to copy.

For details on using AzCopy tool, see [Transfer data with the AzCopy Command-Line Utility](#).

Other options for uploading a VHD

You can also upload a VHD to your storage account using one of the following means:

- [Azure Storage Copy Blob API](#)
- [Azure Storage Explorer Uploading Blobs](#)
- [Storage Import/Export Service REST API Reference](#)

NOTE

We recommend using Import/Export Service if estimated uploading time is longer than 7 days. You can use [DataTransferSpeedCalculator](#) to estimate the time from data size and transfer unit.

Import/Export can be used to copy to a standard storage account. You will need to copy from standard storage to premium storage account using a tool like AzCopy.

Create Azure VMs using Premium Storage

After the VHD is uploaded or copied to the desired storage account, follow the instructions in this section to register the VHD as an OS image, or OS disk depending on your scenario and then create a VM instance from it. The data disk VHD can be attached to the VM once it is created. A sample migration script is provided at the end of this section. This simple script does not match all scenarios. You may need to update the script to match with your specific scenario. To see if this script applies to your scenario, see below [A Sample Migration Script](#).

Checklist

1. Wait until all the VHD disks copying is complete.
2. Make sure Premium Storage is available in the region you are migrating to.
3. Decide the new VM series you will be using. It should be a Premium Storage capable, and the size should be depending on the availability in the region and based on your needs.
4. Decide the exact VM size you will use. VM size needs to be large enough to support the number of data disks you have. E.g. if you have 4 data disks, the VM must have 2 or more cores. Also, consider processing power, memory and network bandwidth needs.
5. Create a Premium Storage account in the target region. This is the account you will use for the new VM.
6. Have the current VM details handy, including the list of disks and corresponding VHD blobs.

Prepare your application for downtime. To do a clean migration, you have to stop all the processing in the current system. Only then you can get it to consistent state which you can migrate to the new platform. Downtime duration will depend on the amount of data in the disks to migrate.

NOTE

If you are creating an Azure Resource Manager VM from a specialized VHD Disk, please refer to [this template](#) for deploying Resource Manager VM using existing disk.

Register your VHD

To create a VM from OS VHD or to attach a data disk to a new VM, you must first register them. Follow steps below depending on your VHD's scenario.

Generalized Operating System VHD to create multiple Azure VM instances

After generalized OS image VHD is uploaded to the storage account, register it as an **Azure VM Image** so that you can create one or more VM instances from it. Use the following PowerShell cmdlets to register your VHD as an Azure VM OS image. Provide the complete container URL where VHD was copied to.

```
Add-AzureVMImage -ImageName "OSImageName" -MediaLocation  
"https://storageaccount.blob.core.windows.net/vhdcontainer/osimage.vhd" -OS Windows
```

Copy and save the name of this new Azure VM Image. In the example above, it is *OSImageName*.

Unique Operating System VHD to create a single Azure VM instance

After the unique OS VHD is uploaded to the storage account, register it as an **Azure OS Disk** so that you can create a VM instance from it. Use these PowerShell cmdlets to register your VHD as an Azure OS Disk. Provide the complete container URL where VHD was copied to.

```
Add-AzureDisk -DiskName "OSDisk" -MediaLocation  
"https://storageaccount.blob.core.windows.net/vhdcontainer/osdisk.vhd" -Label "My OS Disk" -OS "Windows"
```

Copy and save the name of this new Azure OS Disk. In the example above, it is *OSDisk*.

Data disk VHD to be attached to new Azure VM instance(s)

After the data disk VHD is uploaded to storage account, register it as an Azure Data Disk so that it can be attached

to your new DS Series, DSv2 series or GS Series Azure VM instance.

Use these PowerShell cmdlets to register your VHD as an Azure Data Disk. Provide the complete container URL where VHD was copied to.

```
Add-AzureDisk -DiskName "DataDisk" -MediaLocation  
"https://storageaccount.blob.core.windows.net/vhdcontainer/datadisk.vhd" -Label "My Data Disk"
```

Copy and save the name of this new Azure Data Disk. In the example above, it is *DataDisk*.

Create a Premium Storage capable VM

Once the OS image or OS disk are registered, create a new DS-series, DSv2-series or GS-series VM. You will be using the operating system image or operating system disk name that you registered. Select the VM type from the Premium Storage tier. In example below, we are using the *Standard_DS2* VM size.

NOTE

Update the disk size to make sure it matches your capacity and performance requirements and the available Azure disk sizes.

Follow the step by step PowerShell cmdlets below to create the new VM. First, set the common parameters:

```
$serviceName = "yourVM"  
$location = "location-name" (e.g., West US)  
$vmSize ="Standard_DS2"  
$adminUser = "youradmin"  
$adminPassword = "yourpassword"  
$vmName ="yourVM"  
$vmSize = "Standard_DS2"
```

First, create a cloud service in which you will be hosting your new VMs.

```
New-AzureService -ServiceName $serviceName -Location $location
```

Next, depending on your scenario, create the Azure VM instance from either the OS Image or OS Disk that you registered.

Generalized Operating System VHD to create multiple Azure VM instances

Create the one or more new DS Series Azure VM instances using the **Azure OS Image** that you registered. Specify this OS Image name in the VM configuration when creating new VM as shown below.

```
$OSImage = Get-AzureVMImage -ImageName "OSImageName"  
  
$vm = New-AzureVMConfig -Name $vmName -InstanceSize $vmSize -ImageName $OSImage.ImageName  
  
Add-AzureProvisioningConfig -Windows -AdminUserName $adminUser -Password $adminPassword -VM $vm  
  
New-AzureVM -ServiceName $serviceName -VM $vm
```

Unique Operating System VHD to create a single Azure VM instance

Create a new DS series Azure VM instance using the **Azure OS Disk** that you registered. Specify this OS Disk name in the VM configuration when creating the new VM as shown below.

```
$OSDisk = Get-AzureDisk -DiskName "OSDisk"  
  
$vm = New-AzureVMConfig -Name $vmName -InstanceSize $vmSize -DiskName $OSDisk.DiskName  
  
New-AzureVM -ServiceName $serviceName -VM $vm
```

Specify other Azure VM information, such as a cloud service, region, storage account, availability set, and caching policy. Note that the VM instance must be co-located with associated operating system or data disks, so the selected cloud service, region and storage account must all be in the same location as the underlying VHDs of those disks.

Attach data disk

Lastly, if you have registered data disk VHDs, attach them to the new Premium Storage capable Azure VM.

Use following PowerShell cmdlet to attach data disk to the new VM and specify the caching policy. In example below the caching policy is set to *ReadOnly*.

```
$vm = Get-AzureVM -ServiceName $serviceName -Name $vmName  
  
Add-AzureDataDisk -ImportFrom -DiskName "DataDisk" -LUN 0 -HostCaching ReadOnly -VM $vm  
  
Update-AzureVM -VM $vm
```

NOTE

There may be additional steps necessary to support your application that is not be covered by this guide.

Checking and plan backup

Once the new VM is up and running, access it using the same login id and password is as the original VM, and verify that everything is working as expected. All the settings, including the striped volumes, would be present in the new VM.

The last step is to plan backup and maintenance schedule for the new VM based on the application's needs.

A sample migration script

If you have multiple VMs to migrate, automation through PowerShell scripts will be helpful. Following is a sample script that automates the migration of a VM. Note that below script is only an example, and there are few assumptions made about the current VM disks. You may need to update the script to match with your specific scenario.

The assumptions are:

- You are creating classic Azure VMs.
- Your source OS Disks and source Data Disks are in same storage account and same container. If your OS Disks and Data Disks are not in the same place, you can use AzCopy or Azure PowerShell to copy VHDs over storage accounts and containers. Refer to the previous step: [Copy VHD with AzCopy or PowerShell](#). Editing this script to meet your scenario is another choice, but we recommend using AzCopy or PowerShell since it is easier and faster.

The automation script is provided below. Replace text with your information and update the script to match with your specific scenario.

NOTE

Using the existing script does not preserve the network configuration of your source VM. You will need to re-configure the networking settings on your migrated VMs.

```
<#  
.Synopsis  
This script is provided as an EXAMPLE to show how to migrate a VM from a standard storage account to a  
premium storage account. You can customize it according to your specific requirements.
```

.Description

The script will copy the vhds (page blobs) of the source VM to the new storage account. And then it will create a new VM from these copied vhds based on the inputs that you specified for the new VM.

You can modify the script to satisfy your specific requirement, but please be aware of the items specified in the Terms of Use section.

.Terms of Use

Copyright © 2015 Microsoft Corporation. All rights reserved.

THIS CODE AND ANY ASSOCIATED INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK OF USE, INABILITY TO USE, OR RESULTS FROM THE USE OF THIS CODE REMAINS WITH THE USER.

.Example (Save this script as Migrate-AzureVM.ps1)

```
.\Migrate-AzureVM.ps1 -SourceServiceName CurrentServiceName -SourceVMName CurrentVMName -DestStorageAccount newpremiumstorageaccount -DestServiceName NewServiceName -DestVMName NewDSVMName -DestVMSize "Standard_DS2" - Location "Southeast Asia"
```

.Link

To find more information about how to set up Azure PowerShell, refer to the following links.

<http://azure.microsoft.com/documentation/articles/powershell-install-configure/>

<http://azure.microsoft.com/documentation/articles/storage-powershell-guide-full/>

<http://azure.microsoft.com/blog/2014/10/22/migrate-azure-virtual-machines-between-storage-accounts/>

#>

```
Param(  
    # the cloud service name of the VM.  
    [Parameter(Mandatory = $true)]  
    [String] $SourceServiceName,
```

```
    # The VM name to copy.  
    [Parameter(Mandatory = $true)]  
    [String] $SourceVMName,
```

```
    # The destination storage account name.  
    [Parameter(Mandatory = $true)]  
    [String] $DestStorageAccount,
```

```
    # The destination cloud service name  
    [Parameter(Mandatory = $true)]  
    [String] $DestServiceName,
```

```
    # the destination vm name  
    [Parameter(Mandatory = $true)]  
    [String] $DestVMName,
```

```
    # the destination vm size  
    [Parameter(Mandatory = $true)]  
    [String] $DestVMSize,
```

```
    # the location of destination VM.  
    [Parameter(Mandatory = $true)]  
    [String] $Location,
```

```
    # whether or not to copy the os disk, the default is only copy data disks  
    [Parameter(Mandatory = $false)]  
    [Bool] $DataDiskOnly = $true,
```

```
    # how frequently to report the copy status in sceonds  
    [Parameter(Mandatory = $false)]  
    [Int32] $CopyStatusReportInterval = 15,
```

```
    # the name suffix to add to new created disks to avoid conflict with source disk names  
    [Parameter(Mandatory = $false)]
```

```

[String]$DiskNameSuffix = "-prem"

) #end param

#####
# Verify Azure PowerShell module and version
#####

#import the Azure PowerShell module
Write-Host "`n[WORKITEM] - Importing Azure PowerShell module" -ForegroundColor Yellow
$azureModule = Import-Module Azure -PassThru

if ($azureModule -ne $null)
{
    Write-Host "`tSuccess" -ForegroundColor Green
}
else
{
    #show module not found interaction and bail out
    Write-Host "[ERROR] - PowerShell module not found. Exiting." -ForegroundColor Red
    Exit
}

#Check the Azure PowerShell module version
Write-Host "`n[WORKITEM] - Checking Azure PowerShell module verion" -ForegroundColor Yellow
If ($azureModule.Version -ge (New-Object System.Version -ArgumentList "0.8.14"))
{
    Write-Host "`tSuccess" -ForegroundColor Green
}
Else
{
    Write-Host "[ERROR] - Azure PowerShell module must be version 0.8.14 or higher. Exiting." -
ForegroundColor Red
    Exit
}

#Check if there is an azure subscription set up in PowerShell
Write-Host "`n[WORKITEM] - Checking Azure Subscription" -ForegroundColor Yellow
$currentSubs = Get-AzureSubscription -Current
if ($currentSubs -ne $null)
{
    Write-Host "`tSuccess" -ForegroundColor Green
    Write-Host "`tYour current azure subscription in PowerShell is $($currentSubs.SubscriptionName)." -
ForegroundColor Green
}
else
{
    Write-Host "[ERROR] - There is no valid Azure subscription found in PowerShell. Please refer to this
article http://azure.microsoft.com/documentation/articles/powershell-install-configure/ to connect an Azure
subscription. Exiting." -ForegroundColor Red
    Exit
}

#####
# Check if the VM is shut down
# Stopping the VM is a required step so that the file system is consistent when you do the copy operation.
# Azure does not support live migration at this time..
#####

if (($sourceVM = Get-AzureVM -ServiceName $SourceServiceName -Name $SourceVMName) -eq $null)
{
    Write-Host "[ERROR] - The source VM doesn't exist in the current subscription. Exiting." -
ForegroundColor Red
    Exit
}

# check if VM is shut down

```

```

if ( $sourceVM.Status -notmatch "Stopped" )
{
    Write-Host "[Warning] - Stopping the VM is a required step so that the file system is consistent when you do the copy operation. Azure does not support live migration at this time. If you'd like to create a VM from a generalized image, sys-prep the Virtual Machine before stopping it." -ForegroundColor Yellow
    $ContinueAnswer = Read-Host "`n`tDo you wish to stop $SourceVMName now? Input 'N' if you want to shut down the VM manually and come back later.(Y/N)"
    If ($ContinueAnswer -ne "Y") { Write-Host "`n Exiting." -ForegroundColor Red;Exit }
    $sourceVM | Stop-AzureVM

    # wait until the VM is shut down
    $VMStatus = (Get-AzureVM -ServiceName $SourceServiceName -Name $vmName).Status
    while ($VMStatus -notmatch "Stopped")
    {
        Write-Host "`n[Status] - Waiting VM $vmName to shut down" -ForegroundColor Green
        Sleep -Seconds 5
        $VMStatus = (Get-AzureVM -ServiceName $SourceServiceName -Name $vmName).Status
    }
}

# exporting the source vm to a configuration file, you can restore the original VM by importing this config file
# see more information for Import-AzureVM
$workingDir = (Get-Location).Path
$vmConfigurationPath = $env:HOMEPATH + "\VM-" + $SourceVMName + ".xml"
Write-Host "`n[WORKITEM] - Exporting VM configuration to $vmConfigurationPath" -ForegroundColor Yellow
$exportRe = $sourceVM | Export-AzureVM -Path $vmConfigurationPath

#####
# Copy the vhds of the source vm
# You can choose to copy all disks including os and data disks by specifying the
# parameter -DataDiskOnly to be $false. The default is to copy only data disk vhds
# and the new VM will boot from the original os disk.
#####

$sourceOSDisk = $sourceVM.VM.OSVirtualHardDisk
$sourceDataDisks = $sourceVM.VM.DataVirtualHardDisks

# Get source storage account information, not considering the data disks and os disks are in different accounts
$sourceStorageAccountName = $sourceOSDisk.MediaLink.Host -split "\." | select -First 1
$sourceStorageKey = (Get-AzureStorageKey -StorageAccountName $sourceStorageAccountName).Primary
$sourceContext = New-AzureStorageContext -StorageAccountName $sourceStorageAccountName -StorageAccountKey $sourceStorageKey

# Create destination context
$destStorageKey = (Get-AzureStorageKey -StorageAccountName $DestStorageAccount).Primary
$destContext = New-AzureStorageContext -StorageAccountName $DestStorageAccount -StorageAccountKey $destStorageKey

# Create a container of vhds if it doesn't exist
if ((Get-AzureStorageContainer -Context $destContext -Name vhds -ErrorAction SilentlyContinue) -eq $null)
{
    Write-Host "`n[WORKITEM] - Creating a container vhds in the destination storage account." -ForegroundColor Yellow
    New-AzureStorageContainer -Context $destContext -Name vhds
}

$allDisksToCopy = $sourceDataDisks
# check if need to copy os disk
$sourceOSVHD = $sourceOSDisk.MediaLink.Segments[2]
if ($DataDiskOnly)
{
    # copy data disks only, this option requires deleting the source VM so that dest VM can boot
    # from the same vhd blob.
    $ContinueAnswer = Read-Host "`n`t[Warning] You chose to copy data disks only. Moving VM requires removing the original VM (the disks and backing vhd files will NOT be deleted) so that the new VM can boot from

```

```

the same vhd. This is an irreversible action. Do you wish to proceed right now? (Y/N)"
If ($ContinueAnswer -ne "Y") { Write-Host "`n Exiting." -ForegroundColor Red;Exit }
$destOSVHD = Get-AzureStorageBlob -Blob $sourceOSVHD -Container vhds -Context $sourceContext
Write-Host "`n[WORKITEM] - Removing the original VM (the vhd files are NOT deleted)." -ForegroundColor Yellow
Remove-AzureVM -Name $SourceVMName -ServiceName $SourceServiceName

Write-Host "`n[WORKITEM] - Waiting util the OS disk is released by source VM. This may take up to several minutes."
$diskAttachedTo = (Get-AzureDisk -DiskName $sourceOSDisk.DiskName).AttachedTo
while ($diskAttachedTo -ne $null)
{
    Start-Sleep -Seconds 10
    $diskAttachedTo = (Get-AzureDisk -DiskName $sourceOSDisk.DiskName).AttachedTo
}

}
else
{
    # copy the os disk vhd
    Write-Host "`n[WORKITEM] - Starting copying os disk $($disk.DiskName) at $(get-date)." -ForegroundColor Yellow
    $allDisksToCopy += @($sourceOSDisk)
    $targetBlob = Start-AzureStorageBlobCopy -SrcContainer vhds -SrcBlob $sourceOSVHD -DestContainer vhds -DestBlob $sourceOSVHD -Context $sourceContext -DestContext $destContext -Force
    $destOSVHD = $targetBlob
}

# Copy all data disk vhds
# Start all async copy requests in parallel.
foreach($disk in $sourceDataDisks)
{
    $blobName = $disk.MediaLink.Segments[2]
    # copy all data disks
    Write-Host "`n[WORKITEM] - Starting copying data disk $($disk.DiskName) at $(get-date)." -ForegroundColor Yellow
    $targetBlob = Start-AzureStorageBlobCopy -SrcContainer vhds -SrcBlob $blobName -DestContainer vhds -DestBlob $blobName -Context $sourceContext -DestContext $destContext -Force
    # update the media link to point to the target blob link
    $disk.MediaLink = $targetBlob.ICloudBlob.Uri.AbsoluteUri
}

# Wait until all vhd files are copied.
$diskComplete = @()
do
{
    Write-Host "`n[WORKITEM] - Waiting for all disk copy to complete. Checking status every $CopyStatusReportInterval seconds." -ForegroundColor Yellow
    # check status every 30 seconds
    Sleep -Seconds $CopyStatusReportInterval
    foreach ( $disk in $allDisksToCopy)
    {
        if ($diskComplete -contains $disk)
        {
            Continue
        }
        $blobName = $disk.MediaLink.Segments[2]
        $copyState = Get-AzureStorageBlobCopyState -Blob $blobName -Container vhds -Context $destContext
        if ($copyState.Status -eq "Success")
        {
            Write-Host "`n[Status] - Success for disk copy $($disk.DiskName) at $($copyState.CompletionTime)" -ForegroundColor Green
            $diskComplete += $disk
        }
        else
        {
            if ($copyState.TotalBytes -gt 0)
            {

```

```

        $percent = ($copyState.BytesCopied / $copyState.TotalBytes) * 100
        Write-Host "`n[Status] - $($'{0:N2}' -f $percent)% Complete for disk copy $($disk.DiskName)"
-ForegroundColor Green
    }
}
}

while($diskComplete.Count -lt $allDisksToCopy.Count)

#####
# Create a new vm
# the new VM can be created from the copied disks or the original os disk.
# You can add your own logic here to satisfy your specific requirements of the vm.
#####

# Create a VM from the existing os disk
if ($DataDiskOnly)
{
    $vm = New-AzureVMConfig -Name $DestVMName -InstanceSize $DestVMSize -DiskName $sourceOSDisk.DiskName
}
else
{
    $newOSDisk = Add-AzureDisk -OS $sourceOSDisk.OS -DiskName ($sourceOSDisk.DiskName + $DiskNameSuffix) -
MediaLocation $destOSVHD.ICloudBlob.Uri.AbsoluteUri
    $vm = New-AzureVMConfig -Name $DestVMName -InstanceSize $DestVMSize -DiskName $newOSDisk.DiskName
}
# Attached the copied data disks to the new VM
foreach ($dataDisk in $sourceDataDisks)
{
    # add -DiskLabel $dataDisk.DiskLabel if there are labels for disks of the source vm
    $diskLabel = "drive" + $dataDisk.Lun
    $vm | Add-AzureDataDisk -ImportFrom -DiskLabel $diskLabel -LUN $dataDisk.Lun -MediaLocation
$dataDisk.MediaLink
}

# Edit this if you want to add more customization to the new VM
# $vm | Add-AzureEndpoint -Protocol tcp -LocalPort 443 -PublicPort 443 -Name 'HTTPs'
# $vm | Set-AzureSubnet "PubSubnet","PrivSubnet"

New-AzureVM -ServiceName $DestServiceName -VMs $vm -Location $Location

```

Optimization

Your current VM configuration may be customized specifically to work well with Standard disks. For instance, to increase the performance by using many disks in a striped volume. For example, instead of using 4 disks separately on Premium Storage, you may be able to optimize the cost by having a single disk. Optimizations like this need to be handled on a case by case basis and require custom steps after the migration. Also, note that this process may not well work for databases and applications that depend on the disk layout defined in the setup.

Preparation

1. Complete the Simple Migration as described in the earlier section. Optimizations will be performed on the new VM after the migration.
2. Define the new disk sizes needed for the optimized configuration.
3. Determine mapping of the current disks/volumes to the new disk specifications.

Execution steps

1. Create the new disks with the right sizes on the Premium Storage VM.
2. Login to the VM and copy the data from the current volume to the new disk that maps to that volume. Do this for all the current volumes that need to map to a new disk.
3. Next, change the application settings to switch to the new disks, and detach the old volumes.

For tuning the application for better disk performance, please refer to [Optimizing Application Performance](#).

Application migrations

Databases and other complex applications may require special steps as defined by the application provider for the migration. Please refer to respective application documentation. E.g. typically databases can be migrated through backup and restore.

Next steps

See the following resources for specific scenarios for migrating virtual machines:

- [Migrate Azure Virtual Machines between Storage Accounts](#)
- [Create and upload a Windows Server VHD to Azure.](#)
- [Creating and Uploading a Virtual Hard Disk that Contains the Linux Operating System](#)
- [Migrating Virtual Machines from Amazon AWS to Microsoft Azure](#)

Also, see the following resources to learn more about Azure Storage and Azure Virtual Machines:

- [Azure Storage](#)
- [Azure Virtual Machines](#)
- [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#)

Azure Premium Storage: Design for High Performance

1/17/2017 • 40 min to read • [Edit on GitHub](#)

Overview

This article provides guidelines for building high performance applications using Azure Premium Storage. You can use the instructions provided in this document combined with performance best practices applicable to technologies used by your application. To illustrate the guidelines, we have used SQL Server running on Premium Storage as an example throughout this document.

While we address performance scenarios for the Storage layer in this article, you will need to optimize the application layer. For example, if you are hosting a SharePoint Farm on Azure Premium Storage, you can use the SQL Server examples from this article to optimize the database server. Additionally, optimize the SharePoint Farm's Web server and Application server to get the most performance.

This article will help answer following common questions about optimizing application performance on Azure Premium Storage,

- How to measure your application performance?
- Why are you not seeing expected high performance?
- Which factors influence your application performance on Premium Storage?
- How do these factors influence performance of your application on Premium Storage?
- How can you optimize for IOPS, Bandwidth and Latency?

We have provided these guidelines specifically for Premium Storage because workloads running on Premium Storage are highly performance sensitive. We have provided examples where appropriate. You can also apply some of these guidelines to applications running on IaaS VMs with Standard Storage disks.

Before you begin, if you are new to Premium Storage, first read the [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#) article and the [Azure Premium Storage Scalability and Performance Targets](#).

Application Performance Indicators

We assess whether an application is performing well or not using performance indicators like, how fast an application is processing a user request, how much data an application is processing per request, how many requests an application processing in a specific period of time, how long a user has to wait to get a response after submitting their request. The technical terms for these performance indicators are, IOPS, Throughput or Bandwidth, and Latency.

In this section, we will discuss the common performance indicators in the context of Premium Storage. In the following section, Gathering Application Requirements, you will learn how to measure these performance indicators for your application. Later in Optimizing Application Performance, you will learn about the factors affecting these performance indicators and recommendations to optimize them.

IOPS

IOPS is number of requests that your application is sending to the storage disks in one second. An input/output operation could be read or write, sequential or random. OLTP applications like an online retail website need to process many concurrent user requests immediately. The user requests are insert and update intensive database

transactions, which the application must process quickly. Therefore, OLTP applications require very high IOPS. Such applications handle millions of small and random IO requests. If you have such an application, you must design the application infrastructure to optimize for IOPS. In the later section, *Optimizing Application Performance*, we discuss in detail all the factors that you must consider to get high IOPS.

When you attach a premium storage disk to your high scale VM, Azure provisions for you a guaranteed number of IOPS as per the disk specification. For example, a P30 disk provisions 5000 IOPS. Each high scale VM size also has a specific IOPS limit that it can sustain. For example, a Standard GS5 VM has 80,000 IOPS limit.

Throughput

Throughput or Bandwidth is the amount of data that your application is sending to the storage disks in a specified interval. If your application is performing input/output operations with large IO unit sizes, it requires high Throughput. Data warehouse applications tend to issue scan intensive operations that access large portions of data at a time and commonly perform bulk operations. In other words, such applications require higher Throughput. If you have such an application, you must design its infrastructure to optimize for Throughput. In the next section, we discuss in detail the factors you must tune to achieve this.

When you attach a premium storage disk to a high scale VM, Azure provisions Throughput as per that disk specification. For example, a P30 disk provisions 200 MB per second disk Throughput. Each high scale VM size also has a specific Throughput limit that it can sustain. For example, Standard GS5 VM has a maximum throughput of 2,000 MB per second.

There is a relation between Throughput and IOPS as shown in the formula below.

$$\text{IOPS} \times \text{IO Size} = \text{Throughput}$$

Therefore, it is important to determine the optimal Throughput and IOPS values that your application requires. As you try to optimize one, the other also gets affected. In a later section, *Optimizing Application Performance*, we will discuss in more details about optimizing IOPS and Throughput.

Latency

Latency is the time it takes an application to receive a single request, send it to the storage disks and send the response to the client. This is a critical measure of an application's performance in addition to IOPS and Throughput. The Latency of a premium storage disk is the time it takes to retrieve the information for a request and communicate it back to your application. Premium Storage provides consistent low latencies. If you enable ReadOnly host caching on premium storage disks, you can get much lower read latency. We will discuss Disk Caching in more detail in later section on *Optimizing Application Performance*.

When you are optimizing your application to get higher IOPS and Throughput, it will affect the Latency of your application. After tuning the application performance, always evaluate the Latency of the application to avoid unexpected high latency behavior.

Gather Application Performance Requirements

The first step in designing high performance applications running on Azure Premium Storage is, to understand the performance requirements of your application. After you gather performance requirements, you can optimize your application to achieve the most optimal performance.

In the previous section, we explained the common performance indicators, IOPS, Throughput and Latency. You must identify which of these performance indicators are critical to your application to deliver the desired user experience. For example, high IOPS matters most to OLTP applications processing millions of transactions in a

second. Whereas, high Throughput is critical for Data Warehouse applications processing large amounts of data in a second. Extremely low Latency is crucial for real-time applications like live video streaming websites.

Next, measure the maximum performance requirements of your application throughout its lifetime. Use the sample checklist below as a start. Record the maximum performance requirements during normal, peak and off-hours workload periods. By identifying requirements for all workloads levels, you will be able to determine the overall performance requirement of your application. For example, the normal workload of an e-commerce website will be the transactions it serves during most days in a year. The peak workload of the website will be the transactions it serves during holiday season or special sale events. The peak workload is typically experienced for a limited period, but can require your application to scale two or more times its normal operation. Find out the 50 percentile, 90 percentile and 99 percentile requirements. This helps filter out any outliers in the performance requirements and you can focus your efforts on optimizing for the right values.

Application Performance Requirements Checklist

PERFORMANCE REQUIREMENTS	50 PERCENTILE	90 PERCENTILE	99 PERCENTILE
Max. Transactions per second			
% Read operations			
% Write operations			
% Random operations			
% Sequential operations			
IO request size			
Average Throughput			
Max. Throughput			
Min. Latency			
Average Latency			
Max. CPU			
Average CPU			
Max. Memory			
Average Memory			
Queue Depth			

NOTE

You should consider scaling these numbers based on expected future growth of your application. It is a good idea to plan for growth ahead of time, because it could be harder to change the infrastructure for improving performance later.

If you have an existing application and want to move to Premium Storage, first build the checklist above for the existing application. Then, build a prototype of your application on Premium Storage and design the application based on guidelines described in *Optimizing Application Performance* in a later section of this document. The next section describes the tools you can use to gather the performance measurements.

Create a checklist similar to your existing application for the prototype. Using Benchmarking tools you can simulate the workloads and measure performance on the prototype application. See the section on [Benchmarking](#) to learn more. By doing so you can determine whether Premium Storage can match or surpass your application performance requirements. Then you can implement the same guidelines for your production application.

Counters to measure application performance requirements

The best way to measure performance requirements of your application, is to use performance-monitoring tools provided by the operating system of the server. You can use PerfMon for Windows and iostat for Linux. These tools capture counters corresponding to each measure explained in the above section. You must capture the values of these counters when your application is running its normal, peak and off-hours workloads.

The PerfMon counters are available for processor, memory and, each logical disk and physical disk of your server. When you use premium storage disks with a VM, the physical disk counters are for each premium storage disk, and logical disk counters are for each volume created on the premium storage disks. You must capture the values for the disks that host your application workload. If there is a one to one mapping between logical and physical disks, you can refer to physical disk counters; otherwise refer to the logical disk counters. On Linux, the iostat command generates a CPU and disk utilization report. The disk utilization report provides statistics per physical device or partition. If you have a database server with its data and log on separate disks, collect this data for both disks. Below table describes counters for disks, processor and memory:

COUNTER	DESCRIPTION	PERFMON	IOSTAT
IOPS or Transactions per second	Number of I/O requests issued to the storage disk per second.	Disk Reads/sec Disk Writes/sec	tps r/s w/s
Disk Reads and Writes	% of Reads and Write operations performed on the disk.	% Disk Read Time % Disk Write Time	r/s w/s
Throughput	Amount of data read from or written to the disk per second.	Disk Read Bytes/sec Disk Write Bytes/sec	kB_read/s kB_wrtn/s
Latency	Total time to complete a disk IO request.	Average Disk sec/Read Average disk sec/Write	await svctm
IO size	The size of I/O requests issues to the storage disks.	Average Disk Bytes/Read Average Disk Bytes/Write	avgrq-sz
Queue Depth	Number of outstanding I/O requests waiting to be read form or written to the storage disk.	Current Disk Queue Length	avgqu-sz
Max. Memory	Amount of memory required to run application smoothly	% Committed Bytes in Use	Use vmstat
Max. CPU	Amount CPU required to run application smoothly	% Processor time	%util

Learn more about [iostat](#) and [PerfMon](#).

Optimizing Application Performance

The main factors that influence performance of an application running on Premium Storage are Nature of IO Requests, VM size, Disk size, Number of disks, Disk Caching, Multithreading and Queue Depth. You can control some of these factors with knobs provided by the system. Most applications may not give you an option to alter the IO size and Queue Depth directly. For example, if you are using SQL Server, you cannot choose the IO size and queue depth. SQL Server chooses the optimal IO size and queue depth values to get the most performance. It is important to understand the effects of both types of factors on your application performance, so that you can provision appropriate resources to meet performance needs.

Throughout this section, refer to the application requirements checklist that you created, to identify how much you need to optimize your application performance. Based on that, you will be able to determine which factors from this section you will need to tune. To witness the effects of each factor on your application performance, run benchmarking tools on your application setup. Refer to the [Benchmarking](#) section at the end of this article for steps to run common benchmarking tools on Windows and Linux VMs.

Optimizing IOPS, Throughput and Latency at a glance

The table below summarizes all the performance factors and the steps to optimize IOPS, Throughput and Latency. The sections following this summary will describe each factor in much more depth.

	IOPS	THROUGHPUT	LATENCY
Example Scenario	Enterprise OLTP application requiring very high transactions per second rate.	Enterprise Data warehousing application processing large amounts of data.	Near real-time applications requiring instant responses to user requests, like online gaming.
Performance factors			
IO size	Smaller IO size yields higher IOPS.	Larger IO size yields higher Throughput.	
VM size	Use a VM size that offers IOPS greater than your application requirement. See VM sizes and their IOPS limits here.	Use a VM size with Throughput limit greater than your application requirement. See VM sizes and their Throughput limits here.	Use a VM size that offers scale limits greater than your application requirement. See VM sizes and their limits here.
Disk size	Use a disk size that offers IOPS greater than your application requirement. See disk sizes and their IOPS limits here.	Use a disk size with Throughput limit greater than your application requirement. See disk sizes and their Throughput limits here.	Use a disk size that offers scale limits greater than your application requirement. See disk sizes and their limits here.
VM and Disk Scale Limits	IOPS limit of the VM size chosen should be greater than total IOPS driven by premium storage disks attached to it.	Throughput limit of the VM size chosen should be greater than total Throughput driven by premium storage disks attached to it.	Scale limits of the VM size chosen must be greater than total scale limits of attached premium storage disks.

	IOPS	THROUGHPUT	LATENCY
Disk Caching	Enable ReadOnly Cache on premium storage disks with Read heavy operations to get higher Read IOPS.		Enable ReadOnly Cache on premium storage disks with Ready heavy operations to get very low Read latencies.
Disk Striping	Use multiple disks and stripe them together to get a combined higher IOPS and Throughput limit. Note that the combined limit per VM should be higher than the combined limits of attached premium disks.		
Stripe Size	Smaller stripe size for random small IO pattern seen in OLTP applications. E.g., use stripe size of 64KB for SQL Server OLTP application.	Larger stripe size for sequential large IO pattern seen in Data Warehouse applications. E.g., use 256KB stripe size for SQL Server Data warehouse application.	
Multithreading	Use multithreading to push higher number of requests to Premium Storage that will lead to higher IOPS and Throughput. For example, on SQL Server set a high MAXDOP value to allocate more CPUs to SQL Server.		
Queue Depth	Larger Queue Depth yields higher IOPS.	Larger Queue Depth yields higher Throughput.	Smaller Queue Depth yields lower latencies.

Nature of IO Requests

An IO request is a unit of input/output operation that your application will be performing. Identifying the nature of IO requests, random or sequential, read or write, small or large, will help you determine the performance requirements of your application. It is very important to understand the nature of IO requests, to make the right decisions when designing your application infrastructure.

IO size is one of the more important factors. The IO size is the size of the input/output operation request generated by your application. The IO size has a significant impact on performance especially on the IOPS and Bandwidth that the application is able to achieve. The following formula shows the relationship between IOPS, IO size and Bandwidth/Throughput.

$$\text{IOPS} \times \text{IO Size} = \text{Throughput}$$

Some applications allow you to alter their IO size, while some applications do not. For example, SQL Server determines the optimal IO size itself, and does not provide users with any knobs to change it. On the other hand, Oracle provides a parameter called `DB_BLOCK_SIZE` using which you can configure the I/O request size of the database.

If you are using an application, which does not allow you to change the IO size, use the guidelines in this article to optimize the performance KPI that is most relevant to your application. For example,

- An OLTP application generates millions of small and random IO requests. To handle these type of IO requests, you must design your application infrastructure to get higher IOPS.
- A data warehousing application generates large and sequential IO requests. To handle these type of IO requests, you must design your application infrastructure to get higher Bandwidth or Throughput.

If you are using an application, which allows you to change the IO size, use this rule of thumb for the IO size in addition to other performance guidelines,

- Smaller IO size to get higher IOPS. For example, 8 KB for an OLTP application.
- Larger IO size to get higher Bandwidth/Throughput. For example, 1024 KB for a data warehouse application.

Here is an example on how you can calculate the IOPS and Throughput/Bandwidth for your application. Consider an application using a P30 disk. The maximum IOPS and Throughput/Bandwidth a P30 disk can achieve is 5000 IOPS and 200 MB per second respectively. Now, if your application requires the maximum IOPS from the P30 disk and you use a smaller IO size like 8 KB, the resulting Bandwidth you will be able to get is 40 MB per second. However, if your application requires the maximum Throughput/Bandwidth from P30 disk, and you use a larger IO size like 1024 KB, the resulting IOPS will be less, 200 IOPS. Therefore, tune the IO size such that it meets both your application's IOPS and Throughput/Bandwidth requirement. Table below summarizes the different IO sizes and their corresponding IOPS and Throughput for a P30 disk.

APPLICATION REQUIREMENT	I/O SIZE	IOPS	THROUGHPUT/BANDWIDTH
Max IOPS	8 KB	5,000	40 MB per second
Max Throughput	1024 KB	200	200 MB per second
Max Throughput + high IOPS	64 KB	3,200	200 MB per second
Max IOPS + high Throughput	32 KB	5,000	160 MB per second

To get IOPS and Bandwidth higher than the maximum value of a single premium storage disk, use multiple premium disks striped together. For example, stripe two P30 disks to get a combined IOPS of 10,000 IOPS or a combined Throughput of 400 MB per second. As explained in the next section, you must use a VM size that supports the combined disk IOPS and Throughput.

NOTE

As you increase either IOPS or Throughput the other also increases, make sure you do not hit throughput or IOPS limits of the disk or VM when increasing either one.

To witness the effects of IO size on application performance, you can run benchmarking tools on your VM and disks. Create multiple test runs and use different IO size for each run to see the impact. Refer to the [Benchmarking](#) section at the end of this article for more details.

High Scale VM Sizes

When you start designing an application, one of the first things to do is, choose a VM to host your application. Premium Storage comes with High Scale VM sizes that can run applications requiring higher compute power and a high local disk I/O performance. These VMs provide faster processors, a higher memory-to-core ratio, and a Solid-State Drive (SSD) for the local disk. Examples of High Scale VMs supporting Premium Storage are the DS, DSv2 and GS series VMs.

High Scale VMs are available in different sizes with a different number of CPU cores, memory, OS and temporary disk size. Each VM size also has maximum number of data disks that you can attach to the VM. Therefore, the chosen VM size will affect how much processing, memory, and storage capacity is available for your application. It also affects the Compute and Storage cost. For example, below are the specifications of the largest VM size in a DS series, DSv2 series and a GS series:

VM SIZE	CPU CORES	MEMORY	VM DISK SIZES	MAX. DATA DISKS	CACHE SIZE	IOPS	BANDWIDTH CACHE IO LIMITS
Standard_D S14	16	112 GB	OS = 1023 GB Local SSD = 224 GB	32	576 GB	50,000 IOPS 512 MB per second	4,000 IOPS and 33 MB per second
Standard_G S5	32	448 GB	OS = 1023 GB Local SSD = 896 GB	64	4224 GB	80,000 IOPS 2,000 MB per second	5,000 IOPS and 50 MB per second

To view a complete list of all available Azure VM sizes, refer to [Windows VM sizes](#) or [Linux VM sizes](#). Choose a VM size that can meet and scale to your desired application performance requirements. In addition to this, take into account following important considerations when choosing VM sizes.

Scale Limits

The maximum IOPS limits per VM and per disk are different and independent of each other. Make sure that the application is driving IOPS within the limits of the VM as well as the premium disks attached to it. Otherwise, application performance will experience throttling.

As an example, suppose an application requirement is a maximum of 4,000 IOPS. To achieve this, you provision a P30 disk on a DS1 VM. The P30 disk can deliver up to 5,000 IOPS. However, the DS1 VM is limited to 3,200 IOPS. Consequently, the application performance will be constrained by the VM limit at 3,200 IOPS and there will be degraded performance. To prevent this situation, choose a VM and disk size that will both meet application requirements.

Cost of Operation

In many cases, it is possible that your overall cost of operation using Premium Storage is lower than using Standard Storage.

For example, consider an application requiring 16,000 IOPS. To achieve this performance, you will need a Standard_D14 Azure IaaS VM, which can give a maximum IOPS of 16,000 using 32 standard storage 1TB disks. Each 1TB standard storage disk can achieve a maximum of 500 IOPS. The estimated cost of this VM per month will be \$1,570. The monthly cost of 32 standard storage disks will be \$1,638. The estimated total monthly cost will be \$3,208.

However, if you hosted the same application on Premium Storage, you will need a smaller VM size and fewer premium storage disks, thus reducing the overall cost. A Standard_DS13 VM can meet the 16,000 IOPS requirement using four P30 disks. The DS13 VM has a maximum IOPS of 25,600 and each P30 disk has a maximum IOPS of 5,000. Overall, this configuration can achieve $5,000 \times 4 = 20,000$ IOPS. The estimated cost of this VM per month will be \$1,003. The monthly cost of four P30 premium storage disks will be \$544.34. The estimated total monthly cost will be \$1,544.

Table below summarizes the cost breakdown of this scenario for Standard and Premium Storage.

	STANDARD	PREMIUM
Cost of VM per month	\$1,570.58 (Standard_D14)	\$1,003.66 (Standard_DS13)
Cost of Disks per month	\$1,638.40 (32 x 1 TB disks)	\$544.34 (4 x P30 disks)
Overall Cost per month	\$3,208.98	\$1,544.34

Linux Distros

With Azure Premium Storage, you get the same level of Performance for VMs running Windows and Linux. We support many flavors of Linux distros, and you can see the complete list [here](#). It is important to note that different distros are better suited for different types of workloads. You will see different levels of performance depending on the distro your workload is running on. Test the Linux distros with your application and choose the one that works best.

When running Linux with Premium Storage, check the latest updates about required drivers to ensure high performance.

Premium Storage Disk Sizes

Azure Premium Storage offers three disk sizes currently. Each disk size has a different scale limit for IOPS, Bandwidth and Storage. Choose the right Premium Storage Disk size depending on the application requirements and the high scale VM size. The table below shows the three disks sizes and their capabilities.

DISK TYPE	P10	P20	P30
Disk Size	128 GiB	512 GiB	1024 GiB (1 TB)
IOPS per disk	500	2300	5000
Throughput per disk	100 MB per second	150 MB per second	200 MB per second

How many disks you choose depends on the disk size chosen. You could use a single P30 disk or multiple P10 disks to meet your application requirement. Take into account considerations listed below when making the choice.

Scale Limits (IOPS and Throughput)

The IOPS and Throughput limits of each Premium disk size is different and independent from the VM scale limits. Make sure that the total IOPS and Throughput from the disks are within scale limits of the chosen VM size.

For example, if an application requirement is a maximum of 250 MB/sec Throughput and you are using a DS4 VM with a single P30 disk. The DS4 VM can give up to 256 MB/sec Throughput. However, a single P30 disk has Throughput limit of 200 MB/sec. Consequently, the application will be constrained at 200 MB/sec due to the disk limit. To overcome this limit, provision more than one data disks to the VM.

NOTE

Reads served by the cache are not included in the disk IOPS and Throughput, hence not subject to disk limits. Cache has its separate IOPS and Throughput limit per VM.

For example, initially your reads and writes are 60MB/sec and 40MB/sec respectively. Over time, the cache warms up and serves more and more of the reads from the cache. Then, you can get higher write Throughput from the disk.

Number of Disks

Determine the number of disks you will need by assessing application requirements. Each VM size also has a limit on the number of disks that you can attach to the VM. Typically, this is twice the number of cores. Ensure that the VM size you choose can support the number of disks needed.

Remember, the Premium Storage disks have higher performance capabilities compared to Standard Storage disks. Therefore, if you are migrating your application from Azure IaaS VM using Standard Storage to Premium Storage, you will likely need fewer premium disks to achieve the same or higher performance for your application.

Disk Caching

High Scale VMs that leverage Azure Premium Storage have a multi-tier caching technology called BlobCache. BlobCache uses a combination of the Virtual Machine RAM and local SSD for caching. This cache is available for the Premium Storage persistent disks and the VM local disks. By default, this cache setting is set to Read/Write for OS disks and ReadOnly for data disks hosted on Premium Storage. With disk caching enabled on the Premium Storage disks, the high scale VMs can achieve extremely high levels of performance that exceed the underlying disk performance.

WARNING

Changing the cache setting of an Azure disk detaches and re-attaches the target disk. If it is the operating system disk, the VM is restarted. Stop all applications/services that might be affected by this disruption before changing the disk cache setting.

To learn more about how BlobCache works, refer to the Inside [Azure Premium Storage](#) blog post.

It is important to enable cache on the right set of disks. Whether you should enable disk caching on a premium disk or not will depend on the workload pattern that disk will be handling. Table below shows the default cache settings for OS and Data disks.

DISK TYPE	DEFAULT CACHE SETTING
OS disk	ReadWrite
Data disk	None

Following are the recommended disk cache settings for data disks,

DISK CACHING SETTING	RECOMMENDATION ON WHEN TO USE THIS SETTING
None	Configure host-cache as None for write-only and write-heavy disks.
ReadOnly	Configure host-cache as ReadOnly for read-only and read-write disks.
ReadWrite	Configure host-cache as ReadWrite only if your application properly handles writing cached data to persistent disks when needed.

ReadOnly

By configuring ReadOnly caching on Premium Storage data disks, you can achieve low Read latency and get very high Read IOPS and Throughput for your application. This is due two reasons,

1. Reads performed from cache, which is on the VM memory and local SSD, are much faster than reads from the data disk, which is on the Azure blob storage.

2. Premium Storage does not count the Reads served from cache, towards the disk IOPS and Throughput. Therefore, your application is able to achieve higher total IOPS and Throughput.

ReadWrite

By default, the OS disks have ReadWrite caching enabled. We have recently added support for ReadWrite caching on data disks as well. If you are using ReadWrite caching, you must have a proper way to write the data from cache to persistent disks. For example, SQL Server handles writing cached data to the persistent storage disks on its own. Using ReadWrite cache with an application that does not handle persisting the required data can lead to data loss, if the VM crashes.

As an example, you can apply these guidelines to SQL Server running on Premium Storage by doing the following,

1. Configure "ReadOnly" cache on premium storage disks hosting data files.
 - a. The fast reads from cache lower the SQL Server query time since data pages are retrieved much faster from the cache compared to directly from the data disks.
 - b. Serving reads from cache, means there is additional Throughput available from premium data disks. SQL Server can use this additional Throughput towards retrieving more data pages and other operations like backup/restore, batch loads, and index rebuilds.
2. Configure "None" cache on premium storage disks hosting the log files.
 - a. Log files have primarily write-heavy operations. Therefore, they do not benefit from the ReadOnly cache.

Disk Striping

When a high scale VM is attached with several premium storage persistent disks, the disks can be striped together to aggregate their IOPs, bandwidth, and storage capacity.

On Windows, you can use Storage Spaces to stripe disks together. You must configure one column for each disk in a pool. Otherwise, the overall performance of striped volume can be lower than expected, due to uneven distribution of traffic across the disks.

Important: Using Server Manager UI, you can set the total number of columns up to 8 for a striped volume. When attaching more than 8 disks, use PowerShell to create the volume. Using PowerShell, you can set the number of columns equal to the number of disks. For example, if there are 16 disks in a single stripe set; specify 16 columns in the *NumberOfColumns* parameter of the *New-VirtualDisk* PowerShell cmdlet.

On Linux, use the MDADM utility to stripe disks together. For detailed steps on striping disks on Linux refer to [Configure Software RAID on Linux](#).

Stripe Size

An important configuration in disk striping is the stripe size. The stripe size or block size is the smallest chunk of data that application can address on a striped volume. The stripe size you configure depends on the type of application and its request pattern. If you choose the wrong stripe size, it could lead to IO misalignment, which leads to degraded performance of your application.

For example, if an IO request generated by your application is bigger than the disk stripe size, the storage system writes it across stripe unit boundaries on more than one disk. When it is time to access that data, it will have to seek across more than one stripe units to complete the request. The cumulative effect of such behavior can lead to substantial performance degradation. On the other hand, if the IO request size is smaller than stripe size, and if it is random in nature, the IO requests may add up on the same disk causing a bottleneck and ultimately degrading the IO performance.

Depending on the type of workload your application is running, choose an appropriate stripe size. For random small IO requests, use a smaller stripe size. Whereas, for large sequential IO requests use a larger stripe size. Find out the stripe size recommendations for the application you will be running on Premium Storage. For SQL Server, configure stripe size of 64KB for OLTP workloads and 256KB for data warehousing workloads. See [Performance best practices for SQL Server on Azure VMs](#) to learn more.

NOTE

You can stripe together a maximum of 32 premium storage disks on a DS series VM and 64 premium storage disks on a GS series VM.

Multi-threading

Azure has designed Premium Storage platform to be massively parallel. Therefore, a multi-threaded application achieves much higher performance than a single-threaded application. A multi-threaded application splits up its tasks across multiple threads and increases efficiency of its execution by utilizing the VM and disk resources to the maximum.

For example, if your application is running on a single core VM using two threads, the CPU can switch between the two threads to achieve efficiency. While one thread is waiting on a disk IO to complete, the CPU can switch to the other thread. In this way, two threads can accomplish more than a single thread would. If the VM has more than one core, it further decreases running time since each core can execute tasks in parallel.

You may not be able to change the way an off-the-shelf application implements single threading or multi-threading. For example, SQL Server is capable of handling multi-CPU and multi-core. However, SQL Server decides under what conditions it will leverage one or more threads to process a query. It can run queries and build indexes using multi-threading. For a query that involves joining large tables and sorting data before returning to the user, SQL Server will likely use multiple threads. However, a user cannot control whether SQL Server executes a query using a single thread or multiple threads.

There are configuration settings that you can alter to influence this multi-threading or parallel processing of an application. For example, in case of SQL Server it is the maximum Degree of Parallelism configuration. This setting called MAXDOP, allows you to configure the maximum number of processors SQL Server can use when parallel processing. You can configure MAXDOP for individual queries or index operations. This is beneficial when you want to balance resources of your system for a performance critical application.

For example, say your application using SQL Server is executing a large query and an index operation at the same time. Let us assume that you wanted the index operation to be more performant compared to the large query. In such a case, you can set MAXDOP value of the index operation to be higher than the MAXDOP value for the query. This way, SQL Server has more number of processors that it can leverage for the index operation compared to the number of processors it can dedicate to the large query. Remember, you do not control the number of threads SQL Server will use for each operation. You can control the maximum number of processors being dedicated for multi-threading.

Learn more about [Degrees of Parallelism](#) in SQL Server. Find out such settings that influence multi-threading in your application and their configurations to optimize performance.

Queue Depth

The Queue Depth or Queue Length or Queue Size is the number of pending IO requests in the system. The value of Queue Depth determines how many IO operations your application can line up, which the storage disks will be processing. It affects all the three application performance indicators that we discussed in this article viz., IOPS, Throughput and Latency.

Queue Depth and multi-threading are closely related. The Queue Depth value indicates how much multi-threading can be achieved by the application. If the Queue Depth is large, application can execute more operations concurrently, in other words, more multi-threading. If the Queue Depth is small, even though application is multi-threaded, it will not have enough requests lined up for concurrent execution.

Typically, off the shelf applications do not allow you to change the queue depth, because if set incorrectly it will do more harm than good. Applications will set the right value of queue depth to get the optimal performance.

However, it is important to understand this concept so that you can troubleshoot performance issues with your application. You can also observe the effects of queue depth by running benchmarking tools on your system.

Some applications provide settings to influence the Queue Depth. For example, the MAXDOP (maximum degree of parallelism) setting in SQL Server explained in previous section. MAXDOP is a way to influence Queue Depth and multi-threading, although it does not directly change the Queue Depth value of SQL Server.

High Queue Depth

A high queue depth lines up more operations on the disk. The disk knows the next request in its queue ahead of time. Consequently, the disk can schedule operations ahead of time and process them in an optimal sequence. Since the application is sending more requests to the disk, the disk can process more parallel IOs. Ultimately, the application will be able to achieve higher IOPS. Since application is processing more requests, the total Throughput of the application also increases.

Typically, an application can achieve maximum Throughput with 8-16+ outstanding IOs per attached disk. If a Queue Depth is one, application is not pushing enough IOs to the system, and it will process less amount of in a given period. In other words, less Throughput.

For example, in SQL Server, setting the MAXDOP value for a query to "4" informs SQL Server that it can use up to four cores to execute the query. SQL Server will determine what is best queue depth value and the number of cores for the query execution.

Optimal Queue Depth

Very high queue depth value also has its drawbacks. If queue depth value is too high, the application will try to drive very high IOPS. Unless application has persistent disks with sufficient provisioned IOPS, this can negatively affect application latencies. Following formula shows the relationship between IOPS, Latency and Queue Depth.

$$\text{IOPS} \times \text{Latency} = \text{Queue Depth}$$

You should not configure Queue Depth to any high value, but to an optimal value, which can deliver enough IOPS for the application without affecting latencies. For example, if the application latency needs to be 1 millisecond, the Queue Depth required to achieve 5,000 IOPS is, $\text{QD} = 5000 \times 0.001 = 5$.

Queue Depth for Striped Volume

For a striped volume, maintain a high enough queue depth such that every disk has a peak queue depth individually. For example, consider an application that pushes a queue depth of 2 and there are 4 disks in the stripe. The two IO requests will go to two disks and remaining two disks will be idle. Therefore, configure the queue depth such that all the disks can be busy. Formula below shows how to determine the queue depth of striped volumes.

$$\text{QD per Disk} \times \text{No. of Columns per Volume} = \text{QD of Striped Volume}$$

Throttling

Azure Premium Storage provisions specified number of IOPS and Throughput depending on the VM sizes and disk sizes you choose. Anytime your application tries to drive IOPS or Throughput above these limits of what the VM or disk can handle, Premium Storage will throttle it. This manifests in the form of degraded performance in your application. This can mean higher latency, lower Throughput or lower IOPS. If Premium Storage does not throttle, your application could completely fail by exceeding what its resources are capable of achieving. So, to avoid performance issues due to throttling, always provision sufficient resources for your application. Take into consideration what we discussed in the VM sizes and Disk sizes sections above. Benchmarking is the best way to figure out what resources you will need to host your application.

Benchmarking

Benchmarking is the process of simulating different workloads on your application and measuring the application performance for each workload. Using the steps described in an earlier section, you have gathered the application performance requirements. By running benchmarking tools on the VMs hosting the application, you can determine the performance levels that your application can achieve with Premium Storage. In this section, we provide you examples of benchmarking a Standard DS14 VM provisioned with Azure Premium Storage disks.

We have used common benchmarking tools lometer and FIO, for Windows and Linux respectively. These tools spawn multiple threads simulating a production like workload, and measure the system performance. Using the tools you can also configure parameters like block size and queue depth, which you normally cannot change for an application. This gives you more flexibility to drive the maximum performance on a high scale VM provisioned with premium disks for different types of application workloads. To learn more about each benchmarking tool visit [lometer](#) and [FIO](#).

To follow the examples below, create a Standard DS14 VM and attach 11 Premium Storage disks to the VM. Of the 11 disks, configure 10 disks with host caching as "None" and stripe them into a volume called NoCacheWrites. Configure host caching as "ReadOnly" on the remaining disk and create a volume called CacheReads with this disk. Using this setup, you will be able to see the maximum Read and Write performance from a Standard DS14 VM. For detailed steps about creating a DS14 VM with premium disks, go to [Create and use a Premium Storage account for a virtual machine data disk](#).

Warming up the Cache

The disk with ReadOnly host caching will be able to give higher IOPS than the disk limit. To get this maximum read performance from the host cache, first you must warm up the cache of this disk. This ensures that the Read IOs which benchmarking tool will drive on CacheReads volume actually hits the cache and not the disk directly. The cache hits result in additional IOPS from the single cache enabled disk.

Important:

You must warm up the cache before running benchmarking, every time VM is rebooted.

lometer

[Download the lometer tool](#) on the VM.

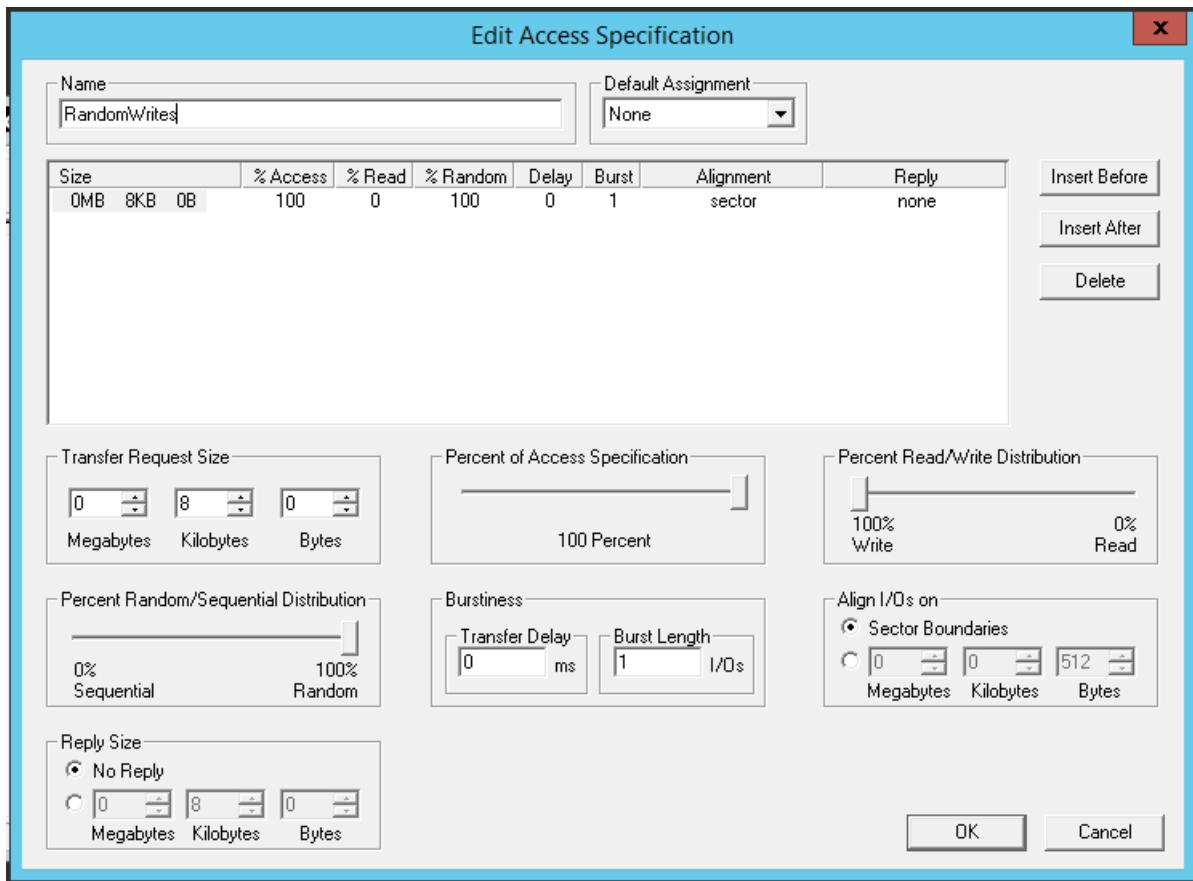
Test file

lometer uses a test file that is stored on the volume on which you will run the benchmarking test. It drives Reads and Writes on this test file to measure the disk IOPS and Throughput. lometer creates this test file if you have not provided one. Create a 200GB test file called iobw.tst on the CacheReads and NoCacheWrites volumes.

Access Specifications

The specifications, request IO size, % read/write, % random/sequential are configured using the "Access Specifications" tab in lometer. Create an access specification for each of the scenarios described below. Create the access specifications and "Save" with an appropriate name like – RandomWrites_8K, RandomReads_8K. Select the corresponding specification when running the test scenario.

An example of access specifications for maximum Write IOPS scenario is shown below,



Maximum IOPS Test Specifications

To demonstrate maximum IOPs, use smaller request size. Use 8K request size and create specifications for Random Writes and Reads.

ACCESS SPECIFICATION	REQUEST SIZE	RANDOM %	READ %
RandomWrites_8K	8K	100	0
RandomReads_8K	8K	100	100

Maximum Throughput Test Specifications

To demonstrate maximum Throughput, use larger request size. Use 64K request size and create specifications for Random Writes and Reads.

ACCESS SPECIFICATION	REQUEST SIZE	RANDOM %	READ %
RandomWrites_64K	64K	100	0
RandomReads_64K	64K	100	100

Running the Iometer Test

Perform the steps below to warm up cache

1. Create two access specifications with values shown below,

NAME	REQUEST SIZE	RANDOM %	READ %
RandomWrites_1MB	1MB	100	0
RandomReads_1MB	1MB	100	100

2. Run the lometer test for initializing cache disk with following parameters. Use three worker threads for the target volume and a queue depth of 128. Set the "Run time" duration of the test to 2hrs on the "Test Setup" tab.

SCENARIO	TARGET VOLUME	NAME	DURATION
Initialize Cache Disk	CacheReads	RandomWrites_1MB	2hrs

3. Run the lometer test for warming up cache disk with following parameters. Use three worker threads for the target volume and a queue depth of 128. Set the "Run time" duration of the test to 2hrs on the "Test Setup" tab.

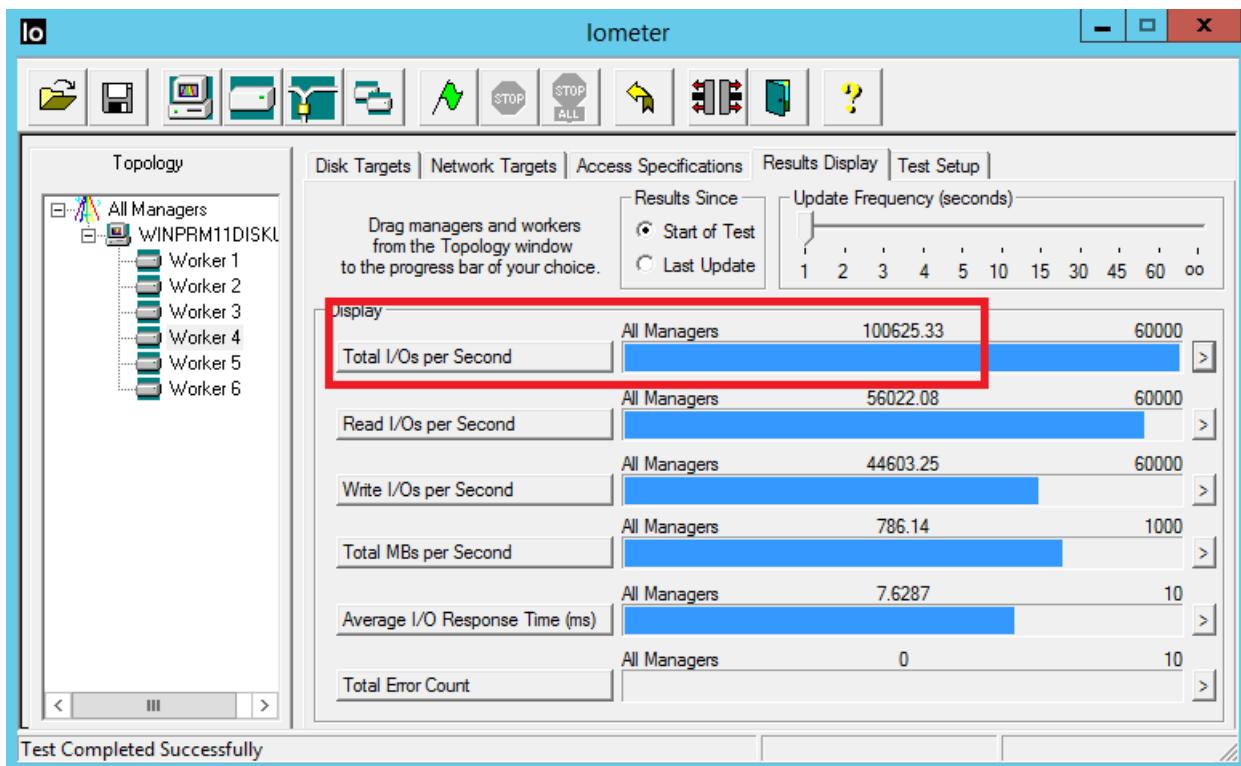
SCENARIO	TARGET VOLUME	NAME	DURATION
Warm up Cache Disk	CacheReads	RandomReads_1MB	2hrs

After cache disk is warmed up, proceed with the test scenarios listed below. To run the lometer test, use at least three worker threads for **each** target volume. For each worker thread, select the target volume, set queue depth and select one of the saved test specifications, as shown in the table below, to run the corresponding test scenario. The table also shows expected results for IOPS and Throughput when running these tests. For all scenarios, a small IO size of 8KB and a high queue depth of 128 is used.

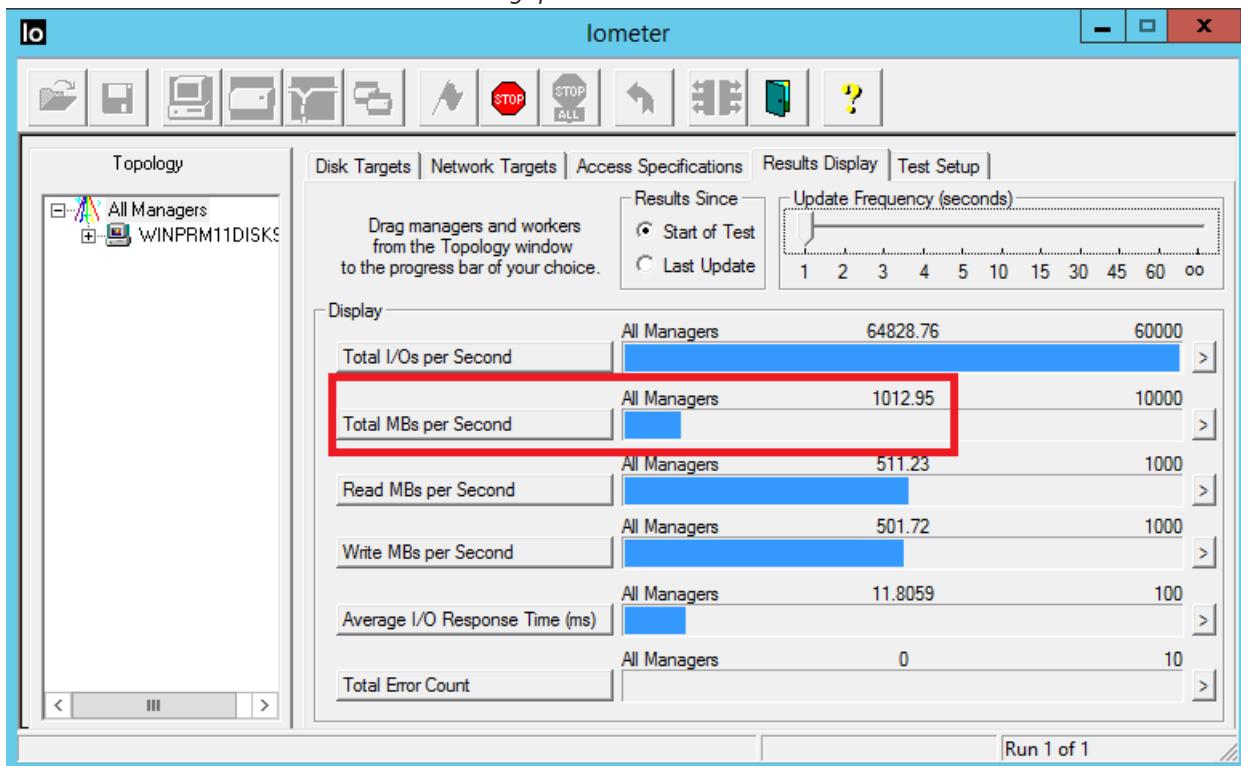
TEST SCENARIO	TARGET VOLUME	NAME	RESULT
Max. Read IOPS	CacheReads	RandomWrites_8K	50,000 IOPS
Max. Write IOPS	NoCacheWrites	RandomReads_8K	64,000 IOPS
Max. Combined IOPS	CacheReads	RandomWrites_8K	100,000 IOPS
NoCacheWrites	RandomReads_8K		
Max. Read MB/sec	CacheReads	RandomWrites_64K	524 MB/sec
Max. Write MB/sec	NoCacheWrites	RandomReads_64K	524 MB/sec
Combined MB/sec	CacheReads	RandomWrites_64K	1000 MB/sec
NoCacheWrites	RandomReads_64K		

Below are screenshots of the lometer test results for combined IOPS and Throughput scenarios.

Combined Reads and Writes Maximum IOPS



Combined Reads and Writes Maximum Throughput



FIO

FIO is a popular tool to benchmark storage on the Linux VMs. It has the flexibility to select different IO sizes, sequential or random reads and writes. It spawns worker threads or processes to perform the specified I/O operations. You can specify the type of I/O operations each worker thread must perform using job files. We created one job file per scenario illustrated in the examples below. You can change the specifications in these job files to benchmark different workloads running on Premium Storage. In the examples, we are using a Standard DS 14 VM running **Ubuntu**. Use the same setup described in the beginning of the [Benchmarking section](#) and warm up the cache before running the benchmarking tests.

Before you begin, [download FIO](#) and install it on your virtual machine.

Run the following command for Ubuntu,

```
apt-get install fio
```

We will use four worker threads for driving Write operations and four worker threads for driving Read operations on the disks. The Write workers will be driving traffic on the "nocache" volume, which has 10 disks with cache set to "None". The Read workers will be driving traffic on the "readcache" volume, which has 1 disk with cache set to "ReadOnly".

Maximum Write IOPS

Create the job file with following specifications to get maximum Write IOPS. Name it "fiowrite.ini".

```
[global]
size=30g
direct=1
iodepth=256
ioengine=libaio
bs=8k

[writer1]
rw=randwrite
directory=/mnt/nocache
[writer2]
rw=randwrite
directory=/mnt/nocache
[writer3]
rw=randwrite
directory=/mnt/nocache
[writer4]
rw=randwrite
directory=/mnt/nocache
```

Note the follow key things that are in line with the design guidelines discussed in previous sections. These specifications are essential to drive maximum IOPS,

- A high queue depth of 256.
- A small block size of 8KB.
- Multiple threads performing random writes.

Run the following command to kick off the FIO test for 30 seconds,

```
sudo fio --runtime 30 fiowrite.ini
```

While the test runs, you will be able to see the number of write IOPS the VM and Premium disks are delivering. As shown in the sample below, the DS14 VM is delivering its maximum write IOPS limit of 50,000 IOPS.

```
demo@DS-VM-Linux-Demo:~$ sudo fio --runtime 30 fiowrite.ini
[sudo] password for demo:
writer1: (g=0): rw=randwrite, bs=8K-8K/8K-8K/8K-8K, ioengine=libaio, iodepth=256
writer2: (g=0): rw=randwrite, bs=8K-8K/8K-8K/8K-8K, ioengine=libaio, iodepth=256
writer3: (g=0): rw=randwrite, bs=8K-8K/8K-8K/8K-8K, ioengine=libaio, iodepth=256
writer4: (g=0): rw=randwrite, bs=8K-8K/8K-8K/8K-8K, ioengine=libaio, iodepth=256
fio-2.1.11
Starting 4 processes
Jobs: 4 (f=4): [w(4)] [63.3% done] [0KB/396.4MB/0KB /s] [0/50.8K/0 iops] [eta 00m:11s]
```

Maximum Read IOPS

Create the job file with following specifications to get maximum Read IOPS. Name it "fioread.ini".

```

[global]
size=30g
direct=1
iodepth=256
ioengine=libaio
bs=8k

[reader1]
rw=randread
directory=/mnt/readcache
[reader2]
rw=randread
directory=/mnt/readcache
[reader3]
rw=randread
directory=/mnt/readcache
[reader4]
rw=randread
directory=/mnt/readcache

```

Note the follow key things that are in line with the design guidelines discussed in previous sections. These specifications are essential to drive maximum IOPS,

- A high queue depth of 256.
- A small block size of 8KB.
- Multiple threads performing random writes.

Run the following command to kick off the FIO test for 30 seconds,

```
sudo fio --runtime 30 fioread.ini
```

While the test runs, you will be able to see the number of read IOPS the VM and Premium disks are delivering. As shown in the sample below, the DS14 VM is delivering more than 64,000 Read IOPS. This is a combination of the disk and the cache performance.

```

demo@DS-VM-Linux-Demo:~$ sudo fio --runtime 30 fioread.ini
[sudo] password for demo:
reader1: (g=0): rw=randread, bs=8K-8K/8K-8K/8K-8K, ioengine=libaio, iodepth=256
reader2: (g=0): rw=randread, bs=8K-8K/8K-8K/8K-8K, ioengine=libaio, iodepth=256
reader3: (g=0): rw=randread, bs=8K-8K/8K-8K/8K-8K, ioengine=libaio, iodepth=256
reader4: (g=0): rw=randread, bs=8K-8K/8K-8K/8K-8K, ioengine=libaio, iodepth=256
fio-2.1.11
Starting 4 processes
Jobs: 4 (f=4): [r(4)] [70.0% done] [514.8MB/0KB/0KB /s] [65.9K/0/0 iops] [eta 00m:09s]
```

Maximum Read and Write IOPS

Create the job file with following specifications to get maximum combined Read and Write IOPS. Name it "fioreadwrite.ini".

```

[global]
size=30g
direct=1
iodepth=128
ioengine=libaio
bs=4k

[reader1]
rw=randread
directory=/mnt/readcache
[reader2]
rw=randread
directory=/mnt/readcache
[reader3]
rw=randread
directory=/mnt/readcache
[reader4]
rw=randread
directory=/mnt/readcache

[writer1]
rw=randwrite
directory=/mnt/nocache
rate_iops=12500
[writer2]
rw=randwrite
directory=/mnt/nocache
rate_iops=12500
[writer3]
rw=randwrite
directory=/mnt/nocache
rate_iops=12500
[writer4]
rw=randwrite
directory=/mnt/nocache
rate_iops=12500

```

Note the follow key things that are in line with the design guidelines discussed in previous sections. These specifications are essential to drive maximum IOPS,

- A high queue depth of 128.
- A small block size of 4KB.
- Multiple threads performing random reads and writes.

Run the following command to kick off the FIO test for 30 seconds,

```
sudo fio --runtime 30 fioreadwrite.ini
```

While the test runs, you will be able to see the number of combined read and write IOPS the VM and Premium disks are delivering. As shown in the sample below, the DS14 VM is delivering more than 100,000 combined Read and Write IOPS. This is a combination of the disk and the cache performance.

```

demo@DS-VM-Linux-Demo:~$ sudo fio --runtime 30 fioreadwrite.ini
reader1: (g=0): rw=randread, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=128
reader2: (g=0): rw=randread, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=128
reader3: (g=0): rw=randread, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=128
reader4: (g=0): rw=randread, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=128
writer1: (g=0): rw=randwrite, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=128
writer2: (g=0): rw=randwrite, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=128
writer3: (g=0): rw=randwrite, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=128
writer4: (g=0): rw=randwrite, bs=4K-4K/4K-4K/4K-4K, ioengine=libaio, iodepth=128
fio-2.1.11
Starting 8 processes
Jobs: 8 (f=8), CR=50000/0 IOPS: [r(4),w(4)] [22.6% done] [251.2MB/183.3MB/0KB /s] [64.3K/46.1K/0 iops] [eta 00m:24s]
```

Maximum Combined Throughput

To get the maximum combined Read and Write Throughput, use a larger block size and large queue depth with

multiple threads performing reads and writes. You can use a block size of 64KB and queue depth of 128.

Next Steps

Learn more about Azure Premium Storage:

- [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#)

For SQL Server users, read articles on Performance Best Practices for SQL Server:

- [Performance Best Practices for SQL Server in Azure Virtual Machines](#)
- [Azure Premium Storage provides highest performance for SQL Server in Azure VM](#)

Back up Azure virtual machine disks with incremental snapshots

1/17/2017 • 7 min to read • [Edit on GitHub](#)

Overview

Azure Storage provides the capability to take snapshots of blobs. Snapshots capture the blob state at that point in time. In this article, we will describe a scenario of how you can maintain backups of virtual machine disks using snapshots. You can use this methodology when you choose not to use Azure Backup and Recovery Service, and wish to create a custom backup strategy for your virtual machine disks.

Azure virtual machine disks are stored as page blobs in Azure Storage. Since we are talking about backup strategy for virtual machine disks in this article, we will be referring to snapshots in the context of page blobs. To learn more about snapshots, refer to [Creating a Snapshot of a Blob](#).

What is a snapshot?

A blob snapshot is a read-only version of a blob that is captured at a point in time. Once a snapshot has been created, it can be read, copied, or deleted, but not modified. Snapshots provide a way to back up a blob as it appears at a moment in time. Until REST version 2015-04-05 you had the ability to copy full snapshots. With the REST version 2015-07-08 and above, you can also copy incremental snapshots.

Full snapshot copy

Snapshots can be copied to another storage account as a blob to keep backups of the base blob. You can also copy a snapshot over its base blob, which is like restoring the blob to an earlier version. When a snapshot is copied from one storage account to another, it will occupy the same space as the base page blob. Therefore, copying whole snapshots from one storage account to another will be slow and will also consume lot of space in the target storage account.

NOTE

If you copy the base blob to another destination, the snapshots of the blob are not copied along with it. Similarly, if you overwrite a base blob with a copy, snapshots associated with the base blob are not affected and stay intact under base blob name.

Back up disks using snapshots

As a backup strategy for your virtual machine disks, you can take periodic snapshots of the disk or page blob, and copy them to another storage account using tools like [Copy Blob](#) operation or [AzCopy](#). You can copy a snapshot to a destination page blob with a different name. The resulting destination page blob is a writeable page blob and not a snapshot. Later in this article we will describe steps to take backups of virtual machine disks using snapshots.

Restore disks using snapshots

When it is time to restore your disk to a previous stable version captured in one of the backup snapshots, you can copy a snapshot over the base page blob. After the snapshot is promoted to the base page blob, the snapshot remains, but its source is overwritten with a copy that can be both read and written. Later in this article we will describe steps to restore a previous version of your disk from its snapshot.

Implementing full snapshot copy

You can implement a full snapshot copy by doing the following,

- First, take a snapshot of the base blob using the [Snapshot Blob](#) operation.
- Then, copy the snapshot to a target storage account using [Copy Blob](#).
- Repeat this process to maintain backup copies of your base blob.

Incremental snapshot copy

The new feature in [GetPageRanges](#) API provides a much better way to back up the snapshots of your page blobs or disks. The API returns the list of changes between the base blob and the snapshots. This reduces the amount of storage space used on the backup account. The API supports page blobs on Premium Storage as well as Standard Storage. Using this API, you can now build faster and efficient backup solutions for Azure VMs. This will be available with the REST version 2015-07-08 and higher.

Incremental Snapshot Copy allows you to copy from one storage account to another the difference between,

- Base blob and its Snapshot OR
- Any two snapshots of the base blob

Provided the following conditions are met,

- The blob was created on Jan-1-2016 or later.
- The blob was not overwritten with [PutPage](#) or [Copy Blob](#) between two snapshots.

Note: This feature is available for Premium and Standard Azure Page Blobs.

When you have a custom backup strategy that uses snapshots, copying the snapshots from one storage account to another can be very slow and consumes a lot of storage space. Instead of copying the entire snapshot to a backup storage account, you can write the difference between consecutive snapshots to a backup page blob. This way, the time to copy and space to store backups is substantially reduced.

Implementing Incremental Snapshot Copy

You can implement incremental snapshot copy by doing the following,

- Take a snapshot of the base blob using [Snapshot Blob](#).
- Copy the snapshot to the target backup storage account using [Copy Blob](#). This will be the backup page blob.
Take a snapshot of this backup page blob and store in backup account.
- Take another snapshot of the base blob using Snapshot Blob.
- Get the difference between first and second snapshots of base blob using [GetPageRanges](#). Use the new parameter **prevsnapshot** to specify the DateTime value of the snapshot you want to get the difference with.
When this parameter is present, the REST response will include only the pages that were changed between target snapshot and previous snapshot including clear pages.
- Use [PutPage](#) to apply these changes to the backup page blob.
- Finally, take a snapshot of the backup page blob and store it in the backup storage account.

In the next section, we will describe in more detail how you can maintain backups of disks using Incremental Snapshot Copy

Scenario

In this section we will describe a scenario that involves a custom backup strategy for virtual machine disks using snapshots.

Consider a DS-series Azure VM with a premium storage P30 disk attached. The P30 disk called *mypremiumdisk* is stored in a premium storage account called *mypremiumaccount*. A standard storage account called *mybackupstdaccount* will be used for storing the backup of *mypremiumdisk*. We would like to keep a snapshot of

mypremiumdisk every 12 hours.

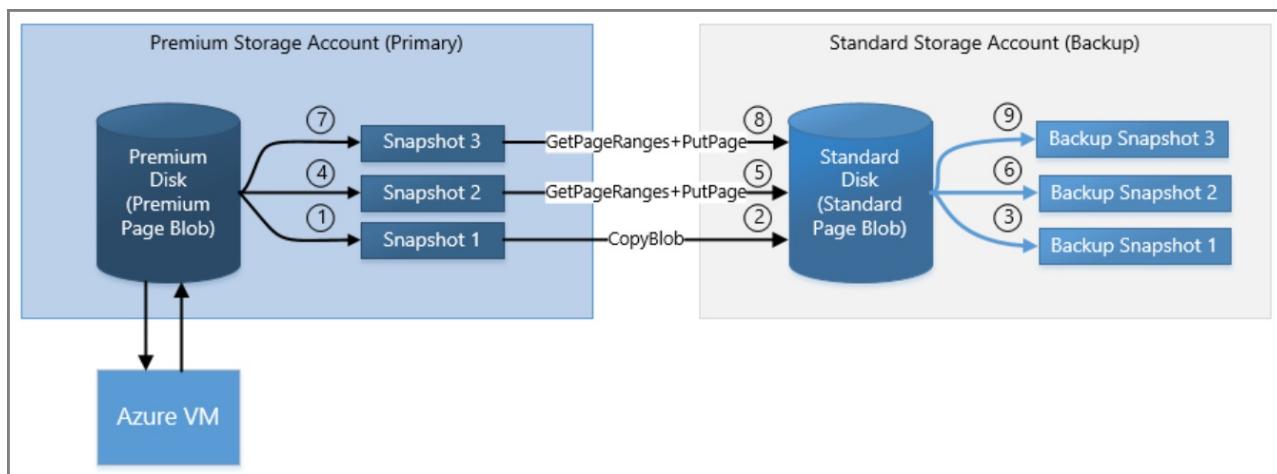
To learn about creating storage account and disks, refer to [About Azure storage accounts](#).

To learn about backing up Azure VMs, refer to [Plan Azure VM backups](#).

Steps to maintain backups of a disk using incremental snapshots

The steps described below will take snapshots of *mypremiumdisk* and maintain the backups in *mybackupstdaccount*. The backup will be a standard page blob called *mybackupstdpageblob*. The backup page blob will always reflect the same state as the last snapshot of *mypremiumdisk*.

1. First, create the backup page blob for your premium storage disk. To do this, take a snapshot of *mypremiumdisk* called *mypremiumdisk_ss1*.
2. Copy this snapshot to *mybackupstdaccount* as a page blob called *mybackupstdpageblob*.
3. Take a snapshot of *mybackupstdpageblob* called *mybackupstdpageblob_ss1*, using [Snapshot Blob](#) and store it in *mybackupstdaccount*.
4. During the backup window, create another snapshot of *mypremiumdisk*, say *mypremiumdisk_ss2*, and store it in *mypremiumaccount*.
5. Get the incremental changes between the two snapshots, *mypremiumdisk_ss2* and *mypremiumdisk_ss1*, using [GetPageRanges](#) on *mypremiumdisk_ss2* with **prevsnapshot** parameter set to the timestamp of *mypremiumdisk_ss1*. Write these incremental changes to the backup page blob *mybackupstdpageblob* in *mybackupstdaccount*. If there are deleted ranges in the incremental changes, they must be cleared from the backup page blob. Use [PutPage](#) to write incremental changes to the backup page blob.
6. Take a snapshot of the backup page blob *mybackupstdpageblob*, called *mybackupstdpageblob_ss2*. Delete the previous snapshot *mypremiumdisk_ss1* from premium storage account.
7. Repeat steps 4-6 every backup window. In this way, you can maintain backups of *mypremiumdisk* in a standard storage account.



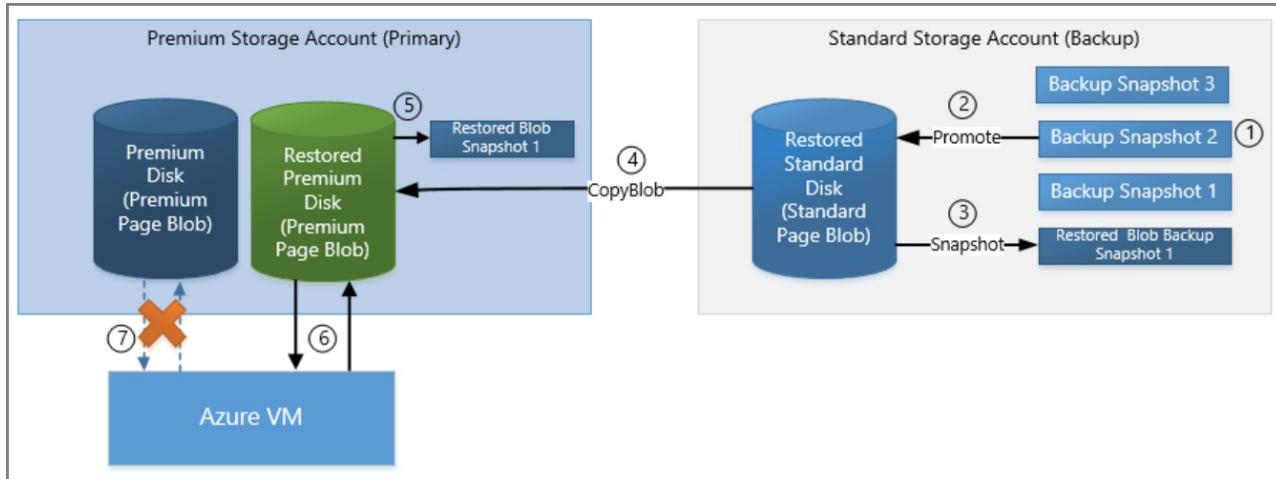
Steps to restore a disk from snapshots

The steps described below will restore premium disk, *mypremiumdisk* to an earlier snapshot from the backup storage account *mybackupstdaccount*.

1. Identify the point in time you wish to restore the premium disk to. Let's say that is snapshot *mybackupstdpageblob_ss2*, which is stored in the backup storage account *mybackupstdaccount*.
2. In *mybackupstdaccount*, promote the snapshot *mybackupstdpageblob_ss2* as the new backup base page blob *mybackupstdpageblobrestored*.
3. Take a snapshot of this restored backup page blob, called *mybackupstdpageblobrestored_ss1*.
4. Copy the restored page blob *mybackupstdpageblobrestored* from *mybackupstdaccount* to *mypremiumaccount*.

as the new premium disk *mypremiumdiskrestored*.

5. Take a snapshot of *mypremiumdiskrestored*, called *mypremiumdiskrestored_ss1* for making future incremental backups.
6. Point the DS series VM to the restored disk *mypremiumdiskrestored* and detach the old *mypremiumdisk* from the VM.
7. Begin the Backup process described in previous section for the restored disk *mypremiumdiskrestored*, using the *mybackupstdpageblobrestored* as the backup page blob.



Next Steps

Learn more about creating snapshots of a blob and planning your VM backup infrastructure using the links below.

- [Creating a Snapshot of a Blob](#)
- [Plan your VM Backup Infrastructure](#)

Azure Storage replication

1/17/2017 • 6 min to read • [Edit on GitHub](#)

The data in your Microsoft Azure storage account is always replicated to ensure durability and high availability. Replication copies your data, either within the same data center, or to a second data center, depending on which replication option you choose. Replication protects your data and preserves your application up-time in the event of transient hardware failures. If your data is replicated to a second data center, that also protects your data against a catastrophic failure in the primary location.

Replication ensures that your storage account meets the [Service-Level Agreement \(SLA\) for Storage](#) even in the face of failures. See the SLA for information about Azure Storage guarantees for durability and availability.

When you create a storage account, you can select one of the following replication options:

- [Locally redundant storage \(LRS\)](#)
- [Zone-redundant storage \(ZRS\)](#)
- [Geo-redundant storage \(GRS\)](#)
- [Read-access geo-redundant storage \(RA-GRS\)](#)

Read-access geo-redundant storage (RA-GRS) is the default option when you create a new storage account.

The following table provides a quick overview of the differences between LRS, ZRS, GRS, and RA-GRS, while subsequent sections address each type of replication in more detail.

REPLICATION STRATEGY	LRS	ZRS	GRS	RA-GRS
Data is replicated across multiple datacenters.	No	Yes	Yes	Yes
Data can be read from the secondary location as well as from the primary location.	No	No	No	Yes
Number of copies of data maintained on separate nodes.	3	3	6	6

See [Azure Storage Pricing](#) for pricing information for the different redundancy options.

NOTE

Premium Storage supports only locally redundant storage (LRS). For information about Premium Storage, see [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#).

Locally redundant storage

Locally redundant storage (LRS) replicates your data three times within a storage scale unit which is hosted in a datacenter in the region in which you created your storage account. A write request returns successfully only

once it has been written to all three replicas. These three replicas each reside in separate fault domains and upgrade domains within one storage scale unit.

A storage scale unit is a collection of racks of storage nodes. A fault domain (FD) is a group of nodes that represent a physical unit of failure and can be considered as nodes belonging to the same physical rack. An upgrade domain (UD) is a group of nodes that are upgraded together during the process of a service upgrade (rollout). The three replicas are spread across UD and FDs within one storage scale unit to ensure that data is available even if hardware failure impacts a single rack or when nodes are upgraded during a rollout.

LRS is the lowest cost option and offers least durability compared to other options. In the event of a datacenter level disaster (fire, flooding etc.) all three replicas might be lost or unrecoverable. To mitigate this risk Geo Redundant Storage (GRS) is recommended for most applications.

Locally redundant storage may still be desirable in certain scenarios:

- Provides highest maximum bandwidth of Azure Storage replication options.
- If your application stores data that can be easily reconstructed, you may opt for LRS.
- Some applications are restricted to replicating data only within a country due to data governance requirements. A paired region could be in another country; please see [Azure regions](#) for information on region pairs.

Zone-redundant storage

Zone-redundant storage (ZRS) replicates your data asynchronously across datacenters within one or two regions in addition to storing three replicas similar to LRS, thus providing higher durability than LRS. Data stored in ZRS is durable even if the primary datacenter is unavailable or unrecoverable. Customers who plan to use ZRS should be aware that:

- ZRS is only available for block blobs in general purpose storage accounts, and is supported only in storage service versions 2014-02-14 and later.
- Since asynchronous replication involves a delay, in the event of a local disaster it is possible that changes that have not yet been replicated to the secondary will be lost if the data cannot be recovered from the primary.
- The replica may not be available until Microsoft initiates failover to the secondary.
- ZRS accounts cannot be converted later to LRS or GRS. Similarly, an existing LRS or GRS account cannot be converted to a ZRS account.
- ZRS accounts do not have metrics or logging capability.

Geo-redundant storage

Geo-redundant storage (GRS) replicates your data to a secondary region that is hundreds of miles away from the primary region. If your storage account has GRS enabled, then your data is durable even in the case of a complete regional outage or a disaster in which the primary region is not recoverable.

For a storage account with GRS enabled, an update is first committed to the primary region, where it is replicated three times. Then the update is replicated asynchronously to the secondary region, where it is also replicated three times.

With GRS both the primary and secondary regions manage replicas across separate fault domains and upgrade domains within a storage scale unit as described with LRS.

Considerations:

- Since asynchronous replication involves a delay, in the event of a regional disaster it is possible that changes that have not yet been replicated to the secondary region will be lost if the data cannot be recovered from the primary region.
- The replica is not available unless Microsoft initiates failover to the secondary region.

- If an application wants to read from the secondary region the user should enable RA-GRS.

When you create a storage account, you select the primary region for the account. The secondary region is determined based on the primary region, and cannot be changed. The following table shows the primary and secondary region pairings.

PRIMARY	SECONDARY
North Central US	South Central US
South Central US	North Central US
East US	West US
West US	East US
US East 2	Central US
Central US	US East 2
North Europe	West Europe
West Europe	North Europe
South East Asia	East Asia
East Asia	South East Asia
East China	North China
North China	East China
Japan East	Japan West
Japan West	Japan East
Brazil South	South Central US
Australia East	Australia Southeast
Australia Southeast	Australia East
India South	India Central
India Central	India South
US Gov Iowa	US Gov Virginia
US Gov Virginia	US Gov Iowa
Canada Central	Canada East
Canada East	Canada Central

PRIMARY	SECONDARY
UK West	UK South
UK South	UK West
Germany Central	Germany Northeast
Germany Northeast	Germany Central
West US 2	West Central US
West Central US	West US 2

For up-to-date information about regions supported by Azure, see [Azure Regions](#).

Read-access geo-redundant storage

Read-access geo-redundant storage (RA-GRS) maximizes availability for your storage account, by providing read-only access to the data in the secondary location, in addition to the replication across two regions provided by GRS.

When you enable read-only access to your data in the secondary region, your data is available on a secondary endpoint, in addition to the primary endpoint for your storage account. The secondary endpoint is similar to the primary endpoint, but appends the suffix `-secondary` to the account name. For example, if your primary endpoint for the Blob service is `myaccount.blob.core.windows.net`, then your secondary endpoint is `myaccount-secondary.blob.core.windows.net`. The access keys for your storage account are the same for both the primary and secondary endpoints.

Considerations:

- Your application has to manage which endpoint it is interacting with when using RA-GRS.
- RA-GRS is intended for high-availability purposes. For scalability guidance, please review the [performance checklist](#).

Next steps

- [Azure Storage Pricing](#)
- [About Azure storage accounts](#)
- [Azure Storage Scalability and Performance Targets](#)
- [Microsoft Azure Storage Redundancy Options and Read Access Geo Redundant Storage](#)
- [SOSP Paper - Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency](#)

Azure Storage Scalability and Performance Targets

1/17/2017 • 8 min to read • [Edit on GitHub](#)

Overview

This topic describes the scalability and performance topics for Microsoft Azure Storage. For a summary of other Azure limits, see [Azure Subscription and Service Limits, Quotas, and Constraints](#).

NOTE

All storage accounts run on the new flat network topology and support the scalability and performance targets outlined below, regardless of when they were created. For more information on the Azure Storage flat network architecture and on scalability, see [Microsoft Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency](#).

IMPORTANT

The scalability and performance targets listed here are high-end targets, but are achievable. In all cases, the request rate and bandwidth achieved by your storage account depends upon the size of objects stored, the access patterns utilized, and the type of workload your application performs. Be sure to test your service to determine whether its performance meets your requirements. If possible, avoid sudden spikes in the rate of traffic and ensure that traffic is well-distributed across partitions.

When your application reaches the limit of what a partition can handle for your workload, Azure Storage will begin to return error code 503 (Server Busy) or error code 500 (Operation Timeout) responses. When this occurs, the application should use an exponential backoff policy for retries. The exponential backoff allows the load on the partition to decrease, and to ease out spikes in traffic to that partition.

If the needs of your application exceed the scalability targets of a single storage account, you can build your application to use multiple storage accounts, and partition your data objects across those storage accounts. See [Azure Storage Pricing](#) for information on volume pricing.

Scalability targets for blobs, queues, tables, and files

RESOURCE	DEFAULT LIMIT
Number of storage accounts per subscription	200 ¹
TB per storage account	500 TB
Max number of blob containers, blobs, file shares, tables, queues, entities, or messages per storage account	Only limit is the 500 TB storage account capacity
Max size of a single blob container, table, or queue	500 TB
Max number of blocks in a block blob or append blob	50,000

RESOURCE	DEFAULT LIMIT
Max size of a block in a block blob	100 MB
Max size of a block blob	50,000 X 100 MB (approx. 4.75 TB)
Max size of a block in an append blob	4 MB
Max size of an append blob	50,000 X 4 MB (approx. 195 GB)
Max size of a page blob	1 TB
Max size of a table entity	1 MB
Max number of properties in a table entity	252
Max size of a message in a queue	64 KB
Max size of a file share	5 TB
Max size of a file in a file share	1 TB
Max number of files in a file share	Only limit is the 5 TB total capacity of the file share
Max 8 KB IOPS per share	1000
Max number of files in a file share	Only limit is the 5 TB total capacity of the file share
Max number of blob containers, blobs, file shares, tables, queues, entities, or messages per storage account	Only limit is the 500 TB storage account capacity
Max number of stored access policies per container, file share, table, or queue	5
Total Request Rate (assuming 1 KB object size) per storage account	Up to 20,000 IOPS, entities per second, or messages per second
Target throughput for single blob	Up to 60 MB per second, or up to 500 requests per second
Target throughput for single queue (1 KB messages)	Up to 2000 messages per second
Target throughput for single table partition (1 KB entities)	Up to 2000 entities per second
Target throughput for single file share	Up to 60 MB per second
Max ingress ² per storage account (US Regions)	10 Gbps if GRS/ZRS ³ enabled, 20 Gbps for LRS
Max egress ² per storage account (US Regions)	20 Gbps if RA-GRS/GRS/ZRS ³ enabled, 30 Gbps for LRS
Max ingress ² per storage account (European and Asian Regions)	5 Gbps if GRS/ZRS ³ enabled, 10 Gbps for LRS

RESOURCE	DEFAULT LIMIT
Max egress ² per storage account (European and Asian Regions)	10 Gbps if RA-GRS/GRS/ZRS ³ enabled, 15 Gbps for LRS

¹This includes both Standard and Premium storage accounts. If you require more than 200 storage accounts, make a request through [Azure Support](#). The Azure Storage team will review your business case and may approve up to 250 storage accounts.

²*Ingress* refers to all data (requests) being sent to a storage account. *Egress* refers to all data (responses) being received from a storage account.

³Azure Storage replication options include:

- **RA-GRS:** Read-access geo-redundant storage. If RA-GRS is enabled, egress targets for the secondary location are identical to those for the primary location.
- **GRS:** Geo-redundant storage.
- **ZRS:** Zone-redundant storage. Available only for block blobs.
- **LRS:** Locally redundant storage.

Scalability targets for virtual machine disks

An Azure virtual machine supports attaching a number of data disks. For optimal performance, you will want to limit the number of highly utilized disks attached to the virtual machine to avoid possible throttling. If all disks are not being highly utilized at the same time, the storage account can support a larger number disks.

- **For standard storage accounts:** A standard storage account has a maximum total request rate of 20,000 IOPS. The total IOPS across all of your virtual machine disks in a standard storage account should not exceed this limit.

You can roughly calculate the number of highly utilized disks supported by a single standard storage account based on the request rate limit. For example, for a Basic Tier VM, the maximum number of highly utilized disks is about 66 (20,000/300 IOPS per disk), and for a Standard Tier VM, it is about 40 (20,000/500 IOPS per disk), as shown in the table below.

- **For premium storage accounts:** A premium storage account has a maximum total throughput rate of 50 Gbps. The total throughput across all of your VM disks should not exceed this limit.

See [Windows VM sizes](#) or [Linux VM sizes](#) for additional details.

Standard storage accounts

Virtual machine disks: per disk limits

VM TIER	BASIC TIER VM	STANDARD TIER VM
Disk size	1023 GB	1023 GB
Max 8 KB IOPS per persistent disk	300	500
Max number of disks performing max IOPS	66	40

Premium storage accounts

Virtual machine disks: per account limits

RESOURCE	DEFAULT LIMIT
Total disk capacity per account	35 TB
Total snapshot capacity per account	10 TB
Max bandwidth per account (ingress + egress ¹)	<=50 Gbps

¹Ingress refers to all data (requests) being sent to a storage account. Egress refers to all data (responses) being received from a storage account.

Virtual machine disks: per disk limits

PREMIUM STORAGE DISK TYPE	P10	P20	P30
Disk size	128 GiB	512 GiB	1024 GiB (1 TB)
Max IOPS per disk	500	2300	5000
Max throughput per disk	100 MB per second	150 MB per second	200 MB per second
Max number of disks per storage account	280	70	35

Virtual machine disks: per VM limits

RESOURCE	DEFAULT LIMIT
Max IOPS Per VM	80,000 IOPS with GS5 VM ¹
Max throughput per VM	2,000 MB/s with GS5 VM ¹

¹Refer to [VM Size](#) for limits on other VM sizes.

Scalability targets for Azure resource manager

The following limits apply when using the Azure Resource Manager and Azure Resource Groups only.

RESOURCE	DEFAULT LIMIT
Storage account management operations (read)	800 per 5 minutes
Storage account management operations (write)	200 per hour
Storage account management operations (list)	100 per 5 minutes

Partitions in Azure Storage

Every object that holds data that is stored in Azure Storage (blobs, messages, entities, and files) belongs to a partition, and is identified by a partition key. The partition determines how Azure Storage load balances blobs, messages, entities, and files across servers to meet the traffic needs of those objects. The partition key is unique and is used to locate a blob, message, or entity.

The table shown above in [Scalability Targets for Standard Storage Accounts](#) lists the performance targets for a single partition for each service.

Partitions affect load balancing and scalability for each of the storage services in the following ways:

- **Blobs:** The partition key for a blob is account name + container name + blob name. This means that each blob can have its own partition if load on the blob demands it. Blobs can be distributed across many servers in order to scale out access to them, but a single blob can only be served by a single server. While blobs can be logically grouped in blob containers, there are no partitioning implications from this grouping.
- **Files:** The partition key for a file is account name + file share name. This means all files in a file share are also in a single partition.
- **Messages:** The partition key for a message is the account name + queue name, so all messages in a queue are grouped into a single partition and are served by a single server. Different queues may be processed by different servers to balance the load for however many queues a storage account may have.
- **Entities:** The partition key for an entity is account name + table name + partition key, where the partition key is the value of the required user-defined **PartitionKey** property for the entity. All entities with the same partition key value are grouped into the same partition and are served by the same partition server. This is an important point to understand in designing your application. Your application should balance the scalability benefits of spreading entities across multiple partitions with the data access advantages of grouping entities in a single partition.

A key advantage to grouping a set of entities in a table into a single partition is that it's possible to perform atomic batch operations across entities in the same partition, since a partition exists on a single server. Therefore, if you wish to perform batch operations on a group of entities, consider grouping them with the same partition key.

On the other hand, entities that are in the same table but have different partition keys can be load balanced across different servers, making it possible to have greater scalability.

Detailed recommendations for designing partitioning strategy for tables can be found [here](#).

See Also

- [Storage Pricing Details](#)
- [Azure Subscription and Service Limits, Quotas, and Constraints](#)
- [Premium Storage: High-Performance Storage for Azure Virtual Machine Workloads](#)
- [Azure Storage Replication](#)
- [Microsoft Azure Storage Performance and Scalability Checklist](#)
- [Microsoft Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency](#)

Microsoft Azure Storage Performance and Scalability Checklist

1/17/2017 • 39 min to read • [Edit on GitHub](#)

Overview

Since the release of the Microsoft Azure Storage services, Microsoft has developed a number of proven practices for using these services in a performant manner, and this article serves to consolidate the most important of them into a checklist-style list. The intention of this article is to help application developers verify they are using proven practices with Azure Storage and to help them identify other proven practices they should consider adopting. This article does not attempt to cover every possible performance and scalability optimization — it excludes those that are small in their impact or not broadly applicable. To the extent that the application's behavior can be predicted during design, it's useful to keep these in mind early on to avoid designs that will run into performance problems.

Every application developer using Azure Storage should take the time to read this article, and check that his or her application follows each of the proven practices listed below.

Checklist

This article organizes the proven practices into the following groups. Proven practices applicable to:

- All Azure Storage services (blobs, tables, queues, and files)
- Blobs
- Tables
- Queues

DONE	AREA	CATEGORY	QUESTION
	All Services	Scalability Targets	Is your application designed to avoid approaching the scalability targets?
	All Services	Scalability Targets	Is your naming convention designed to enable better load-balancing?
	All Services	Networking	Do client side devices have sufficiently high bandwidth and low latency to achieve the performance needed?
	All Services	Networking	Do client side devices have a high enough quality link?
	All Services	Networking	Is the client application located "near" the storage account?
	All Services	Content Distribution	Are you using a CDN for content distribution?

DONE	AREA	CATEGORY	QUESTION
	All Services	Direct Client Access	Are you using SAS and CORS to allow direct access to storage instead of proxy?
	All Services	Caching	Is your application caching data that is repeatedly used and changes rarely?
	All Services	Caching	Is your application batching updates (caching them client side and then uploading in larger sets)?
	All Services	.NET Configuration	Have you configured your client to use a sufficient number of concurrent connections?
	All Services	.NET Configuration	Have you configured .NET to use a sufficient number of threads?
	All Services	.NET Configuration	Are you using .NET 4.5 or later, which has improved garbage collection?
	All Services	Parallelism	Have you ensured that parallelism is bounded appropriately so that you don't overload either your client capabilities or the scalability targets?
	All Services	Tools	Are you using the latest version of Microsoft provided client libraries and tools?
	All Services	Retries	Are you using an exponential backoff retry policy for throttling errors and timeouts?
	All Services	Retries	Is your application avoiding retries for non-retryable errors?
	Blobs	Scalability Targets	Do you have a large number of clients accessing a single object concurrently?
	Blobs	Scalability Targets	Is your application staying within the bandwidth or operations scalability target for a single blob?

DONE	AREA	CATEGORY	QUESTION
	Blobs	Copying Blobs	Are you copying blobs in an efficient manner?
	Blobs	Copying Blobs	Are you using AzCopy for bulk copies of blobs?
	Blobs	Copying Blobs	Are you using Azure Import/Export to transfer very large volumes of data?
	Blobs	Use Metadata	Are you storing frequently used metadata about blobs in their metadata?
	Blobs	Uploading Fast	When trying to upload one blob quickly, are you uploading blocks in parallel?
	Blobs	Uploading Fast	When trying to upload many blobs quickly, are you uploading blobs in parallel?
	Blobs	Correct Blob Type	Are you using page blobs or block blobs when appropriate?
	Tables	Scalability Targets	Are you approaching the scalability targets for entities per second?
	Tables	Configuration	Are you using JSON for your table requests?
	Tables	Configuration	Have you turned Nagle off to improve the performance of small requests?
	Tables	Tables and Partitions	Have you properly partitioned your data?
	Tables	Hot Partitions	Are you avoiding append-only and prepend-only patterns?
	Tables	Hot Partitions	Are your inserts/updates spread across many partitions?
	Tables	Query Scope	Have you designed your schema to allow for point queries to be used in most cases, and table queries to be used sparingly?

DONE	AREA	CATEGORY	QUESTION
	Tables	Query Density	Do your queries typically only scan and return rows that your application will use?
	Tables	Limiting Returned Data	Are you using filtering to avoid returning entities that are not needed?
	Tables	Limiting Returned Data	Are you using projection to avoid returning properties that are not needed?
	Tables	Denormalization	Have you denormalized your data such that you avoid inefficient queries or multiple read requests when trying to get data?
	Tables	Insert/Update/Delete	Are you batching requests that need to be transactional or can be done at the same time to reduce round-trips?
	Tables	Insert/Update/Delete	Are you avoiding retrieving an entity just to determine whether to call insert or update?
	Tables	Insert/Update/Delete	Have you considered storing series of data that will frequently be retrieved together in a single entity as properties instead of multiple entities?
	Tables	Insert/Update/Delete	For entities that will always be retrieved together and can be written in batches (e.g. time series data), have you considered using blobs instead of tables?
	Queues	Scalability Targets	Are you approaching the scalability targets for messages per second?
	Queues	Configuration	Have you turned Nagle off to improve the performance of small requests?
	Queues	Message Size	Are your messages compact to improve the performance of the queue?

DONE	AREA	CATEGORY	QUESTION
	Queues	Bulk Retrieve	Are you retrieving multiple messages in a single "Get" operation?
	Queues	Polling Frequency	Are you polling frequently enough to reduce the perceived latency of your application?
	Queues	Update Message	Are you using UpdateMessage to store progress in processing messages, avoiding having to reprocess the entire message if an error occurs?
	Queues	Architecture	Are you using queues to make your entire application more scalable by keeping long-running workloads out of the critical path and scale them independently?

All Services

This section lists proven practices that are applicable to the use of any of the Azure Storage services (blobs, tables, queues, or files).

Scalability Targets

Each of the Azure Storage services has scalability targets for capacity (GB), transaction rate, and bandwidth. If your application approaches or exceeds any of the scalability targets, it may encounter increased transaction latencies or throttling. When a Storage service throttles your application, the service begins to return "503 Server busy" or "500 Operation timeout" error codes for some storage transactions. This section discusses both the general approach to dealing with scalability targets and bandwidth scalability targets in particular. Later sections that deal with individual storage services discuss scalability targets in the context of that specific service:

- [Blob bandwidth and requests per second](#)
- [Table entities per second](#)
- [Queue messages per second](#)

Bandwidth Scalability Target for All Services

At the time of writing, the bandwidth targets in the US for a geo-redundant storage (GRS) account are 10 gigabits per second (Gbps) for ingress (data sent to the storage account) and 20 Gbps for egress (data sent from the storage account). For a locally redundant storage (LRS) account, the limits are higher – 20 Gbps for ingress and 30 Gbps for egress. International bandwidth limits may be lower and can be found on our [scalability targets page](#).

For more information on the storage redundancy options, see the links in [Useful Resources](#) below.

What to do when approaching a scalability target

If your application is approaching the scalability targets for a single storage account, consider adopting one of the following approaches:

- Reconsider the workload that causes your application to approach or exceed the scalability target. Can you design it differently to use less bandwidth or capacity, or fewer transactions?
- If an application must exceed one of the scalability targets, you should create multiple storage accounts and

partition your application data across those multiple storage accounts. If you use this pattern, then be sure to design your application so that you can add more storage accounts in the future for load balancing. At time of writing, each Azure subscription can have up to 100 storage accounts. Storage accounts also have no cost other than your usage in terms of data stored, transactions made, or data transferred.

- If your application hits the bandwidth targets, consider compressing data in the client to reduce the bandwidth required to send the data to the storage service. Note that while this may save bandwidth and improve network performance, it can also have some negative impacts. You should evaluate the performance impact of this due to the additional processing requirements for compressing and decompressing data in the client. In addition, storing compressed data can make it more difficult to troubleshoot issues since it could be more difficult to view stored data using standard tools.
- If your application hits the scalability targets, then ensure that you are using an exponential backoff for retries (see [Retries](#)). It's better to make sure you never approach the scalability targets (by using one of the above methods), but this will ensure your application won't just keep retrying rapidly, making the throttling worse.

Useful Resources

The following links provide additional detail on scalability targets:

- See [Azure Storage Scalability and Performance Targets](#) for information about scalability targets.
- See [Azure Storage replication](#) and the blog post [Azure Storage Redundancy Options and Read Access Geo Redundant Storage](#) for information about storage redundancy options.
- For current information about pricing for Azure services, see [Azure pricing](#).

Partition Naming Convention

Azure Storage uses a range-based partitioning scheme to scale and load balance the system. The partition key is used to partition data into ranges and these ranges are load-balanced across the system. This means naming conventions such as lexical ordering (e.g. msftpayroll, msftperformance, msftemployees, etc) or using time-stamps (log20160101, log20160102, log20160102, etc) will lend itself to the partitions being potentially co-located on the same partition server, until a load balancing operation splits them out into smaller ranges. For example, all blobs within a container can be served by a single server until the load on these blobs requires further rebalancing of the partition ranges. Similarly, a group of lightly loaded accounts with their names arranged in lexical order may be served by a single server until the load on one or all of these accounts require them to be split across multiple partitions servers. Each load balancing operation may impact the latency of storage calls during the operation. The system's ability to handle a sudden burst of traffic to a partition is limited by the scalability of a single partition server until the load balancing operation kicks-in and rebalances the partition key range.

You can follow some best practices to reduce the frequency of such operations.

- Examine the naming convention you use for accounts, containers, blobs, tables and queues, closely. Consider prefixing account names with a 3-digit hash using a hashing function that best suits your needs.
- If you organize your data using timestamps or numerical identifiers, you have to ensure you are not using an append-only (or prepend-only) traffic patterns. These patterns are not suitable for a range -based partitioning system, and could lead to all the traffic going to a single partition and limiting the system from effectively load balancing. For instance, if you have daily operations that use a blob object with a timestamp such as yyyyymmdd, then all the traffic for that daily operation is directed to a single object which is served by a single partition server. Look at whether the per blob limits and per partition limits meet your needs, and consider breaking this operation into multiple blobs if needed. Similarly, if you store time series data in your tables, all the traffic could be directed to the last part of the key namespace. If you must use timestamps or numerical IDs, prefix the id with a 3-digit hash, or in the case of timestamps prefix the seconds part of the time such as ssyyyyymmdd. If listing and querying operations are routinely performed, choose a hashing function that will limit your number of queries. In other cases, a random prefix may be sufficient.
- For additional information on the partitioning scheme used in Azure Storage, read the SOSP paper [here](#).

Networking

While the API calls matter, often the physical network constraints of the application have a significant impact on performance. The following describe some of limitations users may encounter.

Client Network Capability

Throughput

For bandwidth, the problem is often the capabilities of the client. For example, while a single storage account can handle 10 Gbps or more of ingress (see [bandwidth scalability targets](#)), the network speed in a "Small" Azure Worker Role instance is only capable of approximately 100 Mbps. Larger Azure instances have NICs with greater capacity, so you should consider using a larger instance or more VM's if you need higher network limits from a single machine. If you are accessing a Storage service from an on premises application, then the same rule applies: understand the network capabilities of the client device and the network connectivity to the Azure Storage location and either improve them as needed or design your application to work within their capabilities.

Link Quality

As with any network usage, be aware that network conditions resulting in errors and packet loss will slow effective throughput. Using Wireshark or NetMon may help in diagnosing this issue.

Useful Resources

For more information about virtual machine sizes and allocated bandwidth, see [Windows VM sizes](#) or [Linux VM sizes](#).

Location

In any distributed environment, placing the client near to the server delivers in the best performance. For accessing Azure Storage with the lowest latency, the best location for your client is within the same Azure region. For example, if you have an Azure Web Site that uses Azure Storage, you should locate them both within a single region (for example, US West or Asia Southeast). This reduces the latency and the cost — at the time of writing, bandwidth usage within a single region is free.

If your client applications are not hosted within Azure (such as mobile device apps or on premises enterprise services), then again placing the storage account in a region near to the devices that will access it, will generally reduce latency. If your clients are broadly distributed (for example, some in North America, and some in Europe), then you should consider using multiple storage accounts: one located in a North American region and one in a European region. This will help to reduce latency for users in both regions. This approach is usually easier to implement if the data the application stores is specific to individual users, and does not require replicating data between storage accounts. For broad content distribution, a CDN is recommended – see the next section for more details.

Content Distribution

Sometimes, an application needs to serve the same content to many users (e.g. a product demo video used in the home page of a website), located in either the same or multiple regions. In this scenario, you should use a Content Delivery Network (CDN) such as Azure CDN, and the CDN would use Azure storage as the origin of the data. Unlike an Azure Storage account that exists in a single region and that cannot deliver content with low latency to other regions, Azure CDN uses servers in multiple data centers around the world. Additionally, a CDN can typically support much higher egress limits than a single storage account.

For more information about Azure CDN, see [Azure CDN](#).

Using SAS and CORS

When you need to authorize code such as JavaScript in a user's web browser or a mobile phone app to access data in Azure Storage, one approach is to use an application in web role as a proxy: the user's device authenticates with the web role, which in turn authenticates with the storage service. In this way, you can avoid exposing your storage account keys on insecure devices. However, this places a big overhead on the web role because all the data transferred between the user's device and the storage service must pass through the web role. You can avoid using a web role as a proxy for the storage service by using Shared Access Signatures (SAS), sometimes in conjunction with Cross-Origin Resource Sharing headers (CORS). Using SAS, you can allow your user's device to make requests directly to a storage service by means of a limited access token. For example, if a user wants to

upload a photo to your application, your web role can generate and send to the user's device a SAS token that grants permission to write to a specific blob or container for the next 30 minutes (after which the SAS token expires).

Normally, a browser will not allow JavaScript in a page hosted by a website on one domain to perform specific operations such as a "PUT" to another domain. For example, if you host a web role at "contosomarketing.cloudapp.net," and want to use client side JavaScript to upload a blob to your storage account at "contosoproducts.blob.core.windows.net," the browser's "same origin policy" will forbid this operation. CORS is a browser feature that allows the target domain (in this case the storage account) to communicate to the browser that it trusts requests originating in the source domain (in this case the web role).

Both of these technologies can help you avoid unnecessary load (and bottlenecks) on your web application.

Useful Resources

For more information about SAS, see [Shared Access Signatures, Part 1: Understanding the SAS Model](#).

For more information about CORS, see [Cross-Origin Resource Sharing \(CORS\) Support for the Azure Storage Services](#).

Caching

Getting Data

In general, getting data from a service once is better than getting it twice. Consider the example of an MVC web application running in a web role that has already retrieved a 50MB blob from the storage service to serve as content to a user. The application could then retrieve that same blob every time a user requests it, or it could cache it locally to disk and reuse the cached version for subsequent user requests. Furthermore, whenever a user requests the data, the application could issue GET with a conditional header for modification time, which would avoid getting the entire blob if it hasn't been modified. You can apply this same pattern to working with table entities.

In some cases, you may decide that your application can assume that the blob remains valid for a short period after retrieving it, and that during this period the application does not need to check if the blob was modified.

Configuration, lookup, and other data that are always used by the application are great candidates for caching.

For an example of how to get a blob's properties to discover the last modified date using .NET, see [Set and Retrieve Properties and Metadata](#). For more information about conditional downloads, see [Conditionally Refresh a Local Copy of a Blob](#).

Uploading Data in Batches

In some application scenarios, you can aggregate data locally, and then periodically upload it in a batch instead of uploading each piece of data immediately. For example, a web application might keep a log file of activities: the application could either upload details of every activity as it happens as a table entity (which requires many storage operations), or it could save activity details to a local log file, and then periodically upload all activity details as a delimited file to a blob. If each log entry is 1KB in size, you can upload thousands in a single "Put Blob" transaction (you can upload a blob of up to 64MB in size in a single transaction). Of course, if the local machine crashes prior to the upload, you will potentially lose some log data: the application developer must design for the possibility of client device or upload failures. If the activity data needs to be downloaded for timespans (not just single activity), then blobs are recommended over tables.

.NET Configuration

If using the .NET Framework, this section lists several quick configuration settings that you can use to make significant performance improvements. If using other languages, check to see if similar concepts apply in your chosen language.

Increase default connection limit

In .NET, the following code increases the default connection limit (which is usually 2 in a client environment or 10 in a server environment) to 100. Typically, you should set the value to approximately the number of threads used

by your application.

```
ServicePointManager.DefaultConnectionLimit = 100; //Or More
```

You must set the connection limit before opening any connections.

For other programming languages, see that language's documentation to determine how to set the connection limit.

For additional information, see the blog post [Web Services: Concurrent Connections](#).

Increase ThreadPool Min Threads if using synchronous code with Async Tasks

This code will increase the thread pool min threads:

```
ThreadPool.SetMinThreads(100,100); //Determine the right number for your application
```

For more information, see [ThreadPool.SetMinThreads Method](#).

Take advantage of .NET 4.5 Garbage Collection

Use .NET 4.5 or later for the client application to take advantage of performance improvements in server garbage collection.

For more information, see the article [An Overview of Performance Improvements in .NET 4.5](#).

Unbounded Parallelism

While parallelism can be great for performance, be careful about using unbounded parallelism (no limit on the number of threads and/or parallel requests) to upload or download data, using multiple workers to access multiple partitions (containers, queues, or table partitions) in the same storage account or to access multiple items in the same partition. If the parallelism is unbounded, your application can exceed the client device's capabilities or the storage account's scalability targets resulting in longer latencies and throttling.

Storage Client Libraries and Tools

Always use the latest Microsoft provided client libraries and tools. At the time of writing, there are client libraries available for .NET, Windows Phone, Windows Runtime, Java, and C++, as well as preview libraries for other languages. In addition, Microsoft has released PowerShell cmdlets and Azure CLI commands for working with Azure Storage. Microsoft actively develops these tools with performance in mind, keeps them up to date with the latest service versions, and ensures they handle many of the proven performance practices internally.

Retries

Throttling/ServerBusy

In some cases, the storage service may throttle your application or may simply be unable to serve the request due to some transient condition and return a "503 Server busy" message or "500 Timeout". This can happen if your application is approaching any of the scalability targets, or if the system is rebalancing your partitioned data to allow for higher throughput. The client application should typically retry the operation that causes such an error: attempting the same request later can succeed. However, if the storage service is throttling your application because it is exceeding scalability targets, or even if the service was unable to serve the request for some other reason, aggressive retries usually make the problem worse. For this reason, you should use an exponential back off (the client libraries default to this behavior). For example, your application may retry after 2 seconds, then 4 seconds, then 10 seconds, then 30 seconds, and then give up completely. This behavior results in your application significantly reducing its load on the service rather than exacerbating any problems.

Note that connectivity errors can be retried immediately, because they are not the result of throttling and are expected to be transient.

Non-Retryable Errors

The client libraries are aware of which errors are retry-able and which are not. However, if you are writing your

own code against the storage REST API, remember there are some errors that you should not retry: for example, a 400 (Bad Request) response indicates that the client application sent a request that could not be processed because it was not in an expected form. Resending this request will result the same response every time, so there is no point in retrying it. If you are writing your own code against the storage REST API, be aware of what the error codes mean and the proper way to retry (or not) for each of them.

Useful Resources

For more information about storage error codes, see [Status and Error Codes](#) on the Microsoft Azure web site.

Blobs

In addition to the proven practices for [All Services](#) described previously, the following proven practices apply specifically to the blob service.

Blob-Specific Scalability Targets

Multiple clients accessing a single object concurrently

If you have a large number of clients accessing a single object concurrently you will need to consider per object and storage account scalability targets. The exact number of clients that can access a single object will vary depending on factors such as the number of clients requesting the object simultaneously, the size of the object, network conditions etc.

If the object can be distributed through a CDN such as images or videos served from a website then you should use a CDN. See [here](#).

In other scenarios such as scientific simulations where the data is confidential you have two options. The first is to stagger your workload's access such that the object is accessed over a period of time vs being accessed simultaneously. Alternatively, you can temporarily copy the object to multiple storage accounts thus increasing the total IOPS per object and across storage accounts. In limited testing we found that around 25 VMs could simultaneously download a 100GB blob in parallel (each VM was parallelizing the download using 32 threads). If you had 100 clients needing to access the object, first copy it to a second storage account and then have the first 50 VMs access the first blob and the second 50 VMs access the second blob. Results will vary depending on your applications behavior so you should test this during design.

Bandwidth and operations per Blob

You can read or write to a single blob at up to a maximum of 60 MB/second (this is approximately 480 Mbps which exceeds the capabilities of many client side networks (including the physical NIC on the client device). In addition, a single blob supports up to 500 requests per second. If you have multiple clients that need to read the same blob and you might exceed these limits, you should consider using a CDN for distributing the blob.

For more information about target throughput for blobs, see [Azure Storage Scalability and Performance Targets](#).

Copying and Moving Blobs

Copy Blob

The storage REST API version 2012-02-12 introduced the useful ability to copy blobs across accounts: a client application can instruct the storage service to copy a blob from another source (possibly in a different storage account), and then let the service perform the copy asynchronously. This can significantly reduce the bandwidth needed for the application when you are migrating data from other storage accounts because you do not need to download and upload the data.

One consideration, however, is that, when copying between storage accounts, there is no time guarantee on when the copy will complete. If your application needs to complete a blob copy quickly under your control, it may be better to copy the blob by downloading it to a VM and then uploading it to the destination. For full predictability in that situation, ensure that the copy is performed by a VM running in the same Azure region, or else network conditions may (and probably will) affect your copy performance. In addition, you can monitor the progress of an asynchronous copy programmatically.

Note that copies within the same storage account itself are generally completed quickly.

For more information, see [Copy Blob](#).

Use AzCopy

The Azure Storage team has released a command-line tool "AzCopy" that is meant to help with bulk transferring many blobs to, from, and across storage accounts. This tool is optimized for this scenario, and can achieve high transfer rates. Its use is encouraged for bulk upload, download, and copy scenarios. To learn more about it and download it, see [Transfer data with the AzCopy Command-Line Utility](#).

Azure Import/Export Service

For very large volumes of data (more than 1TB), the Azure Storage offers the Import/Export service, which allows for uploading and downloading from blob storage by shipping hard drives. You can put your data on a hard drive and send it to Microsoft for upload, or send a blank hard drive to Microsoft to download data. For more information, see [Use the Microsoft Azure Import/Export Service to Transfer Data to Blob Storage](#). This can be much more efficient than uploading/downloading this volume of data over the network.

Use metadata

The blob service supports head requests, which can include metadata about the blob. For example, if your application needed the EXIF data out of a photo, it could retrieve the photo and extract it. To save bandwidth and improve performance, your application could store the EXIF data in the blob's metadata when the application uploaded the photo: you can then retrieve the EXIF data in metadata using only a HEAD request, saving significant bandwidth and the processing time needed to extract the EXIF data each time the blob is read. This would be useful in scenarios where you only need the metadata, and not the full content of a blob. Note that only 8 KB of metadata can be stored per blob (the service will not accept a request to store more than that), so if the data does not fit in that size, you may not be able to use this approach.

For an example of how to get a blob's metadata using .NET, see [Set and Retrieve Properties and Metadata](#).

Uploading Fast

To upload blobs fast, the first question to answer is: are you uploading one blob or many? Use the below guidance to determine the correct method to use depending on your scenario.

Uploading one large blob quickly

To upload a single large blob quickly, your client application should upload its blocks or pages in parallel (being mindful of the scalability targets for individual blobs and the storage account as a whole). Note that the official Microsoft-provided RTM Storage Client libraries (.NET, Java) have the ability to do this. For each of the libraries, use the below specified object/property to set the level of concurrency:

- .NET: Set ParallelOperationThreadCount on a BlobRequestOptions object to be used.
- Java/Android: Use BlobRequestOptions.setConcurrentRequestCount()
- Node.js: Use parallelOperationThreadCount on either the request options or on the blob service.
- C++: Use the blob_request_options::set_parallelism_factor method.

Uploading many blobs quickly

To upload many blobs quickly, upload blobs in parallel. This is faster than uploading single blobs at a time with parallel block uploads because it spreads the upload across multiple partitions of the storage service. A single blob only supports a throughput of 60 MB/second (approximately 480 Mbps). At the time of writing, a US-based LRS account supports up to 20 Gbps ingress which is far more than the throughput supported by an individual blob. [AzCopy](#) performs uploads in parallel by default, and is recommended for this scenario.

Choosing the correct type of blob

Azure Storage supports two types of blob: *page* blobs and *block* blobs. For a given usage scenario, your choice of blob type will affect the performance and scalability of your solution. Block blobs are appropriate when you want to upload large amounts of data efficiently: for example, a client application may need to upload photos or video to blob storage. Page blobs are appropriate if the application needs to perform random writes on the data: for

example, Azure VHDs are stored as page blobs.

For more information, see [Understanding Block Blobs, Append Blobs, and Page Blobs](#).

Tables

In addition to the proven practices for [All Services](#) described previously, the following proven practices apply specifically to the table service.

Table-Specific Scalability Targets

In addition to the bandwidth limitations of an entire storage account, tables have the following specific scalability limit. Note that the system will load balance as your traffic increases, but if your traffic has sudden bursts, you may not be able to get this volume of throughput immediately. If your pattern has bursts, you should expect to see throttling and/or timeouts during the burst as the storage service automatically load balances out your table. Ramping up slowly generally has better results as it gives the system time to load balance appropriately.

Entities per Second (Account)

The scalability limit for accessing tables is up to 20,000 entities (1KB each) per second for an account. In general, each entity that is inserted, updated, deleted, or scanned counts toward this target. So a batch insert that contains 100 entities would count as 100 entities. A query that scans 1000 entities and returns 5 would count as 1000 entities.

Entities per Second (Partition)

Within a single partition, the scalability target for accessing tables is 2,000 entities (1KB each) per second, using the same counting as described in the previous section.

Configuration

This section lists several quick configuration settings that you can use to make significant performance improvements in the table service:

Use JSON

Beginning with storage service version 2013-08-15, the table service supports using JSON instead of the XML-based AtomPub format for transferring table data. This can reduce payload sizes by as much as 75% and can significantly improve the performance of your application.

For more information, see the post [Microsoft Azure Tables: Introducing JSON and Payload Format for Table Service Operations](#).

Nagle Off

Nagle's algorithm is widely implemented across TCP/IP networks as a means to improve network performance. However, it is not optimal in all circumstances (such as highly interactive environments). For Azure Storage, Nagle's algorithm has a negative impact on the performance of requests to the table and queue services, and you should disable it if possible.

For more information, see our blog post [Nagle's Algorithm is Not Friendly towards Small Requests](#), which explains why Nagle's algorithm interacts poorly with table and queue requests, and shows how to disable it in your client application.

Schema

How you represent and query your data is the biggest single factor that affects the performance of the table service. While every application is different, this section outlines some general proven practices that relate to:

- Table design
- Efficient queries
- Efficient data updates

Tables and partitions

Tables are divided into partitions. Every entity stored in a partition shares the same partition key and has a unique row key to identify it within that partition. Partitions provide benefits but also introduce scalability limits.

- Benefits: You can update entities in the same partition in a single, atomic, batch transaction that contains up to 100 separate storage operations (limit of 4MB total size). Assuming the same number of entities to be retrieved, you can also query data within a single partition more efficiently than data that spans partitions (though read on for further recommendations on querying table data).
- Scalability limit: Access to entities stored in a single partition cannot be load-balanced because partitions support atomic batch transactions. For this reason, the scalability target for an individual table partition is lower than for the table service as a whole.

Because of these characteristics of tables and partitions, you should adopt the following design principles:

- Data that your client application frequently updated or queried in the same logical unit of work should be located in the same partition. This may be because your application is aggregating writes, or because you want to take advantage of atomic batch operations. Also, data in a single partition can be more efficiently queried in a single query than data across partitions.
- Data that your client application does not insert/update or query in the same logical unit of work (single query or batch update) should be located in separate partitions. One important note is that there is no limit to the number of partition keys in a single table, so having millions of partition keys is not a problem and will not impact performance. For example, if your application is a popular website with user login, using the User Id as the partition key could be a good choice.

Hot Partitions

A hot partition is one that is receiving a disproportionate percentage of the traffic to an account, and cannot be load balanced because it is a single partition. In general, hot partitions are created one of two ways:

Append Only and Prepend Only patterns

The "Append Only" pattern is one where all (or nearly all) of the traffic to a given PK increases and decreases according to the current time. An example is if your application used the current date as a partition key for log data. This results in all of the inserts going to the last partition in your table, and the system cannot load balance because all of the writes are going to the end of your table. If the volume of traffic to that partition exceeds the partition-level scalability target, then it will result in throttling. It's better to ensure that traffic is sent to multiple partitions, to enable load balance the requests across your table.

High-Traffic Data

If your partitioning scheme results in a single partition that just has data that is far more used than other partitions, you may also see throttling as that partition approaches the scalability target for a single partition. It's better to make sure that your partition scheme results in no single partition approaching the scalability targets.

Querying

This section describes proven practices for querying the table service.

Query Scope

There are several ways to specify the range of entities to query. The following is a discussion of the uses of each.

In general, avoid scans (queries larger than a single entity), but if you must scan, try to organize your data so that your scans retrieve the data you need without scanning or returning significant amounts of entities you don't need.

Point Queries

A point query retrieves exactly one entity. It does this by specifying both the partition key and row key of the entity to retrieve. These queries are very efficient, and you should use them wherever possible.

Partition Queries

A partition query is a query that retrieves a set of data that shares a common partition key. Typically, the query specifies a range of row key values or a range of values for some entity property in addition to a partition key. These are less efficient than point queries, and should be used sparingly.

Table Queries

A table query is a query that retrieves a set of entities that does not share a common partition key. These queries are not efficient and you should avoid them if possible.

Query Density

Another key factor in query efficiency is the number of entities returned as compared to the number of entities scanned to find the returned set. If your application performs a table query with a filter for a property value that only 1% of the data shares, the query will scan 100 entities for every one entity it returns. The table scalability targets discussed previously all relate to the number of entities scanned, and not the number of entities returned: a low query density can easily cause the table service to throttle your application because it must scan so many entities to retrieve the entity you are looking for. See the section below on [denormalization](#) for more information on how to avoid this.

Limiting the Amount of Data Returned

Filtering

Where you know that a query will return entities that you don't need in the client application, consider using a filter to reduce the size of the returned set. While the entities not returned to the client still count toward the scalability limits, your application performance will improve because of the reduced network payload size and the reduced number of entities that your client application must process. See above note on [Query Density](#), however – the scalability targets relate to the number of entities scanned, so a query that filters out many entities may still result in throttling, even if few entities are returned.

Projection

If your client application needs only a limited set of properties from the entities in your table, you can use projection to limit the size of the returned data set. As with filtering, this helps to reduce network load and client processing.

Denormalization

Unlike working with relational databases, the proven practices for efficiently querying table data lead to denormalizing your data. That is, duplicating the same data in multiple entities (one for each key you may use to find the data) to minimize the number of entities that a query must scan to find the data the client needs, rather than having to scan large numbers of entities to find the data your application needs. For example, in an e-commerce website, you may want to find an order both by the customer ID (give me this customer's orders) and by the date (give me orders on a date). In Table Storage, it is best to store the entity (or a reference to it) twice – once with Table Name, PK, and RK to facilitate finding by customer ID, once to facilitate finding it by the date.

Insert/Update/Delete

This section describes proven practices for modifying entities stored in the table service.

Batching

Batch transactions are known as Entity Group Transactions (ETG) in Azure Storage; all the operations within an ETG must be on a single partition in a single table. Where possible, use ETGs to perform inserts, updates, and deletes in batches. This reduces the number of round trips from your client application to the server, reduces the number of billable transactions (an ETG counts as a single transaction for billing purposes and can contain up to 100 storage operations), and enables atomic updates (all operations succeed or all fail within an ETG).

Environments with high latencies such as mobile devices will benefit greatly from using ETGs.

Upsert

Use table **Upsert** operations wherever possible. There are two types of **Upsert**, both of which can be more efficient than a traditional **Insert** and **Update** operations:

- **InsertOrMerge**: Use this when you want to upload a subset of the entity's properties, but aren't sure whether the entity already exists. If the entity exists, this call updates the properties included in the **Upsert** operation, and leaves all existing properties as they are, if the entity does not exist, it inserts the new entity. This is similar to using projection in a query, in that you only need to upload the properties that are changing.
- **InsertOrReplace**: Use this when you want to upload an entirely new entity, but you aren't sure whether it already exists. You should only use this when you know that the newly uploaded entity is entirely correct because it completely overwrites the old entity. For example, you want to update the entity that stores a user's current location regardless of whether or not the application has previously stored location data for the user;

the new location entity is complete, and you do not need any information from any previous entity.

Storing Data Series in a Single Entity

Sometimes, an application stores a series of data that it frequently needs to retrieve all at once: for example, an application might track CPU usage over time in order to plot a rolling chart of the data from the last 24 hours. One approach is to have one table entity per hour, with each entity representing a specific hour and storing the CPU usage for that hour. To plot this data, the application needs to retrieve the entities holding the data from the 24 most recent hours.

Alternatively, your application could store the CPU usage for each hour as a separate property of a single entity: to update each hour, your application can use a single **InsertOrMerge Upsert** call to update the value for the most recent hour. To plot the data, the application only needs to retrieve a single entity instead of 24, making for a very efficient query (see above discussion on [query scope](#)).

Storing structured data in blobs

Sometimes structured data feels like it should go in tables, but ranges of entities are always retrieved together and can be batch inserted. A good example of this is a log file. In this case, you can batch several minutes of logs, insert them, and then you are always retrieving several minutes of logs at a time as well. In this case, for performance, it's better to use blobs instead of tables, since you can significantly reduce the number of objects written/returned, as well as usually the number of requests that need made.

Queues

In addition to the proven practices for [All Services](#) described previously, the following proven practices apply specifically to the queue service.

Scalability Limits

A single queue can process approximately 2,000 messages (1KB each) per second (each AddMessage, GetMessage, and DeleteMessage count as a message here). If this is insufficient for your application, you should use multiple queues and spread the messages across them.

View current scalability targets at [Azure Storage Scalability and Performance Targets](#).

Nagle Off

See the section on table configuration that discusses the Nagle algorithm — the Nagle algorithm is generally bad for the performance of queue requests, and you should disable it.

Message Size

Queue performance and scalability decreases as message size increases. You should place only the information the receiver needs in a message.

Batch Retrieval

You can retrieve up to 32 messages from a queue in a single operation. This can reduce the number of roundtrips from the client application, which is especially useful for environments, such as mobile devices, with high latency.

Queue Polling Interval

Most applications poll for messages from a queue, which can be one of the largest sources of transactions for that application. Select your polling interval wisely: polling too frequently could cause your application to approach the scalability targets for the queue. However, at 200,000 transactions for \$0.01 (at the time of writing), a single processor polling once every second for a month would cost less than 15 cents so cost is not typically a factor that affects your choice of polling interval.

For up-to-date cost information, see [Azure Storage Pricing](#).

UpdateMessage

You can use **UpdateMessage** to increase the invisibility timeout or to update state information of a message.

While this is powerful, remember that each **UpdateMessage** operation counts towards the scalability target. However, this can be a much more efficient approach than having a workflow that passes a job from one queue to the next, as each step of the job is completed. Using the **UpdateMessage** operation allows your application to save the job state to the message and then continue working, instead of re-queuing the message for the next step of the job every time a step completes.

For more information, see the article [How to: Change the contents of a queued message](#).

Application architecture

You should use queues to make your application architecture scalable. The following lists some ways you can use queues to make your application more scalable:

- You can use queues to create backlogs of work for processing and smooth out workloads in your application. For example, you could queue up requests from users to perform processor intensive work such as resizing uploaded images.
- You can use queues to decouple parts of your application so that you can scale them independently. For example, a web front-end could place survey results from users into a queue for later analysis and storage. You could add more worker role instances to process the queue data as required.

Conclusion

This article discussed some of the most common, proven practices for optimizing performance when using Azure Storage. We encourage every application developer to assess their application against each of the above practices and consider acting on the recommendations to get great performance for their applications that use Azure Storage.

Managing Concurrency in Microsoft Azure Storage

1/17/2017 • 16 min to read • [Edit on GitHub](#)

Overview

Modern Internet based applications usually have multiple users viewing and updating data simultaneously. This requires application developers to think carefully about how to provide a predictable experience to their end users, particularly for scenarios where multiple users can update the same data. There are three main data concurrency strategies developers will typically consider:

1. Optimistic concurrency – An application performing an update will as part of its update verify if the data has changed since the application last read that data. For example, if two users viewing a wiki page make an update to the same page then the wiki platform must ensure that the second update does not overwrite the first update – and that both users understand whether their update was successful or not. This strategy is most often used in web applications.
2. Pessimistic concurrency – An application looking to perform an update will take a lock on an object preventing other users from updating the data until the lock is released. For example, in a master/slave data replication scenario where only the master will perform updates the master will typically hold an exclusive lock for an extended period of time on the data to ensure no one else can update it.
3. Last writer wins – An approach that allows any update operations to proceed without verifying if any other application has updated the data since the application first read the data. This strategy (or lack of a formal strategy) is usually used where data is partitioned in such a way that there is no likelihood that multiple users will access the same data. It can also be useful where short-lived data streams are being processed.

This article provides an overview of how the Azure Storage platform simplifies development by providing first class support for all three of these concurrency strategies.

Azure Storage – Simplifies Cloud Development

The Azure storage service supports all three strategies, although it is distinctive in its ability to provide full support for optimistic and pessimistic concurrency because it was designed to embrace a strong consistency model which guarantees that when the Storage service commits a data insert or update operation all further accesses to that data will see the latest update. Storage platforms that use an eventual consistency model have a lag between when a write is performed by one user and when the updated data can be seen by other users thus complicating development of client applications in order to prevent inconsistencies from affecting end users.

In addition to selecting an appropriate concurrency strategy developers should also be aware of how a storage platform isolates changes – particularly changes to the same object across transactions. The Azure storage service uses snapshot isolation to allow read operations to happen concurrently with write operations within a single partition. Unlike other isolation levels, snapshot isolation guarantees that all reads see a consistent snapshot of the data even while updates are occurring – essentially by returning the last committed values while an update transaction is being processed.

Managing Concurrency in Blob storage

You can opt to use either optimistic or pessimistic concurrency models to manage access to blobs and containers in the blob service. If you do not explicitly specify a strategy last writes wins is the default.

Optimistic concurrency for blobs and containers

The Storage service assigns an identifier to every object stored. This identifier is updated every time an update

operation is performed on an object. The identifier is returned to the client as part of an HTTP GET response using the ETag (entity tag) header that is defined within the HTTP protocol. A user performing an update on such an object can send in the original ETag along with a conditional header to ensure that an update will only occur if a certain condition has been met – in this case the condition is an “If-Match” header which requires the Storage Service to ensure the value of the ETag specified in the update request is the same as that stored in the Storage Service.

The outline of this process is as follows:

1. Retrieve a blob from the storage service, the response includes an HTTP ETag Header value that identifies the current version of the object in the storage service.
2. When you update the blob, include the ETag value you received in step 1 in the **If-Match** conditional header of the request you send to the service.
3. The service compares the ETag value in the request with the current ETag value of the blob.
4. If the current ETag value of the blob is a different version than the ETag in the **If-Match** conditional header in the request, the service returns a 412 error to the client. This indicates to the client that another process has updated the blob since the client retrieved it.
5. If the current ETag value of the blob is the same version as the ETag in the **If-Match** conditional header in the request, the service performs the requested operation and updates the current ETag value of the blob to show that it has created a new version.

The following C# snippet (using the Client Storage Library 4.2.0) shows a simple example of how to construct an **If-Match AccessCondition** based on the ETag value that is accessed from the properties of a blob that was previously either retrieved or inserted. It then uses the **AccessCondition** object when it updating the blob: the **AccessCondition** object adds the **If-Match** header to the request. If another process has updated the blob, the blob service returns an HTTP 412 (Precondition Failed) status message. You can download the full sample here: [Managing Concurrency using Azure Storage](#).

```
// Retrieve the ETag from the newly created blob
// Etag is already populated as UploadText should cause a PUT Blob call
// to storage blob service which returns the etag in response.
string originalETag = blockBlob.Properties.ETag;

// This code simulates an update by a third party.
string helloText = "Blob updated by a third party.';

// No etag, provided so original blob is overwritten (thus generating a new etag)
blockBlob.UploadText(helloText);
Console.WriteLine("Blob updated. Updated ETag = {0}",
blockBlob.Properties.ETag);

// Now try to update the blob using the original ETag provided when the blob was created
try
{
    Console.WriteLine("Trying to update blob using original etag to generate if-match access condition");
    blockBlob.UploadText(helloText,accessCondition:
    AccessCondition.GenerateIfMatchCondition(originalETag));
}
catch (StorageException ex)
{
    if (ex.RequestInformation.HttpStatusCode == (int)HttpStatusCode.PreconditionFailed)
    {
        Console.WriteLine("Precondition failure as expected. Blob's original etag no longer matches");
        // TODO: client can decide on how it wants to handle the 3rd party updated content.
    }
    else
        throw;
}
```

The Storage Service also includes support for additional conditional headers such as **If-Modified-Since**, **If-**

Unmodified-Since and **If-None-Match** as well as combinations thereof. For more information see [Specifying Conditional Headers for Blob Service Operations](#) on MSDN.

The following table summarizes the container operations that accept conditional headers such as **If-Match** in the request and that return an ETag value in the response.

OPERATION	RETURNS CONTAINER ETAG VALUE	ACCEPTS CONDITIONAL HEADERS
Create Container	Yes	No
Get Container Properties	Yes	No
Get Container Metadata	Yes	No
Set Container Metadata	Yes	Yes
Get Container ACL	Yes	No
Set Container ACL	Yes	Yes (*)
Delete Container	No	Yes
Lease Container	Yes	Yes
List Blobs	No	No

(*) The permissions defined by SetContainerACL are cached and updates to these permissions take 30 seconds to propagate during which period updates are not guaranteed to be consistent.

The following table summarizes the blob operations that accept conditional headers such as **If-Match** in the request and that return an ETag value in the response.

OPERATION	RETURNS ETAG VALUE	ACCEPTS CONDITIONAL HEADERS
Put Blob	Yes	Yes
Get Blob	Yes	Yes
Get Blob Properties	Yes	Yes
Set Blob Properties	Yes	Yes
Get Blob Metadata	Yes	Yes
Set Blob Metadata	Yes	Yes
Lease Blob (*)	Yes	Yes
Snapshot Blob	Yes	Yes
Copy Blob	Yes	Yes (for source and destination blob)
Abort Copy Blob	No	No

OPERATION	RETURNS ETAG VALUE	ACCEPTS CONDITIONAL HEADERS
Delete Blob	No	Yes
Put Block	No	No
Put Block List	Yes	Yes
Get Block List	Yes	No
Put Page	Yes	Yes
Get Page Ranges	Yes	Yes

(*) Lease Blob does not change the ETag on a blob.

Pessimistic concurrency for blobs

To lock a blob for exclusive use, you can acquire a [lease](#) on it. When you acquire a lease, you specify for how long you need the lease: this can be for between 15 to 60 seconds or infinite which amounts to an exclusive lock. You can renew a finite lease to extend it, and you can release any lease when you are finished with it. The blob service automatically releases finite leases when they expire.

Leases enable different synchronization strategies to be supported, including exclusive write / shared read, exclusive write / exclusive read and shared write / exclusive read. Where a lease exists the storage service enforces exclusive writes (put, set and delete operations) however ensuring exclusivity for read operations requires the developer to ensure that all client applications use a lease ID and that only one client at a time has a valid lease ID. Read operations that do not include a lease ID result in shared reads.

The following C# snippet shows an example of acquiring an exclusive lease for 30 seconds on a blob, updating the content of the blob, and then releasing the lease. If there is already a valid lease on the blob when you try to acquire a new lease, the blob service returns an "HTTP (409) Conflict" status result. The snippet below uses an **AccessCondition** object to encapsulate the lease information when it makes a request to update the blob in the storage service. You can download the full sample here: [Managing Concurrency using Azure Storage](#).

```
// Acquire lease for 15 seconds
string lease = blockBlob.AcquireLease(TimeSpan.FromSeconds(15), null);
Console.WriteLine("Blob lease acquired. Lease = {0}", lease);

// Update blob using lease. This operation will succeed
const string helloText = "Blob updated";
var accessCondition = AccessCondition.GenerateLeaseCondition(lease);
blockBlob.UploadText(helloText, accessCondition: accessCondition);
Console.WriteLine("Blob updated using an exclusive lease");

//Simulate third party update to blob without lease
try
{
    // Below operation will fail as no valid lease provided
    Console.WriteLine("Trying to update blob without valid lease");
    blockBlob.UploadText("Update without lease, will fail");
}
catch (StorageException ex)
{
    if (ex.RequestInformation.HttpStatusCode == (int)HttpStatusCode.PreconditionFailed)
        Console.WriteLine("Precondition failure as expected. Blob's lease does not match");
    else
        throw;
}
```

If you attempt a write operation on a leased blob without passing the lease ID, the request fails with a 412 error. Note that if the lease expires before calling the **UploadText** method but you still pass the lease ID, the request also fails with a **412** error. For more information about managing lease expiry times and lease ids, see the [Lease Blob](#) REST documentation.

The following blob operations can use leases to manage pessimistic concurrency:

- Put Blob
- Get Blob
- Get Blob Properties
- Set Blob Properties
- Get Blob Metadata
- Set Blob Metadata
- Delete Blob
- Put Block
- Put Block List
- Get Block List
- Put Page
- Get Page Ranges
- Snapshot Blob - lease ID optional if a lease exists
- Copy Blob - lease ID required if a lease exists on the destination blob
- Abort Copy Blob - lease ID required if an infinite lease exists on the destination blob
- Lease Blob

Pessimistic concurrency for containers

Leases on containers enable the same synchronization strategies to be supported as on blobs (exclusive write / shared read, exclusive write / exclusive read and shared write / exclusive read) however unlike blobs the storage service only enforces exclusivity on delete operations. To delete a container with an active lease, a client must include the active lease ID with the delete request. All other container operations succeed on a leased container without including the lease ID in which case they are shared operations. If exclusivity of update (put or set) or read operations is required then developers should ensure all clients use a lease ID and that only one client at a time has a valid lease ID.

The following container operations can use leases to manage pessimistic concurrency:

- Delete Container
- Get Container Properties
- Get Container Metadata
- Set Container Metadata
- Get Container ACL
- Set Container ACL
- Lease Container

For more information see:

- [Specifying Conditional Headers for Blob Service Operations](#)
- [Lease Container](#)
- [Lease Blob](#)

Managing Concurrency in the Table Service

The table service uses optimistic concurrency checks as the default behavior when you are working with entities,

unlike the blob service where you must explicitly choose to perform optimistic concurrency checks. The other difference between the table and blob services is that you can only manage the concurrency behavior of entities whereas with the blob service you can manage the concurrency of both containers and blobs.

To use optimistic concurrency and to check if another process modified an entity since you retrieved it from the table storage service, you can use the ETag value you receive when the table service returns an entity. The outline of this process is as follows:

1. Retrieve an entity from the table storage service, the response includes an ETag value that identifies the current identifier associated with that entity in the storage service.
2. When you update the entity, include the ETag value you received in step 1 in the mandatory **If-Match** header of the request you send to the service.
3. The service compares the ETag value in the request with the current ETag value of the entity.
4. If the current ETag value of the entity is different than the ETag in the mandatory **If-Match** header in the request, the service returns a 412 error to the client. This indicates to the client that another process has updated the entity since the client retrieved it.
5. If the current ETag value of the entity is the same as the ETag in the mandatory **If-Match** header in the request or the **If-Match** header contains the wildcard character (*), the service performs the requested operation and updates the current ETag value of the entity to show that it has been updated.

Note that unlike the blob service, the table service requires the client to include an **If-Match** header in update requests. However, it is possible to force an unconditional update (last writer wins strategy) and bypass concurrency checks if the client sets the **If-Match** header to the wildcard character (*) in the request.

The following C# snippet shows a customer entity that was previously either created or retrieved having their email address updated. The initial insert or retrieve operation stores the ETag value in the customer object, and because the sample uses the same object instance when it executes the replace operation, it automatically sends the ETag value back to the table service, enabling the service to check for concurrency violations. If another process has updated the entity in table storage, the service returns an HTTP 412 (Precondition Failed) status message. You can download the full sample here: [Managing Concurrency using Azure Storage](#).

```
try
{
    customer.Email = "updatedEmail@contoso.org";
    TableOperation replaceCustomer = TableOperation.Replace(customer);
    customerTable.Execute(replaceCustomer);
    Console.WriteLine("Replace operation succeeded.");
}
catch (StorageException ex)
{
    if (ex.RequestInformation.HttpStatusCode == 412)
        Console.WriteLine("Optimistic concurrency violation - entity has changed since it was retrieved.");
    else
        throw;
}
```

To explicitly disable the concurrency check, you should set the **ETag** property of the **employee** object to "*" before you execute the replace operation.

```
customer.ETag = "*";
```

The following table summarizes how the table entity operations use ETag values:

OPERATION	RETURNS ETAG VALUE	REQUIRES IF-MATCH REQUEST HEADER
Query Entities	Yes	No

OPERATION	RETURNS ETAG VALUE	REQUIRES IF-MATCH REQUEST HEADER
Insert Entity	Yes	No
Update Entity	Yes	Yes
Merge Entity	Yes	Yes
Delete Entity	No	Yes
Insert or Replace Entity	Yes	No
Insert or Merge Entity	Yes	No

Note that the **Insert or Replace Entity** and **Insert or Merge Entity** operations do *not* perform any concurrency checks because they do not send an ETag value to the table service.

In general developers using tables should rely on optimistic concurrency when developing scalable applications. If pessimistic locking is needed, one approach developers can take when accessing Tables is to assign a designated blob for each table and try to take a lease on the blob before operating on the table. This approach does require the application to ensure all data access paths obtain the lease prior to operating on the table. You should also note that the minimum lease time is 15 seconds which requires careful consideration for scalability.

For more information see:

- [Operations on Entities](#)

Managing Concurrency in the Queue Service

One scenario in which concurrency is a concern in the queueing service is where multiple clients are retrieving messages from a queue. When a message is retrieved from the queue, the response includes the message and a pop receipt value, which is required to delete the message. The message is not automatically deleted from the queue, but after it has been retrieved, it is not visible to other clients for the time interval specified by the visibilitytimeout parameter. The client that retrieves the message is expected to delete the message after it has been processed, and before the time specified by the TimeNextVisible element of the response, which is calculated based on the value of the visibilitytimeout parameter. The value of visibilitytimeout is added to the time at which the message is retrieved to determine the value of TimeNextVisible.

The queue service does not have support for either optimistic or pessimistic concurrency and for this reason clients processing messages retrieved from a queue should ensure messages are processed in an idempotent manner. A last writer wins strategy is used for update operations such as SetQueueServiceProperties, SetQueueMetaData, SetQueueACL and UpdateMessage.

For more information see:

- [Queue Service REST API](#)
- [Get Messages](#)

Managing Concurrency in the File Service

The file service can be accessed using two different protocol endpoints – SMB and REST. The REST service does not have support for either optimistic locking or pessimistic locking and all updates will follow a last writer wins strategy. SMB clients that mount file shares can leverage file system locking mechanisms to manage access to shared files – including the ability to perform pessimistic locking. When an SMB client opens a file, it specifies both the file access and share mode. Setting a File Access option of "Write" or "Read/Write" along with a File Share

mode of "None" will result in the file being locked by an SMB client until the file is closed. If REST operation is attempted on a file where an SMB client has the file locked the REST service will return status code 409 (Conflict) with error code SharingViolation.

When an SMB client opens a file for delete, it marks the file as pending delete until all other SMB client open handles on that file are closed. While a file is marked as pending delete, any REST operation on that file will return status code 409 (Conflict) with error code SMBDeletePending. Status code 404 (Not Found) is not returned since it is possible for the SMB client to remove the pending deletion flag prior to closing the file. In other words, status code 404 (Not Found) is only expected when the file has been removed. Note that while a file is in a SMB pending delete state, it will not be included in the List Files results. Also note that the REST Delete File and REST Delete Directory operations are committed atomically and do not result in pending delete state.

For more information see:

- [Managing File Locks](#)

Summary and Next Steps

The Microsoft Azure Storage service has been designed to meet the needs of the most complex online applications without forcing developers to compromise or rethink key design assumptions such as concurrency and data consistency that they have come to take for granted.

For the complete sample application referenced in this blog:

- [Managing Concurrency using Azure Storage - Sample Application](#)

For more information on Azure Storage see:

- [Microsoft Azure Storage Home Page](#)
- [Introduction to Azure Storage](#)
- Storage Getting Started for [Blob](#), [Table](#), [Queues](#), and [Files](#)
- Storage Architecture – [Azure Storage : A Highly Available Cloud Storage Service with Strong Consistency](#)

Azure Storage samples

1/17/2017 • 1 min to read • [Edit on GitHub](#)

Overview

Use the links below to view and download working Azure Storage samples.

Azure Code Samples library

The [Azure Code Samples](#) library includes samples for Azure Storage that you can download and run locally. The Code Sample Library provides sample code in .zip format. Alternatively, you can browse and clone the GitHub repository for each sample.

Getting started samples

- [Get started with Azure Storage in five minutes](#)
- [Visual Studio Quick Starts for Azure Storage](#)

.NET samples

To explore the .NET samples, download the [.NET Storage Client Library](#) from NuGet. The .NET storage client library is also available in the [Azure SDK for .NET](#).

- [Azure Storage samples using .NET](#)

Java samples

To explore the Java samples, download the [Java Storage Client Library](#).

- [Azure Storage samples using Java](#)

Node.js samples

To explore the Node.js samples, download the [Node.js Storage Client Library](#).

- [Blob uploader](#)
- [Upload and download blob](#)
- [Continuation token](#)
- [Retry policy](#)
- [Shared access signature](#)
- [Snapshot](#)
- [Table query](#)

C++ samples

To explore the C++ samples, download the [C++ Storage Client Library](#) from NuGet.

- [Get started with blobs](#)
- [Get started with tables](#)
- [Get started with queues](#)

Next steps

.NET resources

- [Source code for the .NET storage client library](#)
- [.NET Client Library Reference](#)

Java resources

- [Source code for the Java storage client library](#)
- [Java Client Library Reference](#)

Node.js resources

- [Source code for the Node.js storage client library](#)
- [Node.js Client Library Reference](#)

C++ resources

- [Source code for the C++ storage client library](#)
- [C++ Client Library Reference](#)

Configure Azure Storage Connection Strings

1/17/2017 • 7 min to read • [Edit on GitHub](#)

Overview

A connection string includes the authentication information needed to access data in an Azure storage account from your application at runtime. You can configure a connection string to:

- Connect to the Azure storage emulator.
- Access a storage account in Azure.
- Access specified resources in Azure via a shared access signature (SAS).

IMPORTANT

Your storage account key is similar to the root password for your storage account. Always be careful to protect your account key. Avoid distributing it to other users, hard-coding it, or saving it in a plain-text file that is accessible to others.

Regenerate your account key using the Azure Portal if you believe it may have been compromised. To learn how to regenerate your account key, see [How to create, manage, or delete a storage account in the Azure Portal](#).

Storing your connection string

Your application will need to access the connection string at runtime in order to authenticate requests made to Azure Storage. You have a few different options for storing your connection string:

- For an application running on the desktop or on a device, you can store the connection string in an **app.config** file or a **web.config** file. Add the connection string to the **AppSettings** section.
- For an application running in an Azure cloud service, you can store your connection string in the [Azure service configuration schema \(.cscfg\) file](#). Add the connection string to the **ConfigurationSettings** section of the service configuration file.
- You can also use your connection string directly in your code. For most scenarios, however, we recommend that you store your configuration string in a configuration file.

Storing your connection string within a configuration file makes it easy to update the connection string to switch between the storage emulator and an Azure storage account in the cloud. You only need to edit the connection string to point to your target environment.

You can use the [Microsoft Azure Configuration Manager](#) class to access your connection string at runtime regardless of where your application is running.

Create a connection string to the storage emulator

The storage emulator supports a single fixed account and a well-known authentication key for Shared Key authentication. This account and key are the only Shared Key credentials permitted for use with the storage emulator. They are:

```
Account name: devstoreaccount1
Account key: Eby8vdM02xNOcqFlqUwJPLlmEt1CDXJ10UzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPT0tr/KBHBeksoGMGw==
```

NOTE

The authentication key supported by the storage emulator is intended only for testing the functionality of your client authentication code. It does not serve any security purpose. You cannot use your production storage account and key with the storage emulator. Also note that you should not use the development account with production data.

Note that the storage emulator supports connection via HTTP only. However, HTTPS is the recommended protocol for accessing resources in an Azure production storage account.

Connect to the emulator account using a shortcut

The easiest way to connect to the storage emulator from your application is to configure a connection string from within your application's configuration file that references the shortcut `UseDevelopmentStorage=true`. Here's an example of a connection string to the storage emulator in an app.config file:

```
<appSettings>
  <add key="StorageConnectionString" value="UseDevelopmentStorage=true" />
</appSettings>
```

Connect to the emulator account using the well-known account name and key

To create a connection string that references the emulator account name and key, note that you must specify the endpoints for each of the services that you wish to use from the emulator in the connection string. This is necessary so that the connection string will reference the emulator endpoints, which are different than those for a production storage account. For example, the value of your connection string will look like this:

```
DefaultEndpointsProtocol=http;AccountName=devstoreaccount1;
AccountKey=Eby8vdM02xN0cqFlqUwJPLmEt1CDXJ10UzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPT0tr/KBHBeksoMGW==;
BlobEndpoint=http://127.0.0.1:10000/devstoreaccount1;
TableEndpoint=http://127.0.0.1:10002/devstoreaccount1;
QueueEndpoint=http://127.0.0.1:10001/devstoreaccount1;
```

This value is identical to the shortcut shown above, `UseDevelopmentStorage=true`.

Specify an HTTP proxy

You can also specify an HTTP proxy to use when you're testing your service against the storage emulator. This can be useful for observing HTTP requests and responses while you're debugging operations against the storage services. To specify a proxy, add the `DevelopmentStorageProxyUri` option to the connection string, and set its value to the proxy URI. For example, here is a connection string that points to the storage emulator and configures an HTTP proxy:

```
UseDevelopmentStorage=true;DevelopmentStorageProxyUri=http://myProxyUri
```

See [Use the Azure Storage Emulator for Development and Testing](#) for more information about the storage emulator.

Create a connection string to an Azure storage account

To create a connection string to your Azure storage account, use the connection string format below. Indicate whether you want to connect to the storage account through HTTPS (recommended) or HTTP, replace `myAccountName` with the name of your storage account, and replace `myAccountKey` with your account access key:

```
DefaultEndpointsProtocol=[http|https];AccountName=myAccountName;AccountKey=myAccountKey
```

For example, your connection string will look similar to the following sample connection string:

```
DefaultEndpointsProtocol=https;AccountName=storagesample;AccountKey=<account-key>
```

NOTE

Azure Storage supports both HTTP and HTTPS in a connection string; however, using HTTPS is highly recommended.

Create a connection string using a shared access signature

If you possess a shared access signature (SAS) URL that grants you access to resources in a storage account, you can use the SAS in a connection string. Because the SAS includes on the URI the information required to authenticate the request, the SAS URI provides the protocol, the service endpoint, and the necessary credentials to access the resource.

To create a connection string that includes a shared access signature, specify the string in the following format:

```
BlobEndpoint=myBlobEndpoint;  
QueueEndpoint=myQueueEndpoint;  
TableEndpoint=myTableEndpoint;  
FileEndpoint=myFileEndpoint;  
SharedAccessSignature=sasToken
```

Each service endpoint is optional, although the connection string must contain at least one.

NOTE

Using HTTPS with a SAS is recommended as a best practice.

If you are specifying a SAS in a connection string in a configuration file, you may need to encode special characters in the URL.

Service SAS example

Here's an example of a connection string that includes a service SAS for Blob storage:

```
BlobEndpoint=https://storagesample.blob.core.windows.net;SharedAccessSignature=sv=2015-04-05&sr=b&si=tutorial-  
policy-635959936145100803&sig=9aCzs76n0E7y5BpEi2GvsSv433BZa22leDOZXX%2BXXIU%3D
```

And here's an example of the same connection string with encoding of special characters:

```
BlobEndpoint=https://storagesample.blob.core.windows.net;SharedAccessSignature=sv=2015-04-  
05&sr=b&si=tutorial-policy-635959936145100803&sig=9aCzs76n0E7y5BpEi2GvsSv433BZa22leDOZXX%2BXXIU%3D
```

Account SAS example

Here's an example of a connection string that includes an account SAS for Blob and File storage. Note that endpoints for both services are specified:

```
BlobEndpoint=https://storagesample.blob.core.windows.net;  
FileEndpoint=https://storagesample.file.core.windows.net;  
SharedAccessSignature=sv=2015-07-08&sig=iCvQmdZngZNW%2F4vw43j6%2BVz6fndHF5LI639QJba4r8o%3D&spr=https&st=2016-  
04-12T03%3A24%3A31Z&se=2016-04-13T03%3A29%3A31Z&srt=s&ss=bf&sp=rw1
```

And here's an example of the same connection string with URL encoding:

```
BlobEndpoint=https://storagesample.blob.core.windows.net;
FileEndpoint=https://storagesample.file.core.windows.net;
SharedAccessSignature=sv=2015-07-
08&sig=iCvQmdZngZNW%2F4vw43j6%2BVz6fndHF5LI639QJba4r8o%3D&spr=https&st=2016-04-
12T03%3A24%3A31Z&se=2016-04-13T03%3A29%3A31Z&srt=s&ss=bf&sp=rwl
```

Creating a connection string to an explicit storage endpoint

You can explicitly specify the service endpoints in your connection string instead of using the default endpoints.

To create a connection string that specifies an explicit endpoint, specify the complete service endpoint for each service, including the protocol specification (HTTPS (recommended) or HTTP), in the following format:

```
DefaultEndpointsProtocol=[http|https];
BlobEndpoint=myBlobEndpoint;
QueueEndpoint=myQueueEndpoint;
TableEndpoint=myTableEndpoint;
FileEndpoint=myFileEndpoint;
AccountName=myAccountName;
AccountKey=myAccountKey
```

One scenario where you may wish to do specify an explicit endpoint is if you have mapped your Blob storage endpoint to a custom domain. In that case, you can specify your custom endpoint for Blob storage in your connection string, and optionally specify the default endpoints for the other services if your application uses them.

Here are examples of valid connection strings that specify an explicit endpoint for the Blob service:

```
# Blob endpoint only
DefaultEndpointsProtocol=https;
BlobEndpoint=www.mydomain.com;
AccountName=storagesample;
AccountKey=account-key

# All service endpoints
DefaultEndpointsProtocol=https;
BlobEndpoint=www.mydomain.com;
FileEndpoint=myaccount.file.core.windows.net;
QueueEndpoint=myaccount.queue.core.windows.net;
TableEndpoint=myaccount;
AccountName=storagesample;
AccountKey=account-key
```

The endpoint value that is listed in the connection string is used to construct the request URIs to the Blob service, and it dictates the form of any URIs that are returned to your code.

Note that if you choose to omit a service endpoint from the connection string, then you will not be able to use that connection string to access data in that service from your code.

Creating a connection string with an endpoint suffix

To create a connection string for storage service in regions or instances with different endpoint suffixes, such as for Azure China or Azure Governance, use the following connection string format. Indicate whether you want to connect to the storage account through HTTP or HTTPS, replace *myAccountName* with the name of your storage account, replace *myAccountKey* with your account access key, and replace *mySuffix* with the URI suffix:

```
DefaultEndpointsProtocol=[http|https];
AccountName=myAccountName;
AccountKey=myAccountKey;
EndpointSuffix=mySuffix;
```

For example, your connection string should look similar to the following connection string:

```
DefaultEndpointsProtocol=https;
AccountName=storagesample;
AccountKey=<account-key>;
EndpointSuffix=core.chinacloudapi.cn;
```

Parsing a connection string

The [Microsoft Azure Configuration Manager Library for .NET](#) provides a class for parsing a connection string from a configuration file. The [CloudConfigurationManager](#) class parses configuration settings regardless of whether the client application is running on the desktop, on a mobile device, in an Azure virtual machine, or in an Azure cloud service.

To reference the CloudConfigurationManager package, add the following `using` directive:

```
using Microsoft.Azure;      //Namespace for CloudConfigurationManager
```

Here's an example that shows how to retrieve a connection string from a configuration file:

```
// Parse the connection string and return a reference to the storage account.
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));
```

Using the Azure Configuration Manager is optional. You can also use an API like the .NET Framework's [ConfigurationManager](#) class.

Next steps

- [Use the Azure Storage Emulator for Development and Testing](#)
- [Azure Storage Explorers](#)
- [Using Shared Access Signatures \(SAS\)](#)

Use the Azure Storage Emulator for Development and Testing

1/17/2017 • 14 min to read • [Edit on GitHub](#)

Overview

The Microsoft Azure storage emulator provides a local environment that emulates the Azure Blob, Queue, and Table services for development purposes. Using the storage emulator, you can test your application against the storage services locally, without creating an Azure subscription or incurring any costs. When you're satisfied with how your application is working in the emulator, you can switch to using an Azure storage account in the cloud.

NOTE

The storage emulator is available as part of the [Microsoft Azure SDK](#). You can also install the storage emulator using the [standalone installer](#). To configure the storage emulator, you must have administrative privileges on the computer.

The storage emulator currently runs only on Windows.

Data created in one version of the storage emulator is not guaranteed to be accessible when using a different version. If you need to persist your data for the long term, it's recommended that you store that data in an Azure storage account, rather than in the storage emulator.

The storage emulator depends on OData libraries version 5.6. Replacing the OData DLLs used by the storage emulator with higher versions is not supported and causes unexpected behavior. However, any version of OData supported by the storage service may be used to send requests to the emulator.

How the storage emulator works

The storage emulator uses a local Microsoft SQL Server instance and the local file system to emulate the Azure storage services. By default, the storage emulator uses a database in Microsoft SQL Server 2012 Express LocalDB. You can choose to configure the storage emulator to access a local instance of SQL Server instead of the LocalDB instance. For more information, see [Start and initialize the storage emulator](#), below.

You can install SQL Server Management Studio Express to manage your LocalDB installation. The storage emulator connects to SQL Server or LocalDB using Windows authentication.

Some differences in functionality exist between the storage emulator and Azure storage services. For more information about these differences, see [Differences between the storage emulator and Azure Storage](#).

Authenticating requests against the storage emulator

As with Azure Storage in the cloud, every request that you make against the storage emulator must be authenticated, unless it is an anonymous request. You can authenticate requests against the storage emulator using Shared Key authentication or with a shared access signature (SAS).

Authentication with Shared Key credentials

The storage emulator supports a single fixed account and a well-known authentication key for Shared Key

authentication. This account and key are the only Shared Key credentials permitted for use with the storage emulator. They are:

```
Account name: devstoreaccount1  
Account key: Eby8vdM02xNOcqFlqUwJPLlmEt1CDXJ10UzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPT0tr/KBHBeksoGMGw==
```

NOTE

The authentication key supported by the storage emulator is intended only for testing the functionality of your client authentication code. It does not serve any security purpose. You cannot use your production storage account and key with the storage emulator. Also note that you should not use the development account with production data.

Note that the storage emulator supports connection via HTTP only. However, HTTPS is the recommended protocol for accessing resources in an Azure production storage account.

Connect to the emulator account using a shortcut

The easiest way to connect to the storage emulator from your application is to configure a connection string from within your application's configuration file that references the shortcut

`UseDevelopmentStorage=true`. Here's an example of a connection string to the storage emulator in an app.config file:

```
<appSettings>  
  <add key="StorageConnectionString" value="UseDevelopmentStorage=true" />  
</appSettings>
```

Connect to the emulator account using the well-known account name and key

To create a connection string that references the emulator account name and key, note that you must specify the endpoints for each of the services that you wish to use from the emulator in the connection string. This is necessary so that the connection string will reference the emulator endpoints, which are different than those for a production storage account. For example, the value of your connection string will look like this:

```
DefaultEndpointsProtocol=http;AccountName=devstoreaccount1;  
AccountKey=Eby8vdM02xNOcqFlqUwJPLlmEt1CDXJ10UzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPT0tr/KBHBeksoGMGw==;  
BlobEndpoint=http://127.0.0.1:10000/devstoreaccount1;  
TableEndpoint=http://127.0.0.1:10002/devstoreaccount1;  
QueueEndpoint=http://127.0.0.1:10001/devstoreaccount1;
```

This value is identical to the shortcut shown above, `UseDevelopmentStorage=true`.

Specify an HTTP proxy

You can also specify an HTTP proxy to use when you're testing your service against the storage emulator. This can be useful for observing HTTP requests and responses while you're debugging operations against the storage services. To specify a proxy, add the `DevelopmentStorageProxyUri` option to the connection string, and set its value to the proxy URI. For example, here is a connection string that points to the storage emulator and configures an HTTP proxy:

```
UseDevelopmentStorage=true;DevelopmentStorageProxyUri=http://myProxyUri
```

For more information on connection strings, see [Configure Azure Storage Connection Strings](#).

Authentication with a shared access signature

Some Azure storage client libraries, such as the Xamarin library, only support authentication with a shared access signature (SAS) token. Create this SAS token using a tool or application that supports

Shared Key authentication. An easy way to generate the SAS token is via Azure PowerShell:

1. Install Azure PowerShell if you haven't already. Using the latest version of the Azure PowerShell cmdlets is recommended. See [How to install and configure Azure PowerShell](#) for installation instructions.
2. Open Azure PowerShell and run the following commands, replacing `ACCOUNT_NAME` and `ACCOUNT_KEY==` with your own credentials and `CONTAINER_NAME` with a name of your choosing:

```
$context = New-AzureStorageContext -StorageAccountName "ACCOUNT_NAME" -StorageAccountKey  
"ACCOUNT_KEY=="  
  
New-AzureStorageContainer CONTAINER_NAME -Permission Off -Context $context  
  
$now = Get-Date  
  
New-AzureStorageContainerSASToken -Name CONTAINER_NAME -Permission rwdl -ExpiryTime $now.AddDays(1.0) -  
Context $context -FullUri
```

The resulting shared access signature URI for the new container should be similar to:

```
https://storageaccount.blob.core.windows.net/sascontainer?sv=2012-02-12&se=2015-07-  
08T00%3A12%3A08Z&sr=c&sp=w&sig=t%2BbzU9%2B7ry4okULN9S0wst%2F8MCUhTjrHyV9rDNLSe8g%3Dss
```

The shared access signature created with this example is valid for one day. The signature grants full access (i.e. read, write, delete, list) to blobs within the container.

For more information on shared access signatures, see [Using Shared Access Signatures \(SAS\)](#).

Start and initialize the storage emulator

To start the Azure storage emulator, select the Start button or press the Windows key. Begin typing **Azure Storage Emulator**, and select the emulator from the list of applications.

When the emulator is running, you'll see an icon in the Windows taskbar notification area.

When the storage emulator starts, a command-line window will appear. You can use this command-line window to start and stop the storage emulator as well as clear data, get status, and initialize the emulator. For more information, see [Storage Emulator Command-Line Tool Reference](#).

When the command-line window is closed, the storage emulator will continue to run. To bring up the command line again, follow the preceding steps as if starting the storage emulator.

The first time you run the storage emulator, the local storage environment is initialized for you. The initialization process creates a database in LocalDB and reserves HTTP ports for each local storage service.

The storage emulator is installed by default to the C:\Program Files (x86)\Microsoft SDKs\Azure\Storage Emulator directory.

Initialize the storage emulator to use a different SQL database

You can use the storage emulator command-line tool to initialize the storage emulator to point to a SQL database instance other than the default LocalDB instance. You must be running the command-line tool with administrative privileges to initialize the back-end database for the storage emulator:

1. Click the **Start** button or press the **Windows** key. Begin typing **Azure Storage Emulator** and select it when it appears to bring up the storage emulator command-line tool.
2. In the command prompt window, type the following command, where `<SQLServerInstance>` is the name of the SQL Server instance. To use LocalDb, specify `(localdb)\v11.0` as the SQL Server instance.

```
AzureStorageEmulator init /server <SQLServerInstance>
```

You can also use the following command, which directs the emulator to use the default SQL Server instance:

```
AzureStorageEmulator init /server .\
```

Or, you can use the following command, which reinitializes the database to the default LocalDB instance:

```
AzureStorageEmulator init /forceCreate
```

For more information about these commands, see [Storage Emulator Command-Line Tool Reference](#).

Addressing resources in the storage emulator

The service endpoints for the storage emulator are different from those of an Azure storage account. The difference is because the local computer does not perform domain name resolution, requiring the storage emulator endpoints to be local addresses.

When you address a resource in an Azure storage account, use the following scheme. The account name is part of the URI host name, and the resource being addressed is part of the URI path:

```
<http|https>://<account-name>.<service-name>.core.windows.net/<resource-path>
```

For example, the following URI is a valid address for a blob in an Azure storage account:

```
https://myaccount.blob.core.windows.net/mycontainer/myblob.txt
```

In the storage emulator, because the local computer does not perform domain name resolution, the account name is part of the URI path instead of the host name. You use the following scheme for a resource running in the storage emulator:

```
http://<local-machine-address>:<port>/<account-name>/<resource-path>
```

For example, the following address might be used for accessing a blob in the storage emulator:

```
http://127.0.0.1:10000/myaccount/mycontainer/myblob.txt
```

The service endpoints for the storage emulator are:

```
Blob Service: http://127.0.0.1:10000/<account-name>/<resource-path>
Queue Service: http://127.0.0.1:10001/<account-name>/<resource-path>
Table Service: http://127.0.0.1:10002/<account-name>/<resource-path>
```

Addressing the account secondary with RA-GRS

Beginning with version 3.1, the storage emulator account supports read-access geo-redundant replication (RA-GRS). For storage resources both in the cloud and in the local emulator, you can access the secondary location by appending -secondary to the account name. For example, the following address might be used for accessing a blob using the read-only secondary in the storage emulator:

```
http://127.0.0.1:10000/myaccount-secondary/mycontainer/myblob.txt
```

NOTE

For programmatic access to the secondary with the storage emulator, use the Storage Client Library for .NET version 3.2 or later. See the [Microsoft Azure Storage Client Library for .NET](#) for details.

Storage emulator command-line tool reference

Starting in version 3.0, when you launch the Storage Emulator, a command-line window pops up. Use the command-line window to start and stop the emulator as well as to query for status and perform other operations.

NOTE

If you have the Microsoft Azure compute emulator installed, a system tray icon appears when you launch the Storage Emulator. Right-click on the icon to reveal a menu, which provides a graphical way to start and stop the Storage Emulator.

Command Line Syntax

```
AzureStorageEmulator [start] [stop] [status] [clear] [init] [help]
```

Options

To view the list of options, type `/help` at the command prompt.

OPTION	DESCRIPTION	COMMAND	ARGUMENTS
Start	Starts up the storage emulator.	<code>AzureStorageEmulator start [-inprocess]</code>	<code>-inprocess</code> : Start the emulator in the current process instead of creating a new process.
Stop	Stops the storage emulator.	<code>AzureStorageEmulator stop</code>	
Status	Prints the status of the storage emulator.	<code>AzureStorageEmulator status</code>	
Clear	Clears the data in all services specified on the command line.	<code>AzureStorageEmulator clear [blob] [table] [queue] [all]</code>	<code>blob</code> : Clears blob data. <code>queue</code> : Clears queue data. <code>table</code> : Clears table data. <code>all</code> : Clears all data in all services.

OPTION	DESCRIPTION	COMMAND	ARGUMENTS
Init	Performs one-time initialization to set up the emulator.	<pre>AzureStorageEmulator.exe init [-server serverName] [-sqlinstance instanceName] [-forcecreate] [-inprocess]</pre>	-server serverName\instanceName: Specifies the server hosting the SQL instance. -sqlinstance instanceName: Specifies the name of the SQL instance to be used in the default server instance. -forcecreate: Forces creation of the SQL database, even if it already exists. -inprocess: Performs initialization in the current process instead of spawning a new process. The current process must be launched with elevated permissions to perform initialization.

Differences between the storage emulator and Azure Storage

Because the storage emulator is an emulated environment running in a local SQL instance, there are differences in functionality between the emulator and an Azure storage account in the cloud:

- The storage emulator supports only a single fixed account and a well-known authentication key.
- The storage emulator is not a scalable storage service and does not support a large number of concurrent clients.
- As described in [Addressing resources in the storage emulator](#), resources are addressed differently in the storage emulator versus an Azure storage account. This difference is because domain name resolution is available in the cloud but not on the local computer.
- Beginning with version 3.1, the storage emulator account supports read-access geo-redundant replication (RA-GRS). In the emulator, all accounts have RA-GRS enabled, and there is never any lag between the primary and secondary replicas. The Get Blob Service Stats, Get Queue Service Stats, and Get Table Service Stats operations are supported on the account secondary and will always return the value of the `LastSyncTime` response element as the current time according to the underlying SQL database.

For programmatic access to the secondary with the storage emulator, use the Storage Client Library for .NET version 3.2 or later. See the [Microsoft Azure Storage Client Library for .NET](#) for details.

- The File service and SMB protocol service endpoints are not currently supported in the storage emulator.
- If you use a version of the storage services that is not yet supported by the emulator, the storage emulator returns a VersionNotSupportedByEmulator error (HTTP status code 400 - Bad Request).

Differences for Blob storage

The following differences apply to Blob storage in the emulator:

- The storage emulator only supports blob sizes up to 2 GB.
- Incremental copy allows snapshots from overwritten blobs to be copied, which returns a failure on the service.

- Get Page Ranges Diff does not work between snapshots copied using Incremental Copy Blob.
- A Put Blob operation may succeed against a blob that exists in the storage emulator with an active lease, even if the lease ID has not been specified in the request.
- Append Blob operations are not supported by the emulator. Attempting an operation on an append blob returns a FeatureNotSupportedByEmulator error (HTTP status code 400 - Bad Request).

Differences for Table storage

The following differences apply to Table storage in the emulator:

- Date properties in the Table service in the storage emulator support only the range supported by SQL Server 2005 (*i.e.*, they are required to be later than January 1, 1753). All dates before January 1, 1753 are changed to this value. The precision of dates is limited to the precision of SQL Server 2005, meaning that dates are precise to 1/300th of a second.
- The storage emulator supports partition key and row key property values of less than 512 bytes each. Additionally, the total size of the account name, table name, and key property names together cannot exceed 900 bytes.
- The total size of a row in a table in the storage emulator is limited to less than 1 MB.
- In the storage emulator, properties of data type `Edm.Guid` or `Edm.Binary` support only the `Equal (eq)` and `NotEqual (ne)` comparison operators in query filter strings.

Differences for Queue storage

There are no differences specific to Queue storage in the emulator.

Storage emulator release notes

Version 4.6

- The storage emulator now supports version 2016-05-31 of the storage services on Blob, Queue, and Table service endpoints.

Version 4.5

- Fixed a bug that caused initialization and installation of the storage emulator to fail when the backing database was renamed.

Version 4.4

- The storage emulator now supports version 2015-12-11 of the storage services on Blob, Queue, and Table service endpoints.
- The storage emulator's garbage collection of blob data is now more efficient when dealing with large numbers of blobs.
- Fixed a bug that caused container ACL XML to be validated slightly differently from how the storage service does it.
- Fixed a bug that sometimes caused max and min DateTime values to be reported in the incorrect time zone.

Version 4.3

- The storage emulator now supports version 2015-07-08 of the storage services on Blob, Queue, and Table service endpoints.

Version 4.2

- The storage emulator now supports version 2015-04-05 of the storage services on Blob, Queue, and Table service endpoints.

Version 4.1

- The storage emulator now supports version 2015-02-21 of the storage services on Blob, Queue, and

Table service endpoints, except for the new Append Blob features.

- If you use a version of the storage services that is not yet supported by the emulator, the emulator returns a meaningful error message. We recommend using the latest version of the emulator. If you encounter a VersionNotSupportedException (HTTP status code 400 - Bad Request), please download the latest version of the storage emulator.
- Fixed a bug wherein a race condition caused table entity data to be incorrect during concurrent merge operations.

Version 4.0

- The storage emulator executable has been renamed to *AzureStorageEmulator.exe*.

Version 3.2

- The storage emulator now supports version 2014-02-14 of the storage services on Blob, Queue, and Table service endpoints. File service endpoints are not currently supported in the storage emulator. See [Versioning for the Azure Storage Services](#) for details about version 2014-02-14.

Version 3.1

- Read-access geo-redundant storage (RA-GRS) is now supported in the storage emulator. The Get Blob Service Stats, Get Queue Service Stats, and Get Table Service Stats APIs are supported for the account secondary and will always return the value of the LastSyncTime response element as the current time according to the underlying SQL database. For programmatic access to the secondary with the storage emulator, use the Storage Client Library for .NET version 3.2 or later. See the Microsoft Azure Storage Client Library for .NET Reference for details.

Version 3.0

- The Azure storage emulator is no longer shipped in the same package as the compute emulator.
- The storage emulator graphical user interface is deprecated in favor of a scriptable command-line interface. For details on the command-line interface, see [Storage Emulator Command-Line Tool Reference](#). The graphical interface will continue to be present in version 3.0, but it can only be accessed when the Compute Emulator is installed by right-clicking on the system tray icon and selecting Show Storage Emulator UI.
- Version 2013-08-15 of the Azure storage services is now fully supported. (Previously this version was only supported by Storage Emulator version 2.2.1 Preview.)

Set and Retrieve Properties and Metadata

1/17/2017 • 2 min to read • [Edit on GitHub](#)

Overview

Objects in Azure Storage support system properties and user-defined metadata, in addition to the data they contain:

- **System properties.** System properties exist on each storage resource. Some of them can be read or set, while others are read-only. Under the covers, some system properties correspond to certain standard HTTP headers. The Azure storage client library maintains these for you.
- **User-defined metadata.** User-defined metadata is metadata that you specify on a given resource, in the form of a name-value pair. You can use metadata to store additional values with a storage resource; these values are for your own purposes only and do not affect how the resource behaves.

Retrieving property and metadata values for a storage resource is a two-step process. Before you can read these values, you must explicitly fetch them by calling the **FetchAttributes** method.

IMPORTANT

Property and metadata values for a storage resource are not populated unless you call one of the **FetchAttributes** methods.

Setting and Retrieving Properties

To retrieve property values, call the **FetchAttributes** method on your blob or container to populate the properties, then read the values.

To set properties on an object, specify the property value, then call the **SetProperties** method.

The following code example creates a container and writes some of its property values to a console window:

```
//Parse the connection string for the storage account.
const string ConnectionString = "DefaultEndpointsProtocol=https;AccountName=account-name;AccountKey=account-key";
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(ConnectionString);

//Create the service client object for credentialled access to the Blob service.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Retrieve a reference to a container.
CloudBlobContainer container = blobClient.GetContainerReference("mycontainer");

// Create the container if it does not already exist.
container.CreateIfNotExists();

// Fetch container properties and write out their values.
container.FetchAttributes();
Console.WriteLine("Properties for container {0}", container.StorageUri.PrimaryUri.ToString());
Console.WriteLine("LastModifiedUTC: {0}", container.Properties.LastModified.ToString());
Console.WriteLine("ETag: {0}", container.Properties.ETag);
Console.WriteLine();
```

Setting and Retrieving Metadata

You can specify metadata as one or more name-value pairs on a blob or container resource. To set metadata, add name-value pairs to the **Metadata** collection on the resource, then call the **SetMetadata** method to save the values to the service.

NOTE

The name of your metadata must conform to the naming conventions for C# identifiers.

The following code example sets metadata on a container. One value is set using the collection's **Add** method. The other value is set using implicit key/value syntax. Both are valid.

```
public static void AddContainerMetadata(CloudBlobContainer container)
{
    //Add some metadata to the container.
    container.Metadata.Add("docType", "textDocuments");
    container.Metadata["category"] = "guidance";

    //Set the container's metadata.
    container.SetMetadata();
}
```

To retrieve metadata, call the **FetchAttributes** method on your blob or container to populate the **Metadata** collection, then read the values, as shown in the example below.

```
public static void ListContainerMetadata(CloudBlobContainer container)
{
    //Fetch container attributes in order to populate the container's properties and metadata.
    container.FetchAttributes();

    //Enumerate the container's metadata.
    Console.WriteLine("Container metadata:");
    foreach (var metadataItem in container.Metadata)
    {
        Console.WriteLine("\tKey: {0}", metadataItem.Key);
        Console.WriteLine("\tValue: {0}", metadataItem.Value);
    }
}
```

See Also

- [Azure Storage Client Library for .NET Reference](#)
- [Azure Storage Client Library for .NET package](#)

Using Azure PowerShell with Azure Storage

1/17/2017 • 32 min to read • [Edit on GitHub](#)

Overview

Azure PowerShell is a module that provides cmdlets to manage Azure through Windows PowerShell. It is a task-based command-line shell and scripting language designed especially for system administration. With PowerShell, you can easily control and automate the administration of your Azure services and applications. For example, you can use the cmdlets to perform the same tasks that you can perform through the [Azure portal](#).

In this guide, we'll explore how to use the [Azure Storage Cmdlets](#) to perform a variety of development and administration tasks with Azure Storage.

This guide assumes that you have prior experience using [Azure Storage](#) and [Windows PowerShell](#). The guide provides a number of scripts to demonstrate the usage of PowerShell with Azure Storage. You should update the script variables based on your configuration before running each script.

The first section in this guide provides a quick glance at Azure Storage and PowerShell. For detailed information and instructions, start from the [Prerequisites for using Azure PowerShell with Azure Storage](#).

Getting started with Azure Storage and PowerShell in 5 minutes

This section shows you how to access Azure Storage via PowerShell in 5 minutes.

New to Azure: Get a Microsoft Azure subscription and a Microsoft account associated with that subscription. For information on Azure purchase options, see [Free Trial](#), [Purchase Options](#), and [Member Offers](#) (for members of MSDN, Microsoft Partner Network, and BizSpark, and other Microsoft programs).

See [Assigning administrator roles in Azure Active Directory \(Azure AD\)](#) for more information about Azure subscriptions.

After creating a Microsoft Azure subscription and account:

1. Download and install [Azure PowerShell](#).
2. Start Windows PowerShell Integrated Scripting Environment (ISE): In your local computer, go to the **Start** menu. Type **Administrative Tools** and click to run it. In the **Administrative Tools** window, right-click **Windows PowerShell ISE**, click **Run as Administrator**.
3. In **Windows PowerShell ISE**, click **File > New** to create a new script file.
4. Now, we'll give you a simple script that shows basic PowerShell commands to access Azure Storage. The script will first ask your Azure account credentials to add your Azure account to the local PowerShell environment. Then, the script will set the default Azure subscription and create a new storage account in Azure. Next, the script will create a new container in this new storage account and upload an existing image file (blob) to that container. After the script lists all blobs in that container, it will create a new destination directory in your local computer and download the image file.
5. In the following code section, select the script between the remarks **#begin** and **#end**. Press **CTRL+C** to copy it to the clipboard.

```

# begin

# Update with the name of your subscription.
$SubscriptionName = "YourSubscriptionName"

# Give a name to your new storage account. It must be lowercase!
$StorageAccountName = "yourstorageaccountname"

# Choose "West US" as an example.
$Location = "West US"

# Give a name to your new container.
$ContainerName = "imagecontainer"

# Have an image file and a source directory in your local computer.
$imageToUpload = "C:\Images\HelloWorld.png"

# A destination directory in your local computer.
$DestinationFolder = "C:\DownloadImages"

# Add your Azure account to the local PowerShell environment.
Add-AzureAccount

# Set a default Azure subscription.
Select-AzureSubscription -SubscriptionName $SubscriptionName -Default

# Create a new storage account.
New-AzureStorageAccount -StorageAccountName $StorageAccountName -Location $Location

# Set a default storage account.
Set-AzureSubscription -CurrentStorageAccountName $StorageAccountName -SubscriptionName
$SubscriptionName

# Create a new container.
New-AzureStorageContainer -Name $ContainerName -Permission Off

# Upload a blob into a container.
Set-AzureStorageBlobContent -Container $ContainerName -File $ImageToUpload

# List all blobs in a container.
Get-AzureStorageBlob -Container $ContainerName

# Download blobs from the container:
# Get a reference to a list of all blobs in a container.
$blobs = Get-AzureStorageBlob -Container $ContainerName

# Create the destination directory.
New-Item -Path $DestinationFolder -ItemType Directory -Force

# Download blobs into the local destination directory.
$blobs | Get-AzureStorageBlobContent -Destination $DestinationFolder

# end

```

6. In **Windows PowerShell ISE**, press CTRL+V to copy the script. Click **File** > **Save**. In the **Save As** dialog window, type the name of the script file, such as "mystorageScript." Click **Save**.
7. Now, you need to update the script variables based on your configuration settings. You must update the **\$SubscriptionName** variable with your own subscription. You can keep the other variables as specified in the script or update them as you wish.
 - **\$SubscriptionName:** You must update this variable with your own subscription name. Follow one of the following three ways to locate the name of your subscription:
 - a. In **Windows PowerShell ISE**, click **File** > **New** to create a new script file. Copy the following script to the new script file and click **Debug** > **Run**. The following script will first ask

your Azure account credentials to add your Azure account to the local PowerShell environment and then show all the subscriptions that are connected to the local PowerShell session. Take a note of the name of the subscription that you want to use while following this tutorial:

```
Add-AzureAccount  
Get-AzureSubscription | Format-Table SubscriptionName, IsDefault, IsCurrent,  
CurrentStorageAccountName
```

b. To locate and copy your subscription name in the [Azure portal](#), in the Hub menu on the left, click **Subscriptions**. Copy the name of subscription that you want to use while running the scripts in this guide.

The screenshot shows the Microsoft Azure portal interface. On the left, there is a vertical navigation bar with icons for HOME, NOTIFICATIONS, BROWSE, ACTIVE, and BILLING. A blue button at the bottom left says '+ NEW'. The main content area has two tabs: 'Browse' on the left and 'Subscriptions' on the right. The 'Subscriptions' tab is active, displaying a table with columns: SUBSCRIPTION, SUBSCRIPTION ID, and SUBSCRIPTION STATUS. The table lists several subscriptions:

SUBSCRIPTION	SUBSCRIPTION ID	SUBSCRIPTION STATUS
Internal Consumption	3c791a42-	Active
DnA_SQL_DSS	ddce457a-	Active
MSDNNonDallas	f7f09258-	Active
SQLAzure_Project1	29747c44-	Cancelled
SQLAzure_Project2	fbca1d74-	Cancelled
SQLAzure_Introductory...	50f10c263-	Cancelled
Windows Azure Internal...	5d553972-	Cancelled

c. To locate and copy your subscription name in the [Azure Classic Portal](#), scroll down and click **Settings** on the left side of the portal. Click **Subscriptions** to see a list of your subscriptions. Copy the name of subscription that you want to use while running the scripts given in this guide.

SUBSCRIPTION	SUBSCRIPTION ID	ACCOUNT ADMINISTRATOR	DIRECTORY
DnA_SQL_DSS	ddce457a-[REDACTED]	testaccount@microsoft.com	Microsoft (microsoft.on...
Internal Consumption	3c791a42-[REDACTED]	testaccount@microsoft.com	Microsoft (microsoft.on...
MSDNonDallas	f7f09258-[REDACTED]	testaccount@microsoft.com	Microsoft (microsoft.on...

- **\$StorageAccountName:** Use the given name in the script or enter a new name for your storage account. **Important:** The name of the storage account must be unique in Azure. It must be lowercase, too!
- **\$Location:** Use the given "West US" in the script or choose other Azure locations, such as East US, North Europe, and so on.
- **\$ContainerName:** Use the given name in the script or enter a new name for your container.
- **\$ImageToUpload:** Enter a path to a picture on your local computer, such as: "C:\Images\HelloWorld.png".
- **\$DestinationFolder:** Enter a path to a local directory to store files downloaded from Azure Storage, such as: "C:\Download\Images".

8. After updating the script variables in the "mystoragescript.ps1" file, click **File > Save**. Then, click **Debug > Run** or press **F5** to run the script.

After the script runs, you should have a local destination folder that includes the downloaded image file. The following screenshot shows an example output:

```
PS C:\> # Download blobs into the local destination directory.
PS C:\> $blobs | Get-AzureStorageBlobContent -Destination $DestinationFolder

Container Uri: https://yourstorageaccountname.blob.core.windows.net/imagecontainer
Name          BlobType      Length       ContentType      LastModified      SnapshotTime
----          -----      -----       -----          -----          -----
HelloWorld.png BlockBlob    65477       application/octet-... 1/22/2015 5:37:24 ...
VERBOSE: Transfer Summary
-----
Total: 1.
Successful: 1.
Failed: 0.
```

NOTE

The "Getting started with Azure Storage and PowerShell in 5 minutes" section provided a quick introduction on how to use Azure PowerShell with Azure Storage. For detailed information and instructions, we encourage you to read the following sections.

Prerequisites for using Azure PowerShell with Azure Storage

You need an Azure subscription and account to run the PowerShell cmdlets given in this guide, as described above.

Azure PowerShell is a module that provides cmdlets to manage Azure through Windows PowerShell. For information on installing and setting up Azure PowerShell, see [How to install and configure Azure PowerShell](#). We recommend that you download and install or upgrade to the latest Azure PowerShell module before using this guide.

You can run the cmdlets in the standard Windows PowerShell console or the Windows PowerShell Integrated Scripting Environment (ISE). For example, to open **Windows PowerShell ISE**, go to the Start menu, type Administrative Tools and click to run it. In the Administrative Tools window, right-click Windows PowerShell ISE, click Run as Administrator.

How to manage storage accounts in Azure

Let's take a look at managing storage accounts in Azure with PowerShell

How to set a default Azure subscription

To manage Azure Storage using Azure PowerShell, you need to authenticate your client environment with Azure via Azure Active Directory authentication or certificate-based authentication. For detailed information, see [How to install and configure Azure PowerShell](#) tutorial. This guide uses the Azure Active Directory authentication.

1. In Windows PowerShell ISE, type the following command to add your Azure account to the local PowerShell environment:

```
Add-AzureAccount
```

2. In the "Sign in to Microsoft Azure" window, type the email address and password associated with your account. Azure authenticates and saves the credential information, and then closes the window.
3. Next, run the following command to view the Azure accounts in your local PowerShell environment, and verify that your account is listed:

```
Get-AzureAccount
```

4. Then, run the following cmdlet to view all the subscriptions that are connected to the local PowerShell session, and verify that your subscription is listed:

```
Get-AzureSubscription | Format-Table SubscriptionName, IsDefault, IsCurrent,  
CurrentStorageAccountName`
```

5. To set a default Azure subscription, run the Select-AzureSubscription cmdlet:

```
$SubscriptionName = 'Your subscription Name'  
Select-AzureSubscription -SubscriptionName $SubscriptionName -Default
```

6. Verify the name of the default subscription by running the Get-AzureSubscription cmdlet:

```
Get-AzureSubscription -Default
```

7. To see all the available PowerShell cmdlets for Azure Storage, run:

```
Get-Command -Module Azure -Noun *Storage*
```

How to create a new Azure storage account

To use Azure storage, you will need a storage account. You can create a new Azure storage account after you have configured your computer to connect to your subscription.

1. Run the Get-AzureLocation cmdlet to find all the available datacenter locations:

```
Get-AzureLocation | Format-Table -Property Name, AvailableServices, StorageAccountTypes
```

2. Next, run the New-AzureStorageAccount cmdlet to create a new storage account. The following example creates a new storage account in the "West US" datacenter.

```
$location = "West US"  
$StorageAccountName = "yourstorageaccount"  
New-AzureStorageAccount -StorageAccountName $StorageAccountName -Location $location
```

IMPORTANT

The name of your storage account must be unique within Azure and must be lowercase. For naming conventions and restrictions, see [About Azure Storage Accounts](#) and [Naming and Referencing Containers, Blobs, and Metadata](#).

How to set a default Azure storage account

You can have multiple storage accounts in your subscription. You can choose one of them and set it as the default storage account for all the storage commands in the same PowerShell session. This enables you to run the Azure PowerShell storage commands without specifying the storage context explicitly.

1. To set a default storage account for your subscription, you can run the Set-AzureSubscription cmdlet.

```
$SubscriptionName = "Your subscription name"  
$StorageAccountName = "yourstorageaccount"  
Set-AzureSubscription -CurrentStorageAccountName $StorageAccountName -SubscriptionName  
$SubscriptionName
```

2. Next, run the Get-AzureSubscription cmdlet to verify that the storage account is associated with your default subscription account. This command returns the subscription properties on the current subscription including its current storage account.

```
Get-AzureSubscription -Current
```

How to list all Azure storage accounts in a subscription

Each Azure subscription can have up to 100 storage accounts. For the most up-to-date information on limits, see [Azure Subscription and Service Limits, Quotas, and Constraints](#).

Run the following cmdlet to find out the name and status of the storage accounts in the current subscription:

```
Get-AzureStorageAccount | Format-Table -Property StorageAccountName, Location, AccountType,  
StorageAccountStatus
```

How to create an Azure storage context

Azure storage context is an object in PowerShell to encapsulate the storage credentials. Using a storage context while running any subsequent cmdlet enables you to authenticate your request without specifying the storage account and its access key explicitly. You can create a storage context in many ways, such as

using storage account name and access key, shared access signature (SAS) token, connection string, or anonymous. For more information, see [New-AzureStorageContext](#).

Use one of the following three ways to create a storage context:

- Run the [Get-AzureStorageKey](#) cmdlet to find out the primary storage access key for your Azure storage account. Next, call the [New-AzureStorageContext](#) cmdlet to create a storage context:

```
$StorageAccountName = "yourstorageaccount"  
$StorageAccountKey = Get-AzureStorageKey -StorageAccountName $StorageAccountName  
$Ctx = New-AzureStorageContext $StorageAccountName -StorageAccountKey $StorageAccountKey.Primary
```

- Generate a shared access signature token for an Azure storage container and use it to create a storage context:

```
$sasToken = New-AzureStorageContainerSASToken -Container abc -Permission r  
$Ctx = New-AzureStorageContext -StorageAccountName $StorageAccountName -SasToken $sasToken
```

For more information, see [New-AzureStorageContainerSASToken](#) and [Using Shared Access Signatures \(SAS\)](#).

- In some cases, you may want to specify the service endpoints when you create a new storage context. This might be necessary when you have registered a custom domain name for your storage account with the Blob service or you want to use a shared access signature for accessing storage resources. Set the service endpoints in a connection string and use it to create a new storage context as shown below:

```
$ConnectionString = "DefaultEndpointsProtocol=http;BlobEndpoint=<blobEndpoint>;QueueEndpoint=<QueueEndpoint>;TableEndpoint=<TableEndpoint>;AccountName=<AccountName>;AccountKey=<AccountKey>"  
$Ctx = New-AzureStorageContext -ConnectionString $ConnectionString
```

For more information on how to configure a storage connection string, see [Configuring Connection Strings](#).

Now that you have set up your computer and learned how to manage subscriptions and storage accounts using Azure PowerShell, go to the next section to learn how to manage Azure blobs and blob snapshots.

How to retrieve and regenerate Azure storage keys

An Azure Storage account comes with two account keys. You can use the following cmdlet to retrieve your keys.

```
Get-AzureStorageKey -StorageAccountName "yourstorageaccount"
```

Use the following cmdlet to retrieve a specific key. Valid values are Primary and Secondary.

```
(Get-AzureStorageKey -StorageAccountName $StorageAccountName).Primary  
(Get-AzureStorageKey -StorageAccountName $StorageAccountName).Secondary
```

If you would like to regenerate your keys, use the following cmdlet. Valid values for -KeyType are "Primary" and "Secondary"

```
New-AzureStorageKey -StorageAccountName $StorageAccountName -KeyType "Primary"  
New-AzureStorageKey -StorageAccountName $StorageAccountName -KeyType "Secondary"
```

How to manage Azure blobs

Azure Blob storage is a service for storing large amounts of unstructured data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. This section assumes that you are already familiar with the Azure Blob Storage Service concepts. For detailed information, see [Get started with Blob storage using .NET](#) and [Blob Service Concepts](#).

How to create a container

Every blob in Azure storage must be in a container. You can create a private container using the `New-AzureStorageContainer` cmdlet:

```
$StorageContainerName = "yourcontainername"  
New-AzureStorageContainer -Name $StorageContainerName -Permission Off
```

NOTE

There are three levels of anonymous read access: **Off**, **Blob**, and **Container**. To prevent anonymous access to blobs, set the `Permission` parameter to **Off**. By default, the new container is private and can be accessed only by the account owner. To allow anonymous public read access to blob resources, but not to container metadata or to the list of blobs in the container, set the `Permission` parameter to **Blob**. To allow full public read access to blob resources, container metadata, and the list of blobs in the container, set the `Permission` parameter to **Container**. For more information, see [Manage anonymous read access to containers and blobs](#).

How to upload a blob into a container

Azure Blob Storage supports block blobs and page blobs. For more information, see [Understanding Block Blobs, Append Blobs, and Page Blobs](#).

To upload blobs in to a container, you can use the `Set-AzureStorageBlobContent` cmdlet. By default, this command uploads the local files to a block blob. To specify the type for the blob, you can use the `-BlobType` parameter.

The following example runs the `Get-ChildItem` cmdlet to get all the files in the specified folder, and then passes them to the next cmdlet by using the pipeline operator. The `Set-AzureStorageBlobContent` cmdlet uploads the local files to your container:

```
Get-ChildItem -Path C:\Images\* | Set-AzureStorageBlobContent -Container "yourcontainername"
```

How to download blobs from a container

The following example demonstrates how to download blobs from a container. The example first establishes a connection to Azure Storage using the storage account context, which includes the storage account name and its primary access key. Then, the example retrieves a blob reference using the `Get-AzureStorageBlob` cmdlet. Next, the example uses the `Get-AzureStorageBlobContent` cmdlet to download blobs into the local destination folder.

```

#define the variables.
$ContainerName = "yourcontainername"
$DestinationFolder = "C:\DownloadImages"

#define the storage account and context.
$StorageAccountName = "yourstorageaccount"
$StorageAccountKey = "Storage key for yourstorageaccount ends with =="
$Ctx = New-AzureStorageContext -StorageAccountName $StorageAccountName -StorageAccountKey
$StorageAccountKey

#List all blobs in a container.
$blobs = Get-AzureStorageBlob -Container $ContainerName -Context $Ctx

#Download blobs from a container.
New-Item -Path $DestinationFolder -ItemType Directory -Force
$blobs | Get-AzureStorageBlobContent -Destination $DestinationFolder -Context $Ctx

```

How to copy blobs from one storage container to another

You can copy blobs across storage accounts and regions asynchronously. The following example demonstrates how to copy blobs from one storage container to another in two different storage accounts. The example first sets variables for source and destination storage accounts, and then creates a storage context for each account. Next, the example copies blobs from the source container to the destination container using the [Start-AzureStorageBlobCopy](#) cmdlet. The example assumes that the source and destination storage accounts and containers already exist.

```

#define the source storage account and context.
$SourceStorageAccountName = "yoursourcestorageaccount"
$SourceStorageAccountKey = "Storage key for yoursourcestorageaccount"
$SrcContainerName = "yoursrccontainername"
$SourceContext = New-AzureStorageContext -StorageAccountName $SourceStorageAccountName -StorageAccountKey
$SourceStorageAccountKey

#define the destination storage account and context.
$DestStorageAccountName = "yourdeststorageaccount"
$DestStorageAccountKey = "Storage key for yourdeststorageaccount"
$DestContainerName = "destcontainername"
$DestContext = New-AzureStorageContext -StorageAccountName $DestStorageAccountName -StorageAccountKey
$DestStorageAccountKey

#get a reference to blobs in the source container.
$blobs = Get-AzureStorageBlob -Container $SrcContainerName -Context $SourceContext

#Copy blobs from one container to another.
$blobs | Start-AzureStorageBlobCopy -DestContainer $DestContainerName -DestContext $DestContext

```

Note that this example performs an asynchronous copy. You can monitor the status of each copy by running the [Get-AzureStorageBlobCopyState](#) cmdlet.

How to copy blobs from a secondary location

You can copy blobs from the secondary location of a RA-GRS-enabled account.

```

#define secondary storage context using a connection string constructed from secondary endpoints.
$SrcContext = New-AzureStorageContext -ConnectionString
"DefaultEndpointsProtocol=https;AccountName=***;AccountKey=***;BlobEndpoint=http://***-
secondary.blob.core.windows.net;FileEndpoint=http://***-
secondary.file.core.windows.net;QueueEndpoint=http://***-secondary.queue.core.windows.net;
TableEndpoint=http://***-secondary.table.core.windows.net;"
Start-AzureStorageBlobCopy -Container *** -Blob *** -Context $SrcContext -DestContainer *** -DestBlob ***
-DestContext $DestContext

```

How to delete a blob

To delete a blob, first get a blob reference and then call the `Remove-AzureStorageBlob` cmdlet on it. The following example deletes all the blobs in a given container. The example first sets variables for a storage account, and then creates a storage context. Next, the example retrieves a blob reference using the [Get-AzureStorageBlob](#) cmdlet and runs the [Remove-AzureStorageBlob](#) cmdlet to remove blobs from a container in Azure storage.

```
#Define the storage account and context.
$StorageAccountName = "yourstorageaccount"
$StorageAccountKey = "Storage key for yourstorageaccount ends with =="
$ContainerName = "containername"
$Ctx = New-AzureStorageContext -StorageAccountName $StorageAccountName -StorageAccountKey
$StorageAccountKey

#Get a reference to all the blobs in the container.
$blobs = Get-AzureStorageBlob -Container $ContainerName -Context $Ctx

#Delete blobs in a specified container.
$blobs | Remove-AzureStorageBlob
```

How to manage Azure blob snapshots

Azure lets you create a snapshot of a blob. A snapshot is a read-only version of a blob that's taken at a point in time. Once a snapshot has been created, it can be read, copied, or deleted, but not modified. Snapshots provide a way to back up a blob as it appears at a moment in time. For more information, see [Creating a Snapshot of a Blob](#).

How to create a blob snapshot

To create a snapshot of a blob, first get a blob reference and then call the [ICloudBlob.CreateSnapshot](#) method on it. The following example first sets variables for a storage account, and then creates a storage context. Next, the example retrieves a blob reference using the [Get-AzureStorageBlob](#) cmdlet and runs the [ICloudBlob.CreateSnapshot](#) method to create a snapshot.

```
#Define the storage account and context.
$StorageAccountName = "yourstorageaccount"
$StorageAccountKey = "Storage key for yourstorageaccount ends with =="
$ContainerName = "yourcontainername"
$BlobName = "yourblobname"
$Ctx = New-AzureStorageContext -StorageAccountName $StorageAccountName -StorageAccountKey
$StorageAccountKey

#Get a reference to a blob.
$blob = Get-AzureStorageBlob -Context $Ctx -Container $ContainerName -Blob $BlobName

#Create a snapshot of the blob.
$snap = $blob.ICloudBlob.CreateSnapshot()
```

How to list a blob's snapshots

You can create as many snapshots as you want for a blob. You can list the snapshots associated with your blob to track your current snapshots. The following example uses a predefined blob and calls the [Get-AzureStorageBlob](#) cmdlet to list the snapshots of that blob.

```
#Define the blob name.
$BlobName = "yourblobname"

#List the snapshots of a blob.
Get-AzureStorageBlob -Context $Ctx -Prefix $BlobName -Container $ContainerName | Where-Object {
    $_.ICloudBlob.IsSnapshot -and $_.Name -eq $BlobName }
```

How to copy a snapshot of a blob

You can copy a snapshot of a blob to restore the snapshot. For detailed information and restrictions, see [Creating a Snapshot of a Blob](#). The following example first sets variables for a storage account, and then creates a storage context. Next, the example defines the container and blob name variables. The example retrieves a blob reference using the `Get-AzureStorageBlob` cmdlet and runs the `ICloudBlob.CreateSnapshot` method to create a snapshot. Then, the example runs the `Start-AzureStorageBlobCopy` cmdlet to copy the snapshot of a blob using the `ICloudBlob` object for the source blob. Be sure to update the variables based on your configuration before running the example. Note that the following example assumes that the source and destination containers, and the source blob already exist.

```
#Define the storage account and context.
$StorageAccountName = "yourstorageaccount"
$StorageAccountKey = "Storage key for yourstorageaccount ends with =="
$Ctx = New-AzureStorageContext -StorageAccountName $StorageAccountName -StorageAccountKey
$StorageAccountKey

#Define the variables.
$SrcContainerName = "yoursourcecontainername"
$DestContainerName = "yourdestcontainername"
$SrcBlobName = "yourblobname"
$DestBlobName = "CopyBlobName"

#Get a reference to a blob.
$blob = Get-AzureStorageBlob -Context $Ctx -Container $SrcContainerName -Blob $SrcBlobName

#Create a snapshot of a blob.
$snap = $blob.ICloudBlob.CreateSnapshot()

#Copy the snapshot to another container.
Start-AzureStorageBlobCopy -Context $Ctx -ICloudBlob $snap -DestBlob $DestBlobName -DestContainer
$DestContainerName
```

Now that you have learned how to manage Azure blobs and blob snapshots with Azure PowerShell, go to the next section to learn how to manage tables, queues, and files.

How to manage Azure tables and table entities

Azure Table storage service is a NoSQL datastore, which you can use to store and query huge sets of structured, non-relational data. The main components of the service are tables, entities, and properties. A table is a collection of entities. An entity is a set of properties. Each entity can have up to 252 properties, which are all name-value pairs. This section assumes that you are already familiar with the Azure Table Storage Service concepts. For detailed information, see [Understanding the Table Service Data Model](#) and [Get started with Azure Table storage using .NET](#).

In the following subsections, you'll learn how to manage Azure Table storage service using Azure PowerShell. The scenarios covered include **creating, deleting, and retrieving tables**, as well as **adding, querying, and deleting table entities**.

How to create a table

Every table must reside in an Azure storage account. The following example demonstrates how to create a table in Azure Storage. The example first establishes a connection to Azure Storage using the storage account context, which includes the storage account name and its access key. Next, it uses the `New-AzureStorageTable` cmdlet to create a table in Azure Storage.

```
#Define the storage account and context.  
$StorageAccountName = "yourstorageaccount"  
$StorageAccountKey = "Storage key for yourstorageaccount ends with =="  
$Ctx = New-AzureStorageContext $StorageAccountName -StorageAccountKey $StorageAccountKey  
  
#Create a new table.  
$tabName = "yourtablename"  
New-AzureStorageTable -Name $tabName -Context $Ctx
```

How to retrieve a table

You can query and retrieve one or all tables in a Storage account. The following example demonstrates how to retrieve a given table using the [Get-AzureStorageTable](#) cmdlet.

```
#Retrieve a table.  
$tabName = "yourtablename"  
Get-AzureStorageTable -Name $tabName -Context $Ctx
```

If you call the [Get-AzureStorageTable](#) cmdlet without any parameters, it gets all storage tables for a Storage account.

How to delete a table

You can delete a table from a storage account by using the [Remove-AzureStorageTable](#) cmdlet.

```
#Delete a table.  
$tabName = "yourtablename"  
Remove-AzureStorageTable -Name $tabName -Context $Ctx
```

How to manage table entities

Currently, Azure PowerShell does not provide cmdlets to manage table entities directly. To perform operations on table entities, you can use the classes given in the [Azure Storage Client Library for .NET](#).

How to add table entities

To add an entity to a table, first create an object that defines your entity properties. An entity can have up to 255 properties, including 3 system properties: **PartitionKey**, **RowKey**, and **Timestamp**. You are responsible for inserting and updating the values of **PartitionKey** and **RowKey**. The server manages the value of **Timestamp**, which cannot be modified. Together the **PartitionKey** and **RowKey** uniquely identify every entity within a table.

- **PartitionKey**: Determines the partition that the entity is stored in.
- **RowKey**: Uniquely identifies the entity within the partition.

You may define up to 252 custom properties for an entity. For more information, see [Understanding the Table Service Data Model](#).

The following example demonstrates how to add entities to a table. The example shows how to retrieve the employee table and add several entities into it. First, it establishes a connection to Azure Storage using the storage account context, which includes the storage account name and its access key. Next, it retrieves the given table using the [Get-AzureStorageTable](#) cmdlet. If the table does not exist, the [New-AzureStorageTable](#) cmdlet is used to create a table in Azure Storage. Next, the example defines a custom function `Add-Entity` to add entities to the table by specifying each entity's partition and row key. The `Add-Entity` function calls the [New-Object](#) cmdlet on the [Microsoft.WindowsAzure.Storage.Table.DynamicTableEntity](#) class to create an entity object. Later, the example calls the [Microsoft.WindowsAzure.Storage.Table.TableOperation.Insert](#) method on this entity object to add it to a table.

```

#Function Add-Entity: Adds an employee entity to a table.
function Add-Entity() {
    [CmdletBinding()]
    param(
        $table,
        [String]$partitionKey,
        [String]$rowKey,
        [String]$name,
        [Int]$id
    )

    $entity = New-Object -TypeName Microsoft.WindowsAzure.Storage.Table.DynamicTableEntity -ArgumentList
    $partitionKey, $rowKey
    $entity.Properties.Add("Name", $name)
    $entity.Properties.Add("ID", $id)

    $result =
    $table.CloudTable.Execute([Microsoft.WindowsAzure.Storage.Table.TableOperation]::Insert($entity))
}

#Define the storage account and context.
$StorageAccountName = "yourstorageaccount"
$StorageAccountKey = Get-AzureStorageKey -StorageAccountName $StorageAccountName
$Ctx = New-AzureStorageContext $StorageAccountName -StorageAccountKey $StorageAccountKey.Primary
$TableName = "Employees"

#Retrieve the table if it already exists.
$table = Get-AzureStorageTable -Name $TableName -Context $Ctx -ErrorAction Ignore

#Create a new table if it does not exist.
if ($table -eq $null)
{
    $table = New-AzureStorageTable -Name $TableName -Context $Ctx
}

#Add multiple entities to a table.
Add-Entity -Table $table -PartitionKey Partition1 -RowKey Row1 -Name Chris -Id 1
Add-Entity -Table $table -PartitionKey Partition1 -RowKey Row2 -Name Jessie -Id 2
Add-Entity -Table $table -PartitionKey Partition2 -RowKey Row1 -Name Christine -Id 3
Add-Entity -Table $table -PartitionKey Partition2 -RowKey Row2 -Name Steven -Id 4

```

How to query table entities

To query a table, use the [Microsoft.WindowsAzure.Storage.Table.TableQuery](#) class. The following example assumes that you've already run the script given in the How to add entities section of this guide. The example first establishes a connection to Azure Storage using the storage context, which includes the storage account name and its access key. Next, it tries to retrieve the previously created "Employees" table using the [Get-AzureStorageTable](#) cmdlet. Calling the [New-Object](#) cmdlet on the [Microsoft.WindowsAzure.Storage.Table.TableQuery](#) class creates a new query object. The example looks for the entities that have an 'ID' column whose value is 1 as specified in a string filter. For detailed information, see [Querying Tables and Entities](#). When you run this query, it returns all entities that match the filter criteria.

```

#Define the storage account and context.
$StorageAccountName = "yourstorageaccount"
$StorageAccountKey = Get-AzureStorageKey -StorageAccountName $StorageAccountName
$Ctx = New-AzureStorageContext -StorageAccountName $StorageAccountName -StorageAccountKey
$StorageAccountKey.Primary
$TableName = "Employees"

#Get a reference to a table.
$table = Get-AzureStorageTable -Name $TableName -Context $Ctx

#create a table query.
$query = New-Object Microsoft.WindowsAzure.Storage.Table.TableQuery

#define columns to select.
$list = New-Object System.Collections.Generic.List[string]
$list.Add("RowKey")
$list.Add("ID")
$list.Add("Name")

#Set query details.
$query.FilterString = "ID gt 0"
$query.SelectColumns = $list
$query.TakeCount = 20

#Execute the query.
$entities = $table.CloudTable.ExecuteQuery($query)

#Display entity properties with the table format.
$entities | Format-Table PartitionKey, RowKey, @{ Label = "Name"; Expression= ${_.Properties["Name"].StringValue}}, @{ Label = "ID"; Expression=${_.Properties["ID"].Int32Value}} -AutoSize

```

How to delete table entities

You can delete an entity using its partition and row keys. The following example assumes that you've already run the script given in the How to add entities section of this guide. The example first establishes a connection to Azure Storage using the storage context, which includes the storage account name and its access key. Next, it tries to retrieve the previously created "Employees" table using the [Get-AzureStorageTable](#) cmdlet. If the table exists, the example calls the [Microsoft.WindowsAzure.Storage.Table.TableOperation.Retrieve](#) method to retrieve an entity based on its partition and row key values. Then, pass the entity to the [Microsoft.WindowsAzure.Storage.Table.TableOperation.Delete](#) method to delete.

```

#Define the storage account and context.
$StorageAccountName = "yourstorageaccount"
$StorageAccountKey = Get-AzureStorageKey -StorageAccountName $StorageAccountName
$Ctx = New-AzureStorageContext -StorageAccountName $StorageAccountName -StorageAccountKey
$StorageAccountKey.Primary

#Retrieve the table.
$TableName = "Employees"
$table = Get-AzureStorageTable -Name $TableName -Context $Ctx -ErrorAction Ignore

#if the table exists, start deleting its entities.
if ($table -ne $null)
{
    #Together the PartitionKey and RowKey uniquely identify every
    #entity within a table.
    $tableResult =
$table.CloudTable.Execute([Microsoft.WindowsAzure.Storage.Table.TableOperation]::Retrieve("Partition2",
"Row1"))
    $entity = $tableResult.Result
    if ($entity -ne $null)
    {
        #Delete the entity.
        $table.CloudTable.Execute([Microsoft.WindowsAzure.Storage.Table.TableOperation]::Delete($entity))
    }
}

```

How to manage Azure queues and queue messages

Azure Queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS. This section assumes that you are already familiar with the Azure Queue Storage Service concepts. For detailed information, see [Get started with Azure Queue storage using .NET](#).

This section will show you how to manage Azure Queue storage service using Azure PowerShell. The scenarios covered include **inserting** and **deleting** queue messages, as well as **creating, deleting**, and **retrieving queues**.

How to create a queue

The following example first establishes a connection to Azure Storage using the storage account context, which includes the storage account name and its access key. Next, it calls [New-AzureStorageQueue](#) cmdlet to create a queue named 'queuename'.

```

#Define the storage account and context.
$StorageAccountName = "yourstorageaccount"
$StorageAccountKey = Get-AzureStorageKey -StorageAccountName $StorageAccountName
$Ctx = New-AzureStorageContext -StorageAccountName $StorageAccountName -StorageAccountKey
$StorageAccountKey.Primary
$QueueName = "queuename"
$Queue = New-AzureStorageQueue -Name $QueueName -Context $Ctx

```

For information on naming conventions for Azure Queue Service, see [Naming Queues and Metadata](#).

How to retrieve a queue

You can query and retrieve a specific queue or a list of all the queues in a Storage account. The following example demonstrates how to retrieve a specified queue using the [Get-AzureStorageQueue](#) cmdlet.

```

#Retrieve a queue.
$QueueName = "queuename"
$Queue = Get-AzureStorageQueue -Name $QueueName -Context $Ctx

```

If you call the [Get-AzureStorageQueue](#) cmdlet without any parameters, it gets a list of all the queues.

How to delete a queue

To delete a queue and all the messages contained in it, call the [Remove-AzureStorageQueue](#) cmdlet. The following example shows how to delete a specified queue using the [Remove-AzureStorageQueue](#) cmdlet.

```
#Delete a queue.  
$QueueName = "yourqueuename"  
Remove-AzureStorageQueue -Name $QueueName -Context $Ctx
```

How to insert a message into a queue

To insert a message into an existing queue, first create a new instance of the [Microsoft.WindowsAzure.Storage.Queue.CloudQueueMessage](#) class. Next, call the [AddMessage](#) method. A CloudQueueMessage can be created from either a string (in UTF-8 format) or a byte array.

The following example demonstrates how to add message to a queue. The example first establishes a connection to Azure Storage using the storage account context, which includes the storage account name and its access key. Next, it retrieves the specified queue using the [Get-AzureStorageQueue](#) cmdlet. If the queue exists, the [New-Object](#) cmdlet is used to create an instance of the [Microsoft.WindowsAzure.Storage.Queue.CloudQueueMessage](#) class. Later, the example calls the [AddMessage](#) method on this message object to add it to a queue. Here is code which retrieves a queue and inserts the message 'MessageInfo':

```
#Define the storage account and context.  
$StorageAccountName = "yourstorageaccount"  
$StorageAccountKey = Get-AzureStorageKey -StorageAccountName $StorageAccountName  
$Ctx = New-AzureStorageContext -StorageAccountName $StorageAccountName -StorageAccountKey  
$StorageAccountKey.Primary  
  
#Retrieve the queue.  
$QueueName = "queuename"  
$Queue = Get-AzureStorageQueue -Name $QueueName -Context $ctx  
  
#If the queue exists, add a new message.  
if ($Queue -ne $null) {  
    # Create a new message using a constructor of the CloudQueueMessage class.  
    $QueueMessage = New-Object -TypeName Microsoft.WindowsAzure.Storage.Queue.CloudQueueMessage -  
    ArgumentList MessageInfo  
  
    # Add a new message to the queue.  
    $Queue.CloudQueue.AddMessage($QueueMessage)  
}
```

How to de-queue at the next message

Your code de-queues a message from a queue in two steps. When you call the [Microsoft.WindowsAzure.Storage.Queue.CloudQueue.GetMessage](#) method, you get the next message in a queue. A message returned from **GetMessage** becomes invisible to any other code reading messages from this queue. To finish removing the message from the queue, you must also call the [Microsoft.WindowsAzure.Storage.Queue.CloudQueue.DeleteMessage](#) method. This two-step process of removing a message assures that if your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls **DeleteMessage** right after the message has been processed.

```

# Define the storage account and context.
$StorageAccountName = "yourstorageaccount"
$StorageAccountKey = Get-AzureStorageKey -StorageAccountName $StorageAccountName
$Ctx = New-AzureStorageContext -StorageAccountName $StorageAccountName -StorageAccountKey
$StorageAccountKey.Primary

# Retrieve the queue.
$QueueName = "queuename"
$Queue = Get-AzureStorageQueue -Name $QueueName -Context $ctx

$InvisibleTimeout = [System.TimeSpan]::FromSeconds(10)

# Get the message object from the queue.
$QueueMessage = $Queue.CloudQueue.GetMessage($InvisibleTimeout)
# Delete the message.
$Queue.CloudQueue.DeleteMessage($QueueMessage)

```

How to manage Azure file shares and files

Azure File storage offers shared storage for applications using the standard SMB protocol. Microsoft Azure virtual machines and cloud services can share file data across application components via mounted shares, and on-premises applications can access file data in a share via the File storage API or Azure PowerShell.

For more information on Azure File storage, see [Get started with Azure File storage on Windows](#) and [File Service REST API](#).

How to set and query storage analytics

You can use [Azure Storage Analytics](#) to collect metrics for your Azure storage accounts and log data about requests sent to your storage account. You can use storage metrics to monitor the health of a storage account, and storage logging to diagnose and troubleshoot issues with your storage account. By default, storage metrics are not enabled for your storage services. You can enable monitoring using the Azure portal or Windows PowerShell, or programmatically using the storage client library. Storage logging happens server-side and enables you to record details for both successful and failed requests in your storage account. These logs enable you to see details of read, write, and delete operations against your tables, queues, and blobs as well as the reasons for failed requests.

To learn how to enable and view Storage Metrics data using PowerShell, see [How to enable Storage Metrics using PowerShell](#).

To learn how to enable and retrieve Storage Logging data using PowerShell, see [How to enable Storage Logging using PowerShell](#) and [Finding your Storage Logging log data](#). For detailed information on using Storage Metrics and Storage Logging to troubleshoot storage issues, see [Monitoring, Diagnosing, and Troubleshooting Microsoft Azure Storage](#).

How to manage Shared Access Signature (SAS) and Stored Access Policy

Shared access signatures are an important part of the security model for any application using Azure Storage. They are useful for providing limited permissions to your storage account to clients that should not have the account key. By default, only the owner of the storage account may access blobs, tables, and queues within that account. If your service or application needs to make these resources available to other clients without sharing your access key, you have three options:

- Set a container's permissions to permit anonymous read access to the container and its blobs. This is not allowed for tables or queues.

- Use a shared access signature that grants restricted access rights to containers, blobs, queues, and tables for a specific time interval.
- Use a stored access policy to obtain an additional level of control over shared access signatures for a container or its blobs, for a queue, or for a table. The stored access policy allows you to change the start time, expiry time, or permissions for a signature, or to revoke it after it has been issued.

A shared access signature can be in one of two forms:

- **Ad hoc SAS:** When you create an ad hoc SAS, the start time, expiry time, and permissions for the SAS are all specified on the SAS URI. This type of SAS may be created on a container, blob, table, or queue and it is non-revocable.
- **SAS with stored access policy:** A stored access policy is defined on a resource container a blob container, table, or queue - and you can use it to manage constraints for one or more shared access signatures. When you associate a SAS with a stored access policy, the SAS inherits the constraints - the start time, expiry time, and permissions - defined for the stored access policy. This type of SAS is revocable.

For more information, see [Using Shared Access Signatures \(SAS\)](#) and [Manage anonymous read access to containers and blobs](#).

In the next sections, you will learn how to create a shared access signature token and stored access policy for Azure tables. Azure PowerShell provides similar cmdlets for containers, blobs, and queues as well. To run the scripts in this section, download the [Azure PowerShell version 0.8.14](#) or later.

How to create a policy-based Shared Access Signature token

Use the `New-AzureStorageTableStoredAccessPolicy` cmdlet to create a new stored access policy. Then, call the `New-AzureStorageTableSASToken` cmdlet to create a new policy-based shared access signature token for an Azure Storage table.

```
$policy = "policy1"
New-AzureStorageTableStoredAccessPolicy -Name $tableName -Policy $policy -Permission "rd" -StartTime
"2015-01-01" -ExpiryTime "2016-01-01" -Context $Ctx
New-AzureStorageTableSASToken -Name $tableName -Policy $policy -Context $Ctx
```

How to create an ad hoc (non-revocable) Shared Access Signature token

Use the `New-AzureStorageTableSASToken` cmdlet to create a new ad hoc (non-revocable) Shared Access Signature token for an Azure Storage table:

```
New-AzureStorageTableSASToken -Name $tableName -Permission "rqud" -StartTime "2015-01-01" -ExpiryTime
"2015-02-01" -Context $Ctx
```

How to create a stored access policy

Use the `New-AzureStorageTableStoredAccessPolicy` cmdlet to create a new stored access policy for an Azure Storage table:

```
$policy = "policy1"
New-AzureStorageTableStoredAccessPolicy -Name $tableName -Policy $policy -Permission "rd" -StartTime
"2015-01-01" -ExpiryTime "2016-01-01" -Context $Ctx
```

How to update a stored access policy

Use the `Set-AzureStorageTableStoredAccessPolicy` cmdlet to update an existing stored access policy for an Azure Storage table:

```
Set-AzureStorageTableStoredAccessPolicy -Policy $policy -Table $tableName -Permission "rd" -NoExpiryTime  
-NoStartTime -Context $Ctx
```

How to delete a stored access policy

Use the Remove-AzureStorageTableStoredAccessPolicy cmdlet to delete a stored access policy on an Azure Storage table:

```
Remove-AzureStorageTableStoredAccessPolicy -Policy $policy -Table $tableName -Context $Ctx
```

How to use Azure Storage for U.S. government and Azure China

An Azure environment is an independent deployment of Microsoft Azure, such as [Azure Government for U.S. government](#), [AzureCloud for global Azure](#), and [AzureChinaCloud for Azure operated by 21Vianet in China](#). You can deploy new Azure environments for U.S government and Azure China.

To use Azure Storage with AzureChinaCloud, you need to create a storage context that is associated with AzureChinaCloud. Follow these steps to get you started:

1. Run the [Get-AzureEnvironment](#) cmdlet to see the available Azure environments:

```
Get-AzureEnvironment
```

2. Add an Azure China account to Windows PowerShell:

```
Add-AzureAccount -Environment AzureChinaCloud
```

3. Create a storage context for an AzureChinaCloud account:

```
$Ctx = New-AzureStorageContext -StorageAccountName $AccountName -StorageAccountKey $AccountKey > -  
Environment AzureChinaCloud
```

To use Azure Storage with [U.S. Azure Government](#), you should define a new environment and then create a new storage context with this environment:

1. Run the [Get-AzureEnvironment](#) cmdlet to see the available Azure environments:

```
Get-AzureEnvironment
```

2. Add an Azure US Government account to Windows PowerShell:

```
Add-AzureAccount -Environment AzureUSGovernment
```

3. Create a storage context for an AzureUSGovernment account:

```
$Ctx = New-AzureStorageContext -StorageAccountName $AccountName -StorageAccountKey $AccountKey > -  
Environment AzureUSGovernment
```

For more information, see:

- [Microsoft Azure Government Developer Guide](#).
- [Overview of Differences When Creating an Application on China Service](#)

Next Steps

In this guide, you've learned how to manage Azure Storage with Azure PowerShell. Here are some related articles and resources for learning more about them.

- [Azure Storage Documentation](#)
- [Azure Storage PowerShell Cmdlets](#)
- [Windows PowerShell Reference](#)

Using the Azure CLI 2.0 (Preview) with Azure Storage

1/17/2017 • 11 min to read • [Edit on GitHub](#)

Overview

The open-source, cross-platform Azure CLI 2.0 (Preview) provides a set of commands for working with the Azure platform. It provides much of the same functionality found in the [Azure portal](#), including rich data access.

In this guide, we show you how to use the [Azure CLI 2.0 \(Preview\)](#) to perform several tasks working with resources in your Azure Storage account. We recommend that you download and install or upgrade to the latest version of the CLI 2.0 before using this guide.

The examples in the guide assume the use of the Bash shell on Ubuntu, but other platforms should perform similarly.

Versions of the Azure CLI

There are two versions of the Azure Command-line Interface (CLI) currently available:

[Azure CLI 1.0](#)--our CLI written in Node.js, for use with both the classic and resource management deployment models.

[Azure CLI 2.0 \(Preview\)](#)--a next-generation CLI written in Python, for use with the resource management deployment model.

Prerequisites

This guide assumes that you understand the basic concepts of Azure Storage. It also assumes that you're able to satisfy the account creation requirements that are specified below for Azure and the Storage service.

Accounts

- **Azure account:** If you don't already have an Azure subscription, [create a free Azure account](#).
- **Storage account:** See [Create a storage account](#) in [About Azure storage accounts](#).

Install the Azure CLI 2.0 (Preview)

Download and install the Azure CLI 2.0 (Preview) by following the instructions outlined in [Install Azure CLI 2.0 \(Preview\)](#).

TIP

If you have trouble with the installation, check out the [Installation Troubleshooting](#) section of the article, and the [Install Troubleshooting](#) guide on GitHub.

Working with the CLI

Once you've installed the CLI, you can use the `az` command in your command-line interface (Bash, Terminal, Command Prompt) to access the Azure CLI commands. Type the `az` command, and you should

be presented with output similar to:



Welcome to the cool new Azure CLI!

Here are the base commands:

```
account  : Commands to manage subscriptions.
acr       : Commands to manage Azure container registries.
acs       : Commands to manage Azure container services.
ad        : Synchronize on-premises directories and manage Azure Active Directory (AAD)
            resources.
appservice: Commands to manage your Azure web apps and App Service plans.
cloud     : Manage the Azure clouds registered.
component : Commands to manage and update Azure CLI 2.0 (Preview) components.
configure : Configure Azure CLI 2.0 Preview or view your configuration. The command is
            interactive, so just type `az configure` and respond to the prompts.
container : Set up automated builds and deployments for multi-container Docker applications.
context   : Manage contexts.
feature   : Commands to manage resource provider features, such as previews.
feedback  : Loving or hating the CLI? Let us know!
group    : Commands to manage resource groups.
login    : Log in to access Azure subscriptions.
logout   : Log out to remove accesses to Azure subscriptions.
network  : Manages Network resources.
policy   : Commands to manage resource policies.
provider : Manage resource providers.
resource : Generic commands to manage Azure resources.
role     : Use role assignments to manage access to your Azure resources.
storage  : Durable, highly available, and massively scalable cloud storage.
tag      : Manage resource tags.
vm       : Provision Linux and Windows virtual machines in minutes.
vmss    : Create highly available, auto-scalable Linux or Windows virtual machines.
```

In your command-line interface, execute the command `az storage -h` to list the `storage` group commands and its subgroups. The descriptions of the subgroups provide an overview of the functionality the Azure CLI provides for working with your storage resources.

```
Group
az storage: Durable, highly available, and massively scalable cloud storage.

Subgroups:
account  : Manage storage accounts.
blob      : Object storage for unstructured data.
container: Manage blob storage containers.
cors     : Manage Storage service Cross-Origin Resource Sharing (CORS).
directory: Manage file storage directories.
entity   : Manage table storage entities.
file     : File shares that use the standard SMB 3.0 protocol.
logging  : Manage Storage service logging information.
message  : Manage queue storage messages.
metrics  : Manage Storage service metrics.
queue    : Effectively scale apps according to traffic using queues.
share    : Manage file shares.
table   : NoSQL key-value storage using semi-structured datasets.
```

Connect the CLI to your Azure subscription

To work with the resources in your Azure subscription, you must first log in to your Azure account with `az login`. There are several ways you can log in:

- **Interactive login:** `az login`
- **Log in with user name and password:** `az login -u johndoe@contoso.com -p VerySecret`
 - This doesn't work with Microsoft accounts or accounts that use multi-factor authentication.
- **Log in with a service principal:**

```
az login --service-principal -u http://azure-cli-2016-08-05-14-31-15 -p VerySecret --tenant  
contoso.onmicrosoft.com
```

Azure CLI 2.0 sample script

Next, we'll work with a small shell script that issues a few basic Azure CLI 2.0 commands to interact with Azure Storage resources. The script first creates a new container in your storage account, then uploads an existing file (as a blob) to that container. It then lists all blobs in the container, and finally, downloads the file to a destination on your local computer that you specify.

```
#!/bin/bash  
# A simple Azure Storage example script  
  
export AZURE_STORAGE_ACCOUNT=<storage_account_name>  
export AZURE_STORAGE_ACCESS_KEY=<storage_account_key>  
  
export container_name=<container_name>  
export blob_name=<blob_name>  
export file_to_upload=<file_to_upload>  
export destination_file=<destination_file>  
  
echo "Creating the container..."  
az storage container create -n $container_name  
  
echo "Uploading the file..."  
az storage blob upload -f $file_to_upload -c $container_name -n $blob_name  
  
echo "Listing the blobs..."  
az storage blob list -c $container_name  
  
echo "Downloading the file..."  
az storage blob download -c $container_name -n $blob_name -f $destination_file  
  
echo "Done"
```

Configure and run the script

1. Open your favorite text editor, then copy and paste the preceding script into the editor.
2. Next, update the script's variables to reflect your configuration settings. Replace the following values as specified:
 - **<storage_account_name>** The name of your storage account.
 - **<storage_account_key>** The primary or secondary access key for your storage account.
 - **<container_name>** A name the new container to create, such as "azure-cli-sample-container".
 - **<blob_name>** A name for the destination blob in the container.
 - **<file_to_upload>** The path to small file on your local computer, such as "~/images/Helloworld.png".
 - **<destination_file>** The destination file path, such as "~/downloadedImage.png".
3. After you've updated the necessary variables, save the script and exit your editor. The next steps assume you've named your script **my_storage_sample.sh**.

4. Mark the script as executable, if necessary: `chmod +x my_storage_sample.sh`

5. Execute the script. For example, in Bash: `./my_storage_sample.sh`

You should see output similar to the following, and the **<destination_file>** you specified in the script should appear on your local computer.

```
Creating the container...
Success
-----
True
Uploading the file...                               Percent complete: %100.0
Listing the blobs...
Name          Blob Type    Length  Content Type
-----        -----      -----   -----
test_blob.txt  BlockBlob     771    application/octet-stream 2016-12-21T15:35:30+00:00
Downloading the file...
Name
-----
test_blob.txt
Done
```

NOTE

The preceding output is in **table** format. You can specify which output format to use by specifying the `--output` argument in your CLI commands, or set it globally using `az configure`.

Manage storage accounts

Create a new storage account

To use Azure Storage, you need a storage account. You can create a new Azure Storage account after you've configured your computer to [connect to your subscription](#).

```
az storage account create -l <location> -n <account_name> -g <resource_group> --sku <account_sku>
```

- `-l` [Required]: Location. For example, "West US".
- `-n` [Required]: The storage account name. The name must be 3 to 24 characters in length, and use only lowercase alphanumeric characters.
- `-g` [Required]: Name of resource group.
- `--sku` [Required]: The storage account SKU. Allowed values:
 - `Premium_LRS`
 - `Standard_GRS`
 - `Standard_LRS`
 - `Standard_RAGRS`
 - `Standard_ZRS`

Set default Azure storage account environment variables

You can have multiple storage accounts in your Azure subscription. To select one of them to use for all subsequent storage commands, you can set these environment variables:

```
export AZURE_STORAGE_ACCOUNT=<account_name>
export AZURE_STORAGE_ACCESS_KEY=<key>
```

Another way to set a default storage account is by using a connection string. First, get the connection string

with the `show-connection-string` command:

```
az storage account show-connection-string -n <account_name> -g <resource_group>
```

Then copy the output connection string and set the `AZURE_STORAGE_CONNECTION_STRING` environment variable (you might need to enclose the connection string in quotes):

```
export AZURE_STORAGE_CONNECTION_STRING=<connection_string>
```

NOTE

All examples in the following sections of this article assume that you've set the `AZURE_STORAGE_ACCOUNT` and `AZURE_STORAGE_ACCESS_KEY` environment variables.

Create and manage blobs

Azure Blob storage is a service for storing large amounts of unstructured data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. This section assumes that you are already familiar with Azure Blob storage concepts. For detailed information, see [Get started with Azure Blob storage using .NET](#) and [Blob Service Concepts](#).

Create a container

Every blob in Azure storage must be in a container. You can create a container by using the

```
az storage container create
```

 command:

```
az storage container create -n <container_name>
```

You can set one of three levels of read access for a new container by specifying the optional `--public-access` argument:

- `off` (default): Container data is private to the account owner.
- `blob`: Public read access for blobs.
- `container`: Public read and list access to the entire container.

For more information, see [Manage anonymous read access to containers and blobs](#).

Upload a blob to a container

Azure Blob storage supports block, append, and page blobs. Upload blobs to a container by using the

```
blob upload
```

 command:

```
az storage blob upload -f <local_file_path> -c <container_name> -n <blob_name>
```

By default, the `blob upload` command uploads *.vhd files to page blobs, or block blobs otherwise. To specify another type when you upload a blob, you can use the `--type` argument--allowed values are `append`, `block`, and `page`.

For more information on the different blob types, see [Understanding Block Blobs, Append Blobs, and Page Blobs](#).

Download blobs from a container

This example demonstrates how to download a blob from a container:

```
az storage blob download -c mycontainer -n myblob.png -f ~/mydownloadedblob.png
```

Copy blobs

You can copy blobs within or across storage accounts and regions asynchronously.

The following example demonstrates how to copy blobs from one storage account to another. We first create a container in another account, specifying that its blobs are publicly, anonymously accessible. Next, we upload a file to the container, and finally, copy the blob from that container into the **mycontainer** container in the current account.

```
az storage container create -n mycontainer2 --account-name <accountName2> --account-key <accountKey2> --public-access blob

az storage blob upload -f ~/Images/Helloworld.png -c mycontainer2 -n myBlockBlob2 --account-name <accountName2> --account-key <accountKey2>

az storage blob copy start -u https://<accountname2>.blob.core.windows.net/mycontainer2/myBlockBlob2 -b myBlobBlob -c mycontainer
```

The source blob URL (specified by `-u`) must either be publicly accessible, or include a shared access signature (SAS) token.

Delete a blob

To delete a blob, use the `blob delete` command:

```
az storage blob delete -c <container_name> -n <blob_name>
```

Create and manage file shares

Azure File storage offers shared storage for applications using the Server Message Block (SMB) protocol. Microsoft Azure virtual machines and cloud services, as well as on-premises applications, can share file data via mounted shares. You can manage file shares and file data via the Azure CLI. For more information on Azure File storage, see [Get started with Azure File storage on Windows](#) or [How to use Azure File storage with Linux](#).

Create a file share

An Azure File share is an SMB file share in Azure. All directories and files must be created in a file share. An account can contain an unlimited number of shares, and a share can store an unlimited number of files, up to the capacity limits of the storage account. The following example creates a file share named **myshare**.

```
az storage share create -n myshare
```

Create a directory

A directory provides an optional hierarchical structure for an Azure file share. The following example creates a directory named **myDir** in the file share.

```
az storage directory create -n myDir -s myshare
```

Note that directory path can include multiple levels, *e.g.*, **a/b**. However, you must ensure that all parent directories exist. For example, for path **a/b**, you must create directory **a** first, then create directory **b**.

Upload a local file to a share

The following example uploads a file from `~/temp/samplefile.txt` to root of the **myshare** file share. The

`--source` argument specifies the existing local file to upload.

```
az storage file upload --share-name myshare --source ~/temp/samplefile.txt
```

As with directory creation, you can specify a directory path within the share to upload the file to an existing directory within the share:

```
az storage file upload --share-name myshare/myDir --source ~/temp/samplefile.txt
```

A file in the share can be up to 1 TB in size.

List the files in a share

You can list files and directories in a share by using the `az storage file list` command:

```
# List the files in the root of a share
az storage file list -s myshare

# List the files in a directory within a share
az storage file list -s myshare/myDir

# List the files in a path within a share
az storage file list -s myshare -p myDir/mySubDir/MySubDir2
```

Copy files

You can copy a file to another file, a file to a blob, or a blob to a file. For example, to copy a file to a directory in a different share:

```
# Get the URL for the source file you want to copy
az storage file url -s myshare -p /mydir/image.png

# Copy the file to another share
az storage file copy start \
    --source-uri https://mystorageaccount.file.core.windows.net/myshare/mydir/image.png \
    --destination-share myshare2 --destination-path mydir2/image.png
```

NOTE

There is known issue in the CLI 2.0 (Preview) preventing the use of `--source-share` and `--source-path`. You can use the `--source-uri` argument as a workaround until this issue is resolved.

Next steps

Here are some additional resources for learning more about working with the Azure CLI 2.0 (Preview).

- [Get started with Azure CLI 2.0 \(Preview\)](#)
- [Azure CLI 2.0 \(Preview\) command reference](#)
- [Azure CLI 2.0 \(Preview\) on GitHub](#)

Using the Azure CLI 1.0 with Azure Storage

1/17/2017 • 9 min to read • [Edit on GitHub](#)

Overview

The Azure CLI provides a set of open source, cross-platform commands for working with the Azure Platform. It provides much of the same functionality found in the [Azure portal](#) as well as rich data access functionality.

In this guide, we'll explore how to use [Azure Command-Line Interface \(Azure CLI\)](#) to perform a variety of development and administration tasks with Azure Storage. We recommend that you download and install or upgrade to the latest Azure CLI before using this guide.

This guide assumes that you understand the basic concepts of Azure Storage. The guide provides a number of scripts to demonstrate the usage of the Azure CLI with Azure Storage. Be sure to update the script variables based on your configuration before running each script.

NOTE

The guide provides the Azure CLI command and script examples for classic storage accounts. See [Using the Azure CLI for Mac, Linux, and Windows with Azure Resource Management](#) for Azure CLI commands for Resource Manager storage accounts.

Versions of the Azure CLI

There are two versions of the Azure Command-line Interface (CLI) currently available:

[Azure CLI 1.0](#)--our CLI written in Node.js, for use with both the classic and resource management deployment models.

[Azure CLI 2.0 \(Preview\)](#)--a next-generation CLI written in Python, for use with the resource management deployment model.

Get started with Azure Storage and the Azure CLI in 5 minutes

This guide uses Ubuntu for examples, but other OS platforms should perform similarly.

New to Azure: Get a Microsoft Azure subscription and a Microsoft account associated with that subscription. For information on Azure purchase options, see [Free Trial](#), [Purchase Options](#), and [Member Offers](#) (for members of MSDN, Microsoft Partner Network, and BizSpark, and other Microsoft programs).

See [Assigning administrator roles in Azure Active Directory \(Azure AD\)](#) for more information about Azure subscriptions.

After creating a Microsoft Azure subscription and account:

1. Download and install the Azure CLI following the instructions outlined in [Install the Azure CLI](#).
2. Once the Azure CLI has been installed, you will be able to use the `azure` command from your command-line interface (Bash, Terminal, Command prompt) to access the Azure CLI commands. Type the `azure` command and you should see the following output.

```
info: _____
info: | \ / | \ / | \ / | \ / | \ / |
info: | ( ) | ( ) | ( ) | ( ) | ( ) |
info: Microsoft Azure: Microsoft's Cloud Platform
info: Tool version 0.8.17
help: Display help for a given command
help: help [options] [command]
help: Log in to an Azure subscription using Active Directory. Currently, the
user can login only via Microsoft organizational account
help: login [options]
help: Log out from Azure subscription using Active Directory. Currently, the
user can log out only via Microsoft organizational account
help: logout [options] [username]
help: Open the portal in a browser
help: portal [options]
help: Commands:
help: account      Commands to manage your account information and publish
h settings
help: config        Commands to manage your local settings
help: hdinsight     Commands to manage your HDInsight accounts
help: mobile         Commands to manage your Mobile Services
help: network        Commands to manage your Networks
help: sb             Commands to manage your Service Bus configuration
help: service        Commands to manage your Cloud Services
help: site           Commands to manage your Web Sites
help: sql            Commands to manage your SQL Server accounts
help: storage         Commands to manage your Storage objects
help: vm             Commands to manage your Virtual Machines
help: Options:
help: -h, --help      output usage information
help: -v, --version   output the application version
```

3. In the command-line interface, type `azure storage` to list out all the azure storage commands and get a first impression of the functionalities the Azure CLI provides. You can type command name with `-h` parameter (for example, `azure storage share create -h`) to see details of command syntax.
 4. Now, we'll give you a simple script that shows basic Azure CLI commands to access Azure Storage. The script will first ask you to set two variables for your storage account and key. Then, the script will create a new container in this new storage account and upload an existing image file (blob) to that container. After the script lists all blobs in that container, it will download the image file to the destination directory which exists on the local computer.

```

#!/bin/bash
# A simple Azure storage example

export AZURE_STORAGE_ACCOUNT=<storage_account_name>
export AZURE_STORAGE_ACCESS_KEY=<storage_account_key>

export container_name=<container_name>
export blob_name=<blob_name>
export image_to_upload=<image_to_upload>
export destination_folder=<destination_folder>

echo "Creating the container..."
azure storage container create $container_name

echo "Uploading the image..."
azure storage blob upload $image_to_upload $container_name $blob_name

echo "Listing the blobs..."
azure storage blob list $container_name

echo "Downloading the image..."
azure storage blob download $container_name $blob_name $destination_folder

echo "Done"

```

5. In your local computer, open your preferred text editor (vim for example). Type the above script into your text editor.
6. Now, you need to update the script variables based on your configuration settings.

- Use the given name in the script or enter a new name for your storage account. **Important:** The name of the storage account must be unique in Azure. It must be lowercase, too!
- The access key of your storage account.
- Use the given name in the script or enter a new name for your container.
- Enter a path to a picture on your local computer, such as: "~/images/HelloWorld.png".
- Enter a path to a local directory to store files downloaded from Azure Storage, such as: "~/downloadImages".

7. After you've updated the necessary variables in vim, press key combinations `ESC`, `:`, `wq!` to save the script.
8. To run this script, simply type the script file name in the bash console. After this script runs, you should have a local destination folder that includes the downloaded image file. The following screenshot shows an example output:

After the script runs, you should have a local destination folder that includes the downloaded image file.

Manage storage accounts with the Azure CLI

Connect to your Azure subscription

While most of the storage commands will work without an Azure subscription, we recommend you to connect to your subscription from the Azure CLI. To configure the Azure CLI to work with your subscription, follow the steps in [Connect to an Azure subscription from the Azure CLI](#).

Create a new storage account

To use Azure storage, you will need a storage account. You can create a new Azure storage account after you have configured your computer to connect to your subscription.

```
azure storage account create <account_name>
```

The name of your storage account must be between 3 and 24 characters in length and use numbers and lower-

case letters only.

Set a default Azure storage account in environment variables

You can have multiple storage accounts in your subscription. You can choose one of them and set it in the environment variables for all the storage commands in the same session. This enables you to run the Azure CLI storage commands without specifying the storage account and key explicitly.

```
export AZURE_STORAGE_ACCOUNT=<account_name>
export AZURE_STORAGE_ACCESS_KEY=<key>
```

Another way to set a default storage account is using connection string. Firstly get the connection string by command:

```
azure storage account connectionstring show <account_name>
```

Then copy the output connection string and set it to environment variable:

```
export AZURE_STORAGE_CONNECTION_STRING=<connection_string>
```

Create and manage blobs

Azure Blob storage is a service for storing large amounts of unstructured data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. This section assumes that you are already familiar with the Azure Blob storage concepts. For detailed information, see [Get started with Azure Blob storage using .NET](#) and [Blob Service Concepts](#).

Create a container

Every blob in Azure storage must be in a container. You can create a private container using the

```
azure storage container create
```

 command:

```
azure storage container create mycontainer
```

NOTE

There are three levels of anonymous read access: **Off**, **Blob**, and **Container**. To prevent anonymous access to blobs, set the Permission parameter to **Off**. By default, the new container is private and can be accessed only by the account owner. To allow anonymous public read access to blob resources, but not to container metadata or to the list of blobs in the container, set the Permission parameter to **Blob**. To allow full public read access to blob resources, container metadata, and the list of blobs in the container, set the Permission parameter to **Container**. For more information, see [Manage anonymous read access to containers and blobs](#).

Upload a blob into a container

Azure Blob Storage supports block blobs and page blobs. For more information, see [Understanding Block Blobs, Append Blobs, and Page Blobs](#).

To upload blobs in to a container, you can use the `azure storage blob upload`. By default, this command uploads the local files to a block blob. To specify the type for the blob, you can use the `--blobtype` parameter.

```
azure storage blob upload '~/images/Helloworld.png' mycontainer myBlockBlob
```

Download blobs from a container

The following example demonstrates how to download blobs from a container.

```
azure storage blob download mycontainer myBlockBlob ~/downloadImages/downloaded.png'
```

Copy blobs

You can copy blobs within or across storage accounts and regions asynchronously.

The following example demonstrates how to copy blobs from one storage account to another. In this sample we create a container where blobs are publicly, anonymously accessible.

```
azure storage container create mycontainer2 -a <accountName2> -k <accountKey2> -p Blob  
  
azure storage blob upload '~/Images/HelloWorld.png' mycontainer2 myBlockBlob2 -a <accountName2> -k <accountKey2>  
  
azure storage blob copy start 'https://<accountname2>.blob.core.windows.net/mycontainer2/myBlockBlob2'  
mycontainer
```

This sample performs an asynchronous copy. You can monitor the status of each copy operation by running the `azure storage blob copy show` operation.

Note that the source URL provided for the copy operation must either be publicly accessible, or include a SAS (shared access signature) token.

Delete a blob

To delete a blob, use the below command:

```
azure storage blob delete mycontainer myBlockBlob2
```

Create and manage file shares

Azure File storage offers shared storage for applications using the standard SMB protocol. Microsoft Azure virtual machines and cloud services, as well as on-premises applications, can share file data via mounted shares. You can manage file shares and file data via the Azure CLI. For more information on Azure File storage, see [Get started with Azure File storage on Windows](#) or [How to use Azure File storage with Linux](#).

Create a file share

An Azure File share is an SMB file share in Azure. All directories and files must be created in a file share. An account can contain an unlimited number of shares, and a share can store an unlimited number of files, up to the capacity limits of the storage account. The following example creates a file share named **myshare**.

```
azure storage share create myshare
```

Create a directory

A directory provides an optional hierarchical structure for an Azure file share. The following example creates a directory named **myDir** in the file share.

```
azure storage directory create myshare myDir
```

Note that directory path can include multiple levels, e.g., **a/b**. However, you must ensure that all parent directories exist. For example, for path **a/b**, you must create directory **a** first, then create directory **b**.

Upload a local file to directory

The following example uploads a file from **~/temp/samplefile.txt** to the **myDir** directory. Edit the file path so that it points to a valid file on your local machine:

```
azure storage file upload '~/temp/samplefile.txt' myshare myDir
```

Note that a file in the share can be up to 1 TB in size.

List the files in the share root or directory

You can list the files and subdirectories in a share root or a directory using the following command:

```
azure storage file list myshare myDir
```

Note that the directory name is optional for the listing operation. If omitted, the command lists the contents of the root directory of the share.

Copy files

Beginning with version 0.9.8 of Azure CLI, you can copy a file to another file, a file to a blob, or a blob to a file.

Below we demonstrate how to perform these copy operations using CLI commands. To copy a file to the new directory:

```
azure storage file copy start --source-share srcshare --source-path srccdir/hello.txt --dest-share destshare  
--dest-path destdir/hellocopy.txt --connection-string $srcConnectionString --dest-connection-string  
$destConnectionString
```

To copy a blob to a file directory:

```
azure storage file copy start --source-container srcctn --source-blob hello2.txt --dest-share hello  
--dest-path helldir/hello2copy.txt --connection-string $srcConnectionString --dest-connection-string  
$destConnectionString
```

Next Steps

You can find Azure CLI 1.0 command reference for working with Storage resources here:

- [Azure CLI commands in Resource Manager mode](#)
- [Azure CLI commands in Azure Service Management mode](#)

You may also like to try the [Azure CLI 2.0 \(Preview\)](#), our next-generation CLI written in Python, for use with the resource management deployment model.

Managing Azure Storage using Azure Automation

1/17/2017 • 1 min to read • [Edit on GitHub](#)

This guide will introduce you to the Azure Automation service, and how it can be used to simplify management of your Azure Storage blobs, tables, and queues.

What is Azure Automation?

Azure Automation is an Azure service for simplifying cloud management through process automation. Using Azure Automation, long-running, manual, error-prone, and frequently repeated tasks can be automated to increase reliability and efficiency, and reduce time to value for your organization.

Azure Automation provides a highly-reliable and highly-available workflow execution engine that scales to meet your needs as your organization grows. In Azure Automation, processes can be kicked off manually, by 3rd-party systems, or at scheduled intervals so that tasks happen exactly when needed.

Lower operational overhead and free up IT / DevOps staff to focus on work that adds business value by moving your cloud management tasks to be run automatically by Azure Automation.

How can Azure Automation help manage Azure Storage?

Azure Storage can be managed in Azure Automation by using the PowerShell cmdlets that are available in [Azure PowerShell](#). Azure Automation has these Storage PowerShell cmdlets available out of the box, so that you can perform all of your blob, table, and queue management tasks within the service. You can also pair these cmdlets in Azure Automation with the cmdlets for other Azure services, to automate complex tasks across Azure services and 3rd party systems.

The [Azure Automation runbook gallery](#) contains a variety of product team and community runbooks to get started automating management of Azure Storage, other Azure services, and 3rd party systems. Gallery runbooks include:

- [Remove Azure Storage Blobs that are Certain Days Old Using Automation Service](#)
- [Download a Blob from Azure Storage](#)
- [Backup all disks for a single Azure VM or for all VMs in a Cloud Service](#)

Next Steps

Now that you've learned the basics of Azure Automation and how it can be used to manage Azure Storage blobs, tables, and queues, follow these links to learn more about Azure Automation.

See the Azure Automation tutorial [Creating or importing a runbook in Azure Automation](#).

Azure Storage security guide

1/17/2017 • 43 min to read • [Edit on GitHub](#)

Overview

Azure Storage provides a comprehensive set of security capabilities which together enable developers to build secure applications. The storage account itself can be secured using Role-Based Access Control and Azure Active Directory. Data can be secured in transit between an application and Azure by using [Client-Side Encryption](#), HTTPS, or SMB 3.0. Data can be set to be automatically encrypted when written to Azure Storage using [Storage Service Encryption \(SSE\)](#). OS and Data disks used by virtual machines can be set to be encrypted using [Azure Disk Encryption](#). Delegated access to the data objects in Azure Storage can be granted using [Shared Access Signatures](#).

This article will provide an overview of each of these security features that can be used with Azure Storage. Links are provided to articles that will give details of each feature so you can easily do further investigation on each topic.

Here are the topics to be covered in this article:

- [Management Plane Security](#) – Securing your Storage Account

The management plane consists of the resources used to manage your storage account. In this section, we'll talk about the Azure Resource Manager deployment model and how to use Role-Based Access Control (RBAC) to control access to your storage accounts. We will also talk about managing your storage account keys and how to regenerate them.

- [Data Plane Security](#) – Securing Access to Your Data

In this section, we'll look at allowing access to the actual data objects in your Storage account, such as blobs, files, queues, and tables, using Shared Access Signatures and Stored Access Policies. We will cover both service-level SAS and account-level SAS. We'll also see how to limit access to a specific IP address (or range of IP addresses), how to limit the protocol used to HTTPS, and how to revoke a Shared Access Signature without waiting for it to expire.

- [Encryption in Transit](#)

This section discusses how to secure data when you transfer it into or out of Azure Storage. We'll talk about the recommended use of HTTPS and the encryption used by SMB 3.0 for Azure File Shares. We will also take a look at Client-side Encryption, which enables you to encrypt the data before it is transferred into Storage in a client application, and to decrypt the data after it is transferred out of Storage.

- [Encryption at Rest](#)

We will talk about Storage Service Encryption (SSE), and how you can enable it for a storage account, resulting in your block blobs, page blobs, and append blobs being automatically encrypted when written to Azure Storage. We will also look at how you can use Azure Disk Encryption and explore the basic differences and cases of Disk Encryption versus SSE versus Client-Side Encryption. We will briefly look at FIPS compliance for U.S. Government computers.

- Using [Storage Analytics](#) to audit access of Azure Storage

This section discusses how to find information in the storage analytics logs for a request. We'll take a look at real storage analytics log data and see how to discern whether a request is made with the Storage account key, with a Shared Access signature, or anonymously, and whether it succeeded or failed.

- [Enabling Browser-Based Clients using CORS](#)

This section talks about how to allow cross-origin resource sharing (CORS). We'll talk about cross-domain access, and how to handle it with the CORS capabilities built into Azure Storage.

Management Plane Security

The management plane consists of operations that affect the storage account itself. For example, you can create or delete a storage account, get a list of storage accounts in a subscription, retrieve the storage account keys, or regenerate the storage account keys.

When you create a new storage account, you select a deployment model of Classic or Resource Manager. The Classic model of creating resources in Azure only allows all-or-nothing access to the subscription, and in turn, the storage account.

This guide focuses on the Resource Manager model which is the recommended means for creating storage accounts. With the Resource Manager storage accounts, rather than giving access to the entire subscription, you can control access on a more finite level to the management plane using Role-Based Access Control (RBAC).

How to secure your storage account with Role-Based Access Control (RBAC)

Let's talk about what RBAC is, and how you can use it. Each Azure subscription has an Azure Active Directory. Users, groups, and applications from that directory can be granted access to manage resources in the Azure subscription that use the Resource Manager deployment model. This is referred to as Role-Based Access Control (RBAC). To manage this access, you can use the [Azure portal](#), the [Azure CLI tools](#), [PowerShell](#), or the [Azure Storage Resource Provider REST APIs](#).

With the Resource Manager model, you put the storage account in a resource group and control access to the management plane of that specific storage account using Azure Active Directory. For example, you can give specific users the ability to access the storage account keys, while other users can view information about the storage account, but cannot access the storage account keys.

Granting Access

Access is granted by assigning the appropriate RBAC role to users, groups, and applications, at the right scope. To grant access to the entire subscription, you assign a role at the subscription level. You can grant access to all of the resources in a resource group by granting permissions to the resource group itself. You can also assign specific roles to specific resources, such as storage accounts.

Here are the main points that you need to know about using RBAC to access the management operations of an Azure Storage account:

- When you assign access, you basically assign a role to the account that you want to have access. You can control access to the operations used to manage that storage account, but not to the data objects in the account. For example, you can grant permission to retrieve the properties of the storage account (such as redundancy), but not to a container or data within a container inside Blob Storage.
- For someone to have permission to access the data objects in the storage account, you can give them permission to read the storage account keys, and that user can then use those keys to access the blobs, queues, tables, and files.
- Roles can be assigned to a specific user account, a group of users, or to a specific application.
- Each role has a list of Actions and Not Actions. For example, the Virtual Machine Contributor role has an Action of "listKeys" that allows the storage account keys to be read. The Contributor has "Not Actions" such as updating the access for users in the Active Directory.
- Roles for storage include (but are not limited to) the following:
 - Owner – They can manage everything, including access.
 - Contributor – They can do anything the owner can do except assign access. Someone with this role can view and regenerate the storage account keys. With the storage account keys, they can access the data objects.

- Reader – They can view information about the storage account, except secrets. For example, if you assign a role with reader permissions on the storage account to someone, they can view the properties of the storage account, but they can't make any changes to the properties or view the storage account keys.
- Storage Account Contributor – They can manage the storage account – they can read the subscription's resource groups and resources, and create and manage subscription resource group deployments. They can also access the storage account keys, which in turn means they can access the data plane.
- User Access Administrator – They can manage user access to the storage account. For example, they can grant Reader access to a specific user.
- Virtual Machine Contributor – They can manage virtual machines but not the storage account to which they are connected. This role can list the storage account keys, which means that the user to whom you assign this role can update the data plane.

In order for a user to create a virtual machine, they have to be able to create the corresponding VHD file in a storage account. To do that, they need to be able to retrieve the storage account key and pass it to the API creating the VM. Therefore, they must have this permission so they can list the storage account keys.

- The ability to define custom roles is a feature that allows you to compose a set of actions from a list of available actions that can be performed on Azure resources.
- The user has to be set up in your Azure Active Directory before you can assign a role to them.
- You can create a report of who granted/revoked what kind of access to/from whom and on what scope using PowerShell or the Azure CLI.

Resources

- [Azure Active Directory Role-based Access Control](#)

This article explains the Azure Active Directory Role-based Access Control and how it works.

- [RBAC: Built in Roles](#)

This article details all of the built-in roles available in RBAC.

- [Understanding Resource Manager deployment and classic deployment](#)

This article explains the Resource Manager deployment and classic deployment models, and explains the benefits of using the Resource Manager and resource groups. It explains how the Azure Compute, Network, and Storage Providers work under the Resource Manager model.

- [Managing Role-Based Access Control with the REST API](#)

This article shows how to use the REST API to manage RBAC.

- [Azure Storage Resource Provider REST API Reference](#)

This is the reference for the APIs you can use to manage your storage account programmatically.

- [Developer's guide to auth with Azure Resource Manager API](#)

This article shows how to authenticate using the Resource Manager APIs.

- [Role-Based Access Control for Microsoft Azure from Ignite](#)

This is a link to a video on Channel 9 from the 2015 MS Ignite conference. In this session, they talk about access management and reporting capabilities in Azure, and explore best practices around securing access to Azure subscriptions using Azure Active Directory.

Managing Your Storage Account Keys

Storage account keys are 512-bit strings created by Azure that, along with the storage account name, can be used to access the data objects stored in the storage account, e.g. blobs, entities within a table, queue messages, and files

on an Azure Files share. Controlling access to the storage account keys controls access to the data plane for that storage account.

Each storage account has two keys referred to as "Key 1" and "Key 2" in the [Azure portal](#) and in the PowerShell cmdlets. These can be regenerated manually using one of several methods, including, but not limited to using the [Azure portal](#), PowerShell, the Azure CLI, or programmatically using the .NET Storage Client Library or the Azure Storage Services REST API.

There are any number of reasons to regenerate your storage account keys.

- You might regenerate them on a regular basis for security reasons.
- You would regenerate your storage account keys if someone managed to hack into an application and retrieve the key that was hardcoded or saved in a configuration file, giving them full access to your storage account.
- Another case for key regeneration is if your team is using a Storage Explorer application that retains the storage account key, and one of the team members leaves. The application would continue to work, giving them access to your storage account after they're gone. This is actually the primary reason they created account-level Shared Access Signatures – you can use an account-level SAS instead of storing the access keys in a configuration file.

Key regeneration plan

You don't want to just regenerate the key you are using without some planning. If you do that, you could cut off all access to that storage account, which can cause major disruption. This is why there are two keys. You should regenerate one key at a time.

Before you regenerate your keys, be sure you have a list of all of your applications that are dependent on the storage account, as well as any other services you are using in Azure. For example, if you are using Azure Media Services that are dependent on your storage account, you must re-sync the access keys with your media service after you regenerate the key. If you are using any applications such as a storage explorer, you will need to provide the new keys to those applications as well. Note that if you have VMs whose VHD files are stored in the storage account, they will not be affected by regenerating the storage account keys.

You can regenerate your keys in the Azure portal. Once keys are regenerated they can take up to 10 minutes to be synchronized across Storage Services.

When you're ready, here's the general process detailing how you should change your key. In this case, the assumption is that you are currently using Key 1 and you are going to change everything to use Key 2 instead.

1. Regenerate Key 2 to ensure that it is secure. You can do this in the Azure portal.
2. In all of the applications where the storage key is stored, change the storage key to use Key 2's new value. Test and publish the application.
3. After all of the applications and services are up and running successfully, regenerate Key 1. This ensures that anybody to whom you have not expressly given the new key will no longer have access to the storage account.

If you are currently using Key 2, you can use the same process, but reverse the key names.

You can migrate over a couple of days, changing each application to use the new key and publishing it. After all of them are done, you should then go back and regenerate the old key so it no longer works.

Another option is to put the storage account key in an [Azure Key Vault](#) as a secret and have your applications retrieve the key from there. Then when you regenerate the key and update the Azure Key Vault, the applications will not need to be redeployed because they will pick up the new key from the Azure Key Vault automatically. Note that you can have the application read the key each time you need it, or you can cache it in memory and if it fails when using it, retrieve the key again from the Azure Key Vault.

Using Azure Key Vault also adds another level of security for your storage keys. If you use this method, you will never have the storage key hardcoded in a configuration file, which removes that avenue of somebody getting access to the keys without specific permission.

Another advantage of using Azure Key Vault is you can also control access to your keys using Azure Active Directory. This means you can grant access to the handful of applications that need to retrieve the keys from Azure Key Vault, and know that other applications will not be able to access the keys without granting them permission specifically.

Note: it is recommended to use only one of the keys in all of your applications at the same time. If you use Key 1 in some places and Key 2 in others, you will not be able to rotate your keys without some application losing access.

Resources

- [About Azure Storage Accounts](#)

This article gives an overview of storage accounts and discusses viewing, copying, and regenerating storage access keys.

- [Azure Storage Resource Provider REST API Reference](#)

This article contains links to specific articles about retrieving the storage account keys and regenerating the storage account keys for an Azure Account using the REST API. Note: This is for Resource Manager storage accounts.

- [Operations on storage accounts](#)

This article in the Storage Service Manager REST API Reference contains links to specific articles on retrieving and regenerating the storage account keys using the REST API. Note: This is for the Classic storage accounts.

- [Say goodbye to key management – manage access to Azure Storage data using Azure AD](#)

This article shows how to use Active Directory to control access to your Azure Storage keys in Azure Key Vault. It also shows how to use an Azure Automation job to regenerate the keys on an hourly basis.

Data Plane Security

Data Plane Security refers to the methods used to secure the data objects stored in Azure Storage – the blobs, queues, tables, and files. We've seen methods to encrypt the data and security during transit of the data, but how do you go about allowing access to the objects?

There are basically two methods for controlling access to the data objects themselves. The first is by controlling access to the storage account keys, and the second is using Shared Access Signatures to grant access to specific data objects for a specific amount of time.

One exception to note is that you can allow public access to your blobs by setting the access level for the container that holds the blobs accordingly. If you set access for a container to Blob or Container, it will allow public read access for the blobs in that container. This means anyone with a URL pointing to a blob in that container can open it in a browser without using a Shared Access Signature or having the storage account keys.

Storage Account Keys

Storage account keys are 512-bit strings created by Azure that, along with the storage account name, can be used to access the data objects stored in the storage account.

For example, you can read blobs, write to queues, create tables, and modify files. Many of these actions can be performed through the Azure portal, or using one of many Storage Explorer applications. You can also write code to use the REST API or one of the Storage Client Libraries to perform these operations.

As discussed in the section on the [Management Plane Security](#), access to the storage keys for a Classic storage account can be granted by giving full access to the Azure subscription. Access to the storage keys for a storage account using the Azure Resource Manager model can be controlled through Role-Based Access Control (RBAC).

How to delegate access to objects in your account using Shared Access Signatures and Stored Access Policies

A Shared Access Signature is a string containing a security token that can be attached to a URI that allows you to delegate access to storage objects and specify constraints such as the permissions and the date/time range of access.

You can grant access to blobs, containers, queue messages, files, and tables. With tables, you can actually grant permission to access a range of entities in the table by specifying the partition and row key ranges to which you want the user to have access. For example, if you have data stored with a partition key of geographical state, you could give someone access to just the data for California.

In another example, you might give a web application a SAS token that enables it to write entries to a queue, and give a worker role application a SAS token to get messages from the queue and process them. Or you could give one customer a SAS token they can use to upload pictures to a container in Blob Storage, and give a web application permission to read those pictures. In both cases, there is a separation of concerns – each application can be given just the access that they require in order to perform their task. This is possible through the use of Shared Access Signatures.

Why you want to use Shared Access Signatures

Why would you want to use an SAS instead of just giving out your storage account key, which is so much easier? Giving out your storage account key is like sharing the keys of your storage kingdom. It grants complete access. Someone could use your keys and upload their entire music library to your storage account. They could also replace your files with virus-infected versions, or steal your data. Giving away unlimited access to your storage account is something that should not be taken lightly.

With Shared Access Signatures, you can give a client just the permissions required for a limited amount of time. For example, if someone is uploading a blob to your account, you can grant them write access for just enough time to upload the blob (depending on the size of the blob, of course). And if you change your mind, you can revoke that access.

Additionally, you can specify that requests made using a SAS are restricted to a certain IP address or IP address range external to Azure. You can also require that requests are made using a specific protocol (HTTPS or HTTP/HTTPS). This means if you only want to allow HTTPS traffic, you can set the required protocol to HTTPS only, and HTTP traffic will be blocked.

Definition of a Shared Access Signature

A Shared Access Signature is a set of query parameters appended to the URL pointing at the resource that provides information about the access allowed and the length of time for which the access is permitted. Here is an example; this URI provides read access to a blob for five minutes. Note that SAS query parameters must be URL Encoded, such as %3A for colon (:) or %20 for a space.

```
http://mystorage.blob.core.windows.net/mycontainer/myblob.txt (URL to the blob)
?sv=2015-04-05 (storage service version)
&st=2015-12-10T22%3A18%3A26Z (start time, in UTC time and URL encoded)
&se=2015-12-10T22%3A23%3A26Z (end time, in UTC time and URL encoded)
&sr=b (resource is a blob)
&sp=r (read access)
&sip=168.1.5.60-168.1.5.70 (requests can only come from this range of IP addresses)
&spr=https (only allow HTTPS requests)
&sig=Z%2FRHIX5Xcg0Mq2rqI301WTjEg2tYkboXr1P9ZUXDtkk%3D (signature used for the authentication of the SAS)
```

How the Shared Access Signature is authenticated by the Azure Storage Service

When the storage service receives the request, it takes the input query parameters and creates a signature using the same method as the calling program. It then compares the two signatures. If they agree, then the storage service can check the storage service version to make sure it's valid, verify that the current date and time are within the specified window, make sure the access requested corresponds to the request made, etc.

For example, with our URL above, if the URL was pointing to a file instead of a blob, this request would fail because it specifies that the Shared Access Signature is for a blob. If the REST command being called was to update a blob, it

would fail because the Shared Access Signature specifies that only read access is permitted.

Types of Shared Access Signatures

- A service-level SAS can be used to access specific resources in a storage account. Some examples of this are retrieving a list of blobs in a container, downloading a blob, updating an entity in a table, adding messages to a queue or uploading a file to a file share.
- An account-level SAS can be used to access anything that a service-level SAS can be used for. Additionally, it can give options to resources that are not permitted with a service-level SAS, such as the ability to create containers, tables, queues, and file shares. You can also specify access to multiple services at once. For example, you might give someone access to both blobs and files in your storage account.

Creating an SAS URI

1. You can create an ad hoc URI on demand, defining all of the query parameters each time.

This is really flexible, but if you have a logical set of parameters that are similar each time, using a Stored Access Policy is a better idea.

2. You can create a Stored Access Policy for an entire container, file share, table, or queue. Then you can use this as the basis for the SAS URIs you create. Permissions based on Stored Access Policies can be easily revoked. You can have up to 5 policies defined on each container, queue, table, or file share.

For example, if you were going to have many people read the blobs in a specific container, you could create a Stored Access Policy that says "give read access" and any other settings that will be the same each time. Then you can create an SAS URI using the settings of the Stored Access Policy and specifying the expiration date/time. The advantage of this is that you don't have to specify all of the query parameters every time.

Revocation

Suppose your SAS has been compromised, or you want to change it because of corporate security or regulatory compliance requirements. How do you revoke access to a resource using that SAS? It depends on how you created the SAS URI.

If you are using ad hoc URI's, you have three options. You can issue SAS tokens with short expiration policies and simply wait for the SAS to expire. You can rename or delete the resource (assuming the token was scoped to a single object). You can change the storage account keys. This last option can have a big impact, depending on how many services are using that storage account, and probably isn't something you want to do without some planning.

If you are using a SAS derived from a Stored Access Policy, you can remove access by revoking the Stored Access Policy – you can just change it so it has already expired, or you can remove it altogether. This takes effect immediately, and invalidates every SAS created using that Stored Access Policy. Updating or removing the Stored Access Policy may impact people accessing that specific container, file share, table, or queue via SAS, but if the clients are written so they request a new SAS when the old one becomes invalid, this will work fine.

Because using a SAS derived from a Stored Access Policy gives you the ability to revoke that SAS immediately, it is the recommended best practice to always use Stored Access Policies when possible.

Resources

For more detailed information on using Shared Access Signatures and Stored Access Policies, complete with examples, please refer to the following articles:

- These are the reference articles.

- [Service SAS](#)

This article provides examples of using a service-level SAS with blobs, queue messages, table ranges, and files.

- [Constructing a service SAS](#)

- [Constructing an account SAS](#)
- These are tutorials for using the .NET client library to create Shared Access Signatures and Stored Access Policies.
 - [Using Shared Access Signatures \(SAS\)](#)
 - [Shared Access Signatures, Part 2: Create and Use a SAS with the Blob Service](#)

This article includes an explanation of the SAS model, examples of Shared Access Signatures, and recommendations for the best practice use of SAS. Also discussed is the revocation of the permission granted.

- Limiting access by IP Address (IP ACLs)
 - [What is an endpoint Access Control List \(ACLs\)?](#)
 - [Constructing a Service SAS](#)

This is the reference article for service-level SAS; it includes an example of IP ACLing.

- [Constructing an Account SAS](#)

This is the reference article for account-level SAS; it includes an example of IP ACLing.

- Authentication
 - [Authentication for the Azure Storage Services](#)
- Shared Access Signatures Getting Started Tutorial
 - [SAS Getting Started Tutorial](#)

Encryption in Transit

Transport-Level Encryption – Using HTTPS

Another step you should take to ensure the security of your Azure Storage data is to encrypt the data between the client and Azure Storage. The first recommendation is to always use the [HTTPS](#) protocol, which ensures secure communication over the public Internet.

You should always use HTTPS when calling the REST APIs or accessing objects in storage. Also, **Shared Access Signatures**, which can be used to delegate access to Azure Storage objects, include an option to specify that only the HTTPS protocol can be used when using Shared Access Signatures, ensuring that anybody sending out links with SAS tokens will use the proper protocol.

Resources

- [Enable HTTPS for an app in Azure App Service](#)

This article shows you how to enable HTTPS for an Azure Web App.

Using encryption during transit with Azure File Shares

Azure File Storage supports HTTPS when using the REST API, but is more commonly used as an SMB file share attached to a VM. SMB 2.1 does not support encryption, so connections are only allowed within the same region in Azure. However, SMB 3.0 supports encryption, and can be used with Windows Server 2012 R2, Windows 8, Windows 8.1, and Windows 10, allowing cross-region access and even access on the desktop.

Note that while Azure File Shares can be used with Unix, the Linux SMB client does not yet support encryption, so access is only allowed within an Azure region. Encryption support for Linux is on the roadmap of Linux developers responsible for SMB functionality. When they add encryption, you will have the same ability for accessing an Azure File Share on Linux as you do for Windows.

Resources

- [How to use Azure File Storage with Linux](#)

This article shows how to mount an Azure File Share on a Linux system and upload/download files.

- [Get started with Azure File storage on Windows](#)

This article gives an overview of Azure File shares and how to mount and use them using PowerShell and .NET.

- [Inside Azure File Storage](#)

This article announces the general availability of Azure File Storage and provides technical details about the SMB 3.0 encryption.

Using Client-side encryption to secure data that you send to storage

Another option that helps you ensure that your data is secure while being transferred between a client application and Storage is Client-side Encryption. The data is encrypted before being transferred into Azure Storage. When retrieving the data from Azure Storage, the data is decrypted after it is received on the client side. Even though the data is encrypted going across the wire, we recommend that you also use HTTPS, as it has data integrity checks built in which help mitigate network errors affecting the integrity of the data.

Client-side encryption is also a method for encrypting your data at rest, as the data is stored in its encrypted form. We'll talk about this in more detail in the section on [Encryption at Rest](#).

Encryption at Rest

There are three Azure features that provide encryption at rest. Azure Disk Encryption is used to encrypt the OS and data disks in IaaS Virtual Machines. The other two – Client-side Encryption and SSE – are both used to encrypt data in Azure Storage. Let's look at each of these, and then do a comparison and see when each one can be used.

While you can use Client-side Encryption to encrypt the data in transit (which is also stored in its encrypted form in Storage), you may prefer to simply use HTTPS during the transfer, and have some way for the data to be automatically encrypted when it is stored. There are two ways to do this -- Azure Disk Encryption and SSE. One is used to directly encrypt the data on OS and data disks used by VMs, and the other is used to encrypt data written to Azure Blob Storage.

Storage Service Encryption (SSE)

SSE allows you to request that the storage service automatically encrypt the data when writing it to Azure Storage. When you read the data from Azure Storage, it will be decrypted by the storage service before being returned. This enables you to secure your data without having to modify code or add code to any applications.

This is a setting that applies to the whole storage account. You can enable and disable this feature by changing the value of the setting. To do this, you can use the Azure portal, PowerShell, the Azure CLI, the Storage Resource Provider REST API, or the .NET Storage Client Library. By default, SSE is turned off.

At this time, the keys used for the encryption are managed by Microsoft. We generate the keys originally, and manage the secure storage of the keys as well as the regular rotation as defined by internal Microsoft policy. In the future, you will get the ability to manage your own encryption keys, and provide a migration path from Microsoft-managed keys to customer-managed keys.

This feature is available for Standard and Premium Storage accounts created using the Resource Manager deployment model. SSE applies only to block blobs, page blobs, and append blobs. The other types of data, including tables, queues, and files, will not be encrypted.

Data is only encrypted when SSE is enabled and the data is written to Blob Storage. Enabling or disabling SSE does not impact existing data. In other words, when you enable this encryption, it will not go back and encrypt data that already exists; nor will it decrypt the data that already exists when you disable SSE.

If you want to use this feature with a Classic storage account, you can create a new Resource Manager storage account and use AzCopy to copy the data to the new account.

Client-side Encryption

We mentioned client-side encryption when discussing the encryption of the data in transit. This feature allows you to programmatically encrypt your data in a client application before sending it across the wire to be written to Azure Storage, and to programmatically decrypt your data after retrieving it from Azure Storage.

This does provide encryption in transit, but it also provides the feature of Encryption at Rest. Note that although the data is encrypted in transit, we still recommend using HTTPS to take advantage of the built-in data integrity checks which help mitigate network errors affecting the integrity of the data.

An example of where you might use this is if you have a web application that stores blobs and retrieves blobs, and you want the application and data to be as secure as possible. In that case, you would use client-side encryption. The traffic between the client and the Azure Blob Service contains the encrypted resource, and nobody can interpret the data in transit and reconstitute it into your private blobs.

Client-side encryption is built into the Java and the .NET storage client libraries, which in turn use the Azure Key Vault APIs, making it pretty easy for you to implement. The process of encrypting and decrypting the data uses the envelope technique, and stores metadata used by the encryption in each storage object. For example, for blobs, it stores it in the blob metadata, while for queues, it adds it to each queue message.

For the encryption itself, you can generate and manage your own encryption keys. You can also use keys generated by the Azure Storage Client Library, or you can have the Azure Key Vault generate the keys. You can store your encryption keys in your on-premises key storage, or you can store them in an Azure Key Vault. Azure Key Vault allows you to grant access to the secrets in Azure Key Vault to specific users using Azure Active Directory. This means that not just anybody can read the Azure Key Vault and retrieve the keys you're using for client-side encryption.

Resources

- [Encrypt and decrypt blobs in Microsoft Azure Storage using Azure Key Vault](#)

This article shows how to use client-side encryption with Azure Key Vault, including how to create the KEK and store it in the vault using PowerShell.

- [Client-Side Encryption and Azure Key Vault for Microsoft Azure Storage](#)

This article gives an explanation of client-side encryption, and provides examples of using the storage client library to encrypt and decrypt resources from the four storage services. It also talks about Azure Key Vault.

Using Azure Disk Encryption to encrypt disks used by your virtual machines

Azure Disk Encryption is a new feature that is currently in preview. This feature allows you to encrypt the OS disks and Data disks used by an IaaS Virtual Machine. For Windows, the drives are encrypted using industry-standard BitLocker encryption technology. For Linux, the disks are encrypted using the DM-Crypt technology. This is integrated with Azure Key Vault to allow you to control and manage the disk encryption keys.

The Azure Disk Encryption solution supports the following three customer encryption scenarios:

- Enable encryption on new IaaS VMs created from customer-encrypted VHD files and customer-provided encryption keys, which are stored in Azure Key Vault.
- Enable encryption on new IaaS VMs created from the Azure Marketplace.
- Enable encryption on existing IaaS VMs already running in Azure.

NOTE

For Linux VMs already running in Azure, or new Linux VMs created from images in the Azure Marketplace, encryption of the OS disk is not currently supported. Encryption of the OS Volume for Linux VMs is supported only for VMs that were encrypted on-premises and uploaded to Azure. This restriction only applies to the OS disk; encryption of data volumes for a Linux VM is supported.

The solution supports the following for IaaS VMs for public preview release when enabled in Microsoft Azure:

- Integration with Azure Key Vault
- Standard [A, D and G series IaaS VMs](#)
- Enable encryption on IaaS VMs created using [Azure Resource Manager](#) model
- All Azure public [regions](#)

This feature ensures that all data on your virtual machine disks is encrypted at rest in Azure Storage.

Resources

- [Azure Disk Encryption for Windows and Linux IaaS Virtual Machines](#)

This article discusses the preview release of Azure Disk Encryption and provides a link to download the white paper.

Comparison of Azure Disk Encryption, SSE, and Client-Side Encryption

IaaS VMs and their VHD files

For disks used by IaaS VMs, we recommend using Azure Disk Encryption. You can turn on SSE to encrypt the VHD files that are used to back those disks in Azure Storage, but it only encrypts newly written data. This means if you create a VM and then enable SSE on the storage account that holds the VHD file, only the changes will be encrypted, not the original VHD file.

If you create a VM using an image from the Azure Marketplace, Azure performs a [shallow copy](#) of the image to your storage account in Azure Storage, and it is not encrypted even if you have SSE enabled. After it creates the VM and starts updating the image, SSE will start encrypting the data. For this reason, it's best to use Azure Disk Encryption on VMs created from images in the Azure Marketplace if you want them fully encrypted.

If you bring a pre-encrypted VM into Azure from on-premises, you will be able to upload the encryption keys to Azure Key Vault, and continue using the encryption for that VM that you were using on-premises. Azure Disk Encryption is enabled to handle this scenario.

If you have non-encrypted VHD from on-premises, you can upload it into the gallery as a custom image and provision a VM from it. If you do this using the Resource Manager templates, you can ask it to turn on Azure Disk Encryption when it boots up the VM.

When you add a data disk and mount it on the VM, you can turn on Azure Disk Encryption on that data disk. It will encrypt that data disk locally first, and then the service management layer will do a lazy write against storage so the storage content is encrypted.

Client-side encryption

Client-side encryption is the most secure method of encrypting your data, because it encrypts it before transit, and encrypts the data at rest. However, it does require that you add code to your applications using storage, which you may not want to do. In those cases, you can use HTTPS for your data in transit, and SSE to encrypt the data at rest.

With client-side encryption, you can encrypt table entities, queue messages, and blobs. With SSE, you can only encrypt blobs. If you need table and queue data to be encrypted, you should use client-side encryption.

Client-side encryption is managed entirely by the application. This is the most secure approach, but does require you to make programmatic changes to your application and put key management processes in place. You would use this when you want the extra security during transit, and you want your stored data to be encrypted.

Client-side encryption is more load on the client, and you have to account for this in your scalability plans, especially if you are encrypting and transferring a lot of data.

Storage Service Encryption (SSE)

SSE is managed by Azure Storage. Using SSE does not provide for the security of the data in transit, but it does encrypt the data as it is written to Azure Storage. There is no impact on the performance when using this feature.

You can only encrypt block blobs, append blobs, and page blobs using SSE. If you need to encrypt table data or queue data, you should consider using client-side encryption.

If you have an archive or library of VHD files that you use as a basis for creating new virtual machines, you can create a new storage account, enable SSE, and then upload the VHD files to that account. Those VHD files will be encrypted by Azure Storage.

If you have Azure Disk Encryption enabled for the disks in a VM and SSE enabled on the storage account holding the VHD files, it will work fine; it will result in any newly-written data being encrypted twice.

Storage Analytics

Using Storage Analytics to monitor authorization type

For each storage account, you can enable Azure Storage Analytics to perform logging and store metrics data. This is a great tool to use when you want to check the performance metrics of a storage account, or need to troubleshoot a storage account because you are having performance problems.

Another piece of data you can see in the storage analytics logs is the authentication method used by someone when they access storage. For example, with Blob Storage, you can see if they used a Shared Access Signature or the storage account keys, or if the blob accessed was public.

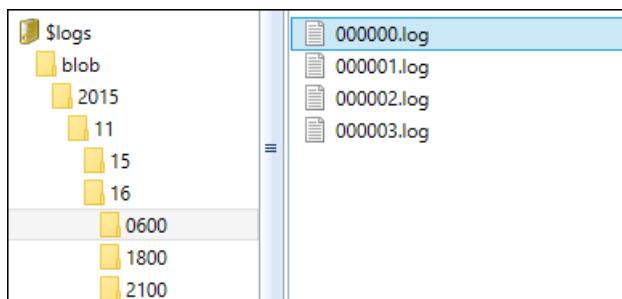
This can be really helpful if you are tightly guarding access to storage. For example, in Blob Storage you can set all of the containers to private and implement the use of an SAS service throughout your applications. Then you can check the logs regularly to see if your blobs are accessed using the storage account keys, which may indicate a breach of security, or if the blobs are public but they shouldn't be.

What do the logs look like?

After you enable the storage account metrics and logging through the Azure portal, analytics data will start to accumulate quickly. The logging and metrics for each service is separate; the logging is only written when there is activity in that storage account, while the metrics will be logged every minute, every hour, or every day, depending on how you configure it.

The logs are stored in block blobs in a container named \$logs in the storage account. This container is automatically created when Storage Analytics is enabled. Once this container is created, you can't delete it, although you can delete its contents.

Under the \$logs container, there is a folder for each service, and then there are subfolders for the year/month/day/hour. Under hour, the logs are simply numbered. This is what the directory structure will look like:



Every request to Azure Storage is logged. Here's a snapshot of a log file, showing the first few fields.

```

1.0;2015-11-16T06:13:26.9046078Z;GetBlobServiceProperties;Success;200;3;3;authenticated;mystorage;mystorage;blob;"https:
1.0;2015-11-16T06:13:27.2588724Z;GetBlobServiceProperties;Success;200;2;2;authenticated;mystorage;mystorage;blob;"https:
1.0;2015-11-16T06:14:28.0166751Z;GetBlobServiceProperties;Success;200;2;2;authenticated;mystorage;mystorage;blob;"https:
1.0;2015-11-16T06:14:29.2558837Z;GetBlobServiceProperties;Success;200;3;3;authenticated;mystorage;mystorage;blob;"https:
1.0;2015-11-16T06:14:43.4307865Z;BlobPreflightRequest;AnonymousSuccess;200;2;2;anonymous;;mystorage;mystorage;blob;"https://mystor
1.0;2015-11-16T06:14:43.4528051Z;GetBlobServiceProperties;Success;200;1;1;authenticated;mystorage;mystorage;blob;"https:
1.0;2015-11-16T06:15:30.3567270Z;GetBlobServiceProperties;Success;200;2;2;authenticated;mystorage;mystorage;blob;"https:
1.0;2015-11-16T06:15:29.2735098Z;GetBlobServiceProperties;Success;200;5;5;authenticated;mystorage;mystorage;blob;"https:
1.0;2015-11-16T06:16:32.9445742Z;GetBlobServiceProperties;Success;200;4;3;authenticated;mystorage;mystorage;blob;"https:
1.0;2015-11-16T06:16:44.2766486Z;ListContainers;Success;200;4;4;authenticated;mystorage;mystorage;blob;"https://mystorag
1.0;2015-11-16T06:16:56.0216743Z;CreateContainer;Success;201;10;10;authenticated;mystorage;mystorage;blob;"https://mysto
1.0;2015-11-16T06:16:56.0517020Z;ListContainers;Success;200;2;2;authenticated;mystorage;mystorage;blob;"https://mystorag
1.0;2015-11-16T06:16:59.9423538Z;ListContainers;Success;200;3;3;authenticated;mystorage;mystorage;blob;"https://mystorag
1.0;2015-11-16T06:16:59.9984102Z;ListBlobs;Success;200;3;3;authenticated;mystorage;mystorage;blob;"https://mystorage.blo
1.0;2015-11-16T06:17:23.7717291Z;GetBlobProperties;ClientOtherError;404;3;3;authenticated;mystorage;mystorage;blob;"http
1.0;2015-11-16T06:17:23.8347867Z;PutBlob;Success;201;71;8;authenticated;mystorage;mystorage;blob;"https://mystorage.blob
1.0;2015-11-16T06:17:23.9549008Z;GetBlobProperties;Success;200;2;2;authenticated;mystorage;mystorage;blob;"https://mysto
1.0;2015-11-16T06:17:31.9243814Z;GetBlobProperties;Success;200;2;2;authenticated;mystorage;mystorage;blob;"https://mysto
1.0;2015-11-16T06:17:31.9554107Z;GetBlob;Success;206;81;5;authenticated;mystorage;mystorage;blob;"https://mystorage.blob
1.0;2015-11-16T06:17:46.1437305Z;GetContainerACL;Success;200;2;2;authenticated;mystorage;mystorage;blob;"https://mystora
1.0;2015-11-16T06:16:30.3890982Z;GetBlobServiceProperties;Success;200;2;2;authenticated;mystorage;mystorage;blob;"https:

```

You can see that you can use the logs to track any kind of calls to a storage account.

What are all of those fields for?

There is an article listed in the resources below that provides the list of the many fields in the logs and what they are used for. Here is the list of fields in order:

```

<version-number>,<request-start-time>,<operation-type>,<request-status>,<http-status-code>,<end-
to-end-latency-in-ms>,<server-latency-in-ms>,<authentication-type>,<requester-account-
name>,<owner-account-name>,<service-type>,<request-url>,<requested-object-key>,<request-id-
header>,<operation-count>,<requester-ip-address>,<request-version-header>,<request-header-
size>,<request-packet-size>,<response-header-size>,<response-packet-size>,<request-content-
length>,<request-md5>,<server-md5>,<etag-identifier>,<last-modified-time>,<conditions-used>,<user-
agent-header>,<referrer-header>,<client-request-id>

```

We're interested in the entries for GetBlob, and how they are authenticated, so we need to look for entries with operation-type "Get-Blob", and check the request-status (4th column) and the authorization-type (8th column).

For example, in the first few rows in the listing above, the request-status is "Success" and the authorization-type is "authenticated". This means the request was validated using the storage account key.

How are my blobs being authenticated?

We have three cases that we are interested in.

1. The blob is public and it is accessed using a URL without a Shared Access Signature. In this case, the request-status is "AnonymousSuccess" and the authorization-type is "anonymous".

1.0;2015-11-17T02:01:29.0488963Z;GetBlob;**AnonymousSuccess**;200;124;37;**anonymous**;;mystorage...

2. The blob is private and was used with a Shared Access Signature. In this case, the request-status is "SASSuccess" and the authorization-type is "sas".

1.0;2015-11-16T18:30:05.6556115Z;GetBlob;**SASSuccess**;200;416;64;**sas**;;mystorage...

3. The blob is private and the storage key was used to access it. In this case, the request-status is "**Success**" and the authorization-type is "**authenticated**".

1.0;2015-11-16T18:32:24.3174537Z;GetBlob;**Success**;206;59;22;**authenticated**;mystorage...

You can use the Microsoft Message Analyzer to view and analyze these logs. It includes search and filter capabilities. For example, you might want to search for instances of GetBlob to see if the usage is what you expect, i.e. to make sure someone is not accessing your storage account inappropriately.

Resources

- [Storage Analytics](#)

This article is an overview of storage analytics and how to enable them.

- [Storage Analytics Log Format](#)

This article illustrates the Storage Analytics Log Format, and details the fields available therein, including authentication-type, which indicates the type of authentication used for the request.

- [Monitor a Storage Account in the Azure portal](#)

This article shows how to configure monitoring of metrics and logging for a storage account.

- [End-to-End Troubleshooting using Azure Storage Metrics and Logging, AzCopy, and Message Analyzer](#)

This article talks about troubleshooting using the Storage Analytics and shows how to use the Microsoft Message Analyzer.

- [Microsoft Message Analyzer Operating Guide](#)

This article is the reference for the Microsoft Message Analyzer and includes links to a tutorial, quick start, and feature summary.

Cross-Origin Resource Sharing (CORS)

Cross-domain access of resources

When a web browser running in one domain makes an HTTP request for a resource from a different domain, this is called a cross-origin HTTP request. For example, an HTML page served from contoso.com makes a request for a jpeg hosted on fabrikam.blob.core.windows.net. For security reasons, browsers restrict cross-origin HTTP requests initiated from within scripts, such as JavaScript. This means that when some JavaScript code on a web page on contoso.com requests that jpeg on fabrikam.blob.core.windows.net, the browser will not allow the request.

What does this have to do with Azure Storage? Well, if you are storing static assets such as JSON or XML data files in Blob Storage using a storage account called Fabrikam, the domain for the assets will be fabrikam.blob.core.windows.net, and the contoso.com web application will not be able to access them using JavaScript because the domains are different. This is also true if you're trying to call one of the Azure Storage Services – such as Table Storage – that return JSON data to be processed by the JavaScript client.

Possible solutions

One way to resolve this is to assign a custom domain like "storage.contoso.com" to fabrikam.blob.core.windows.net. The problem is that you can only assign that custom domain to one storage account. What if the assets are stored in multiple storage accounts?

Another way to resolve this is to have the web application act as a proxy for the storage calls. This means if you are uploading a file to Blob Storage, the web application would either write it locally and then copy it to Blob Storage, or it would read all of it into memory and then write it to Blob Storage. Alternately, you could write a dedicated web application (such as a Web API) that uploads the files locally and writes them to Blob Storage. Either way, you have to account for that function when determining the scalability needs.

How can CORS help?

Azure Storage allows you to enable CORS – Cross Origin Resource Sharing. For each storage account, you can specify domains that can access the resources in that storage account. For example, in our case outlined above, we can enable CORS on the fabrikam.blob.core.windows.net storage account and configure it to allow access to contoso.com. Then the web application contoso.com can directly access the resources in fabrikam.blob.core.windows.net.

One thing to note is that CORS allows access, but it does not provide authentication, which is required for all non-public access of storage resources. This means you can only access blobs if they are public or you include a Shared Access Signature giving you the appropriate permission. Tables, queues, and files have no public access, and require a SAS.

By default, CORS is disabled on all services. You can enable CORS by using the REST API or the storage client

library to call one of the methods to set the service policies. When you do that, you include a CORS rule, which is in XML. Here's an example of a CORS rule that has been set using the Set Service Properties operation for the Blob Service for a storage account. You can perform that operation using the storage client library or the REST APIs for Azure Storage.

```
<Cors>
  <CorsRule>
    <AllowedOrigins>http://www.contoso.com, http://www.fabrikam.com</AllowedOrigins>
    <AllowedMethods>PUT,GET</AllowedMethods>
    <AllowedHeaders>x-ms-meta-data*,x-ms-meta-target*,x-ms-meta-abc</AllowedHeaders>
    <ExposedHeaders>x-ms-meta-*</ExposedHeaders>
    <MaxAgeInSeconds>200</MaxAgeInSeconds>
  </CorsRule>
<Cors>
```

Here's what each row means:

- **AllowedOrigins** This tells which non-matching domains can request and receive data from the storage service. This says that both contoso.com and fabrikam.com can request data from Blob Storage for a specific storage account. You can also set this to a wildcard (*) to allow all domains to access requests.
- **AllowedMethods** This is the list of methods (HTTP request verbs) that can be used when making the request. In this example, only PUT and GET are allowed. You can set this to a wildcard (*) to allow all methods to be used.
- **AllowedHeaders** This is the request headers that the origin domain can specify when making the request. In this example, all metadata headers starting with x-ms-meta-data, x-ms-meta-target, and x-ms-meta-abc are permitted. The wildcard character (*) indicates that any header beginning with the specified prefix is allowed.
- **ExposedHeaders** This tells which response headers should be exposed by the browser to the request issuer. In this example, any header starting with "x-ms-meta-" will be exposed.
- **MaxAgeInSeconds** This is the maximum amount of time that a browser will cache the preflight OPTIONS request. (For more information about the preflight request, check the first article below.)

Resources

For more information about CORS and how to enable it, please check out these resources.

- [Cross-Origin Resource Sharing \(CORS\) Support for the Azure Storage Services on Azure.com](#)

This article provides an overview of CORS and how to set the rules for the different storage services.

- [Cross-Origin Resource Sharing \(CORS\) Support for the Azure Storage Services on MSDN](#)

This is the reference documentation for CORS support for the Azure Storage Services. This has links to articles applying to each storage service, and shows an example and explains each element in the CORS file.

- [Microsoft Azure Storage: Introducing CORS](#)

This is a link to the initial blog article announcing CORS and showing how to use it.

Frequently asked questions about Azure Storage security

1. How can I verify the integrity of the blobs I'm transferring into or out of Azure Storage if I can't use the HTTPS protocol?

If for any reason you need to use HTTP instead of HTTPS and you are working with block blobs, you can use MD5 checking to help verify the integrity of the blobs being transferred. This will help with protection from network/transport layer errors, but not necessarily with intermediary attacks.

If you can use HTTPS, which provides transport level security, then using MD5 checking is redundant and unnecessary.

For more information, please check out the [Azure Blob MD5 Overview](#).

2. What about FIPS-Compliance for the U.S. Government?

The United States Federal Information Processing Standard (FIPS) defines cryptographic algorithms approved for use by U.S. Federal government computer systems for the protection of sensitive data. Enabling FIPS mode on a Windows server or desktop tells the OS that only FIPS-validated cryptographic algorithms should be used. If an application uses non-compliant algorithms, the applications will break. With .NET Framework versions 4.5.2 or higher, the application automatically switches the cryptography algorithms to use FIPS-compliant algorithms when the computer is in FIPS mode.

Microsoft leaves it up to each customer to decide whether to enable FIPS mode. We believe there is no compelling reason for customers who are not subject to government regulations to enable FIPS mode by default.

Resources

- [Why We're Not Recommending "FIPS Mode" Anymore](#)

This blog article gives an overview of FIPS and explains why they don't enable FIPS mode by default.

- [FIPS 140 Validation](#)

This article provides information on how Microsoft products and cryptographic modules comply with the FIPS standard for the U.S. Federal government.

- ["System cryptography: Use FIPS compliant algorithms for encryption, hashing, and signing" security settings effects in Windows XP and in later versions of Windows](#)

This article talks about the use of FIPS mode in older Windows computers.

Azure Storage Service Encryption for Data at Rest

1/17/2017 • 8 min to read • [Edit on GitHub](#)

Azure Storage Service Encryption (SSE) for Data 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.

The following sections provide detailed guidance on how to use the Storage Service Encryption features as well as the supported scenarios and user experiences.

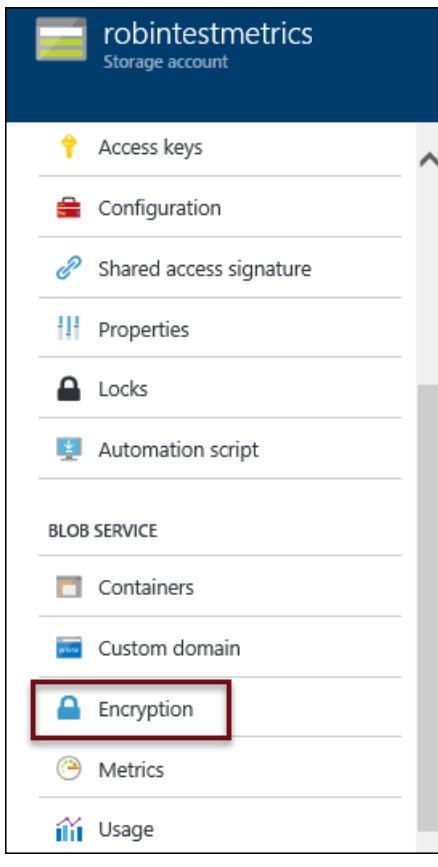
Overview

Azure Storage provides a comprehensive set of security capabilities which together enable developers to build secure applications. Data can be secured in transit between an application and Azure by using [Client-Side Encryption](#), HTTPS, or SMB 3.0. Storage Service Encryption provides encryption at rest, handling encryption, decryption, and key management in a totally transparent fashion. All data is encrypted using 256-bit [AES encryption](#), one of the strongest block ciphers available.

SSE works by encrypting the data when it is written to Azure Storage, and can be used for block blobs, page blobs and append blobs. It works for the following:

- General purpose storage accounts and Blob storage accounts
- Standard storage and Premium storage
- All redundancy levels (LRS, ZRS, GRS, RA-GRS)
- Azure Resource Manager storage accounts (but not classic)
- All regions

To enable or disable Storage Service Encryption for a storage account, log into the [Azure portal](#) and select a storage account. On the Settings blade, look for the Blob Service section as shown in this screenshot and click Encryption.



After you click the Encryption setting, you can enable or disable Storage Service Encryption.

Save Discard

Storage service encryption protects your data at rest. Azure Storage encrypts your data as it's written in our datacenters, and automatically decrypts it for you as you access it.

Currently, this feature is available only for the Azure Blob service. Note that after enabling Storage Service Encryption, only new data will be encrypted, and any existing blobs in this storage account will remain unencrypted.

[Learn more](#)

* Storage service encryption

Disabled Enabled

Encryption Scenarios

Storage Service Encryption can be enabled at a storage account level. It supports the following customer scenarios:

- Encryption of block blobs, append blobs, and page blobs.
- Encryption of archived VHDs and templates brought to Azure from on-premises.
- Encryption of underlying OS and data disks for IaaS VMs created using your VHDs.

SSE has the following limitations:

- Encryption of classic storage accounts is not supported.
- Encryption of classic storage accounts migrated to Resource Manager storage accounts is not supported.
- Existing Data - SSE only encrypts newly created data after the encryption is enabled. If for example you create a new Resource Manager storage account but don't turn on encryption, and then you upload blobs or archived VHDs to that storage account and then turn on SSE, those blobs will not be encrypted unless they are rewritten or copied.

- Marketplace Support - Enable encryption of VMs created from the Marketplace using the [Azure portal](#), PowerShell, and Azure CLI. The VHD base image will remain unencrypted; however, any writes done after the VM has spun up will be encrypted.
- Table, Queues, and Files data will not be encrypted.

Getting Started

Step 1: Create a new storage account.

Step 2: Enable encryption.

You can enable encryption using the [Azure portal](#).

NOTE

If you want to programmatically enable or disable the Storage Service Encryption on a storage account, you can use the [Azure Storage Resource Provider REST API](#), the [Storage Resource Provider Client Library for .NET](#), [Azure PowerShell](#), or the [Azure CLI](#).

Step 3: Copy data to storage account

If you enable SSE on a storage account and then write blobs to that storage account, the blobs will be encrypted. Any blobs already located in that storage account will not be encrypted until they are rewritten. You can copy the data from one storage account to one with SSE encrypted, or even enable SSE and copy the blobs from one container to another to ensure that previous data is encrypted. You can use any of the following tools to accomplish this.

Using AzCopy

AzCopy is a Windows command-line utility designed for copying data to and from Microsoft Azure Blob, File, and Table storage using simple commands with optimal performance. You can use this to copy your blobs from one storage account to another one that has SSE enabled.

To learn more, please visit [Transfer data with the AzCopy Command-Line Utility](#).

Using the Storage Client Libraries

You can copy blob data to and from blob storage or between storage accounts using our rich set of Storage Client Libraries including .NET, C++, Java, Android, Node.js, PHP, Python, and Ruby.

To learn more, please visit our [Get started with Azure Blob storage using .NET](#).

Using a Storage Explorer

You can use a Storage explorer to create storage accounts, upload and download data, view contents of blobs, and navigate through directories. You can use one of these to upload blobs to your storage account with encryption enabled. With some storage explorers, you can also copy data from existing blob storage to a different container in the storage account or a new storage account that has SSE enabled.

To learn more, please visit [Azure Storage Explorers](#).

Step 4: Query the status of the encrypted data

An updated version of the Storage Client libraries has been deployed that allows you to query the state of an object to determine if it is encrypted or not. Examples will be added to this document in the near future.

In the meantime, you can call [Get Account Properties](#) to verify that the storage account has encryption enabled or view the storage account properties in the Azure portal.

Encryption and Decryption Workflow

Here is a brief description of the encryption/decryption workflow:

- The customer enables encryption on the storage account.
- When the customer writes new data (PUT Blob, PUT Block, PUT Page, etc.) to Blob storage; every write is encrypted using 256-bit [AES encryption](#), one of the strongest block ciphers available.
- When the customer needs to access data (GET Blob, etc.), data is automatically decrypted before returning to the user.
- If encryption is disabled, new writes are no longer encrypted and existing encrypted data remains encrypted until rewritten by the user. While encryption is enabled, writes to Blob storage will be encrypted. The state of data does not change with the user toggling between enabling/disabling encryption for the storage account.
- All encryption keys are stored, encrypted, and managed by Microsoft.

Frequently asked questions about Storage Service Encryption for Data at Rest

Q: I have an existing classic storage account. Can I enable SSE on it?

A: No, SSE is only supported on Resource Manager storage accounts.

Q: How can I encrypt data in my classic storage account?

A: You can create a new Resource Manager storage account and copy your data using [AzCopy](#) from your existing classic storage account to your newly created Resource Manager storage account.

Another option is to migrate your classic storage account to a Resource Manager storage account. For more information, see [Platform Supported Migration of IaaS Resources from Classic to Resource Manager](#).

Q: I have an existing Resource Manager storage account. Can I enable SSE on it?

A: Yes, but only newly written blobs will be encrypted. It does not go back and encrypt data that was already present.

Q: I would like to encrypt the current data in an existing Resource Manager storage account?

A: You can enable SSE at any time in a Resource Manager storage account. However, blobs that were already present will not be encrypted. To encrypt those blobs, you can copy them to another name or another container and then remove the unencrypted versions.

Q: I'm using Premium storage; can I use SSE?

A: Yes, SSE is supported on both Standard Storage and Premium Storage.

Q: If I create a new storage account and enable SSE, then create a new VM using that storage account, does that mean my VM is encrypted?

A: Yes. Any disks created that use the new storage account will be encrypted, as long as they are created after SSE is enabled. If the VM was created using Azure Market Place, the VHD base image will remain unencrypted; however, any writes done after the VM has spun up will be encrypted.

Q: Can I create new storage accounts with SSE enabled using Azure PowerShell and Azure CLI?

A: Yes.

Q: How much more does Azure Storage cost if SSE is enabled?

A: There is no additional cost.

Q: Who manages the encryption keys?

A: The keys are managed by Microsoft.

Q: Can I use my own encryption keys?

A: We are working on providing capabilities for customers to bring their own encryption keys.

Q: Can I revoke access to the encryption keys?

A: Not at this time; the keys are fully managed by Microsoft.

Q: Is SSE enabled by default when I create a new storage account?

A: SSE is not enabled by default; you can use the Azure portal to enable it. You can also programmatically enable this feature using the Storage Resource Provider REST API.

Q: How is this different from Azure Drive Encryption?

A: This feature is used to encrypt data in Azure Blob storage. The Azure Disk Encryption is used to encrypt OS and Data disks in IaaS VMs. For more details, please visit our [Storage Security Guide](#).

Q: What if I enable SSE, and then go in and enable Azure Disk Encryption on the disks?

A: This will work seamlessly. Your data will be encrypted by both methods.

Q: My storage account is set up to be replicated geo-redundantly. If I enable SSE, will my redundant copy also be encrypted?

A: Yes, all copies of the storage account are encrypted, and all redundancy options – Locally Redundant Storage (LRS), Zone-Redundant Storage (ZRS), Geo-Redundant Storage (GRS), and Read Access Geo-Redundant Storage (RA-GRS) – are supported.

Q: I can't enable encryption on my storage account.

A: Is it a Resource Manager storage account? Classic storage accounts are not supported.

Q: Is SSE only permitted in specific regions?

A: The SSE is available in all regions.

Q: How do I contact someone if I have any issues or want to provide feedback?

A: Please contact ssediscussions@microsoft.com for any issues related to Storage Service Encryption.

Next Steps

Azure Storage provides a comprehensive set of security capabilities which together enable developers to build secure applications. For more details, visit the [Storage Security Guide](#).

Using Shared Access Signatures (SAS)

1/17/2017 • 25 min to read • [Edit on GitHub](#)

Overview

Using a shared access signature (SAS) is a powerful way to grant limited access to objects in your storage account to other clients, without having to expose your account key. In Part 1 of this tutorial on shared access signatures, we'll provide an overview of the SAS model and review SAS best practices.

For additional code examples using SAS beyond those presented here, see [Getting Started with Azure Blob Storage in .NET](#) and other samples available in the [Azure Code Samples](#) library. You can download the sample applications and run them, or browse the code on GitHub.

What is a shared access signature?

A shared access signature provides delegated access to resources in your storage account. With a SAS, you can grant clients access to resources in your storage account, without sharing your account keys. This is the key point of using shared access signatures in your applications — a SAS is a secure way to share your storage resources without compromising your account keys.

IMPORTANT

Your storage account key is similar to the root password for your storage account. Always be careful to protect your account key. Avoid distributing it to other users, hard-coding it, or saving it in a plain-text file that is accessible to others. Regenerate your account key using the Azure Portal if you believe it may have been compromised. To learn how to regenerate your account key, see [How to create, manage, or delete a storage account in the Azure Portal](#).

A SAS gives you granular control over what type of access you grant to clients who have the SAS, including:

- The interval over which the SAS is valid, including the start time and the expiry time.
- The permissions granted by the SAS. For example, a SAS on a blob might grant a user read and write permissions to that blob, but not delete permissions.
- An optional IP address or range of IP addresses from which Azure Storage will accept the SAS. For example, you might specify a range of IP addresses belonging to your organization. This provides another measure of security for your SAS.
- The protocol over which Azure Storage will accept the SAS. You can use this optional parameter to restrict access to clients using HTTPS.

When should you use a shared access signature?

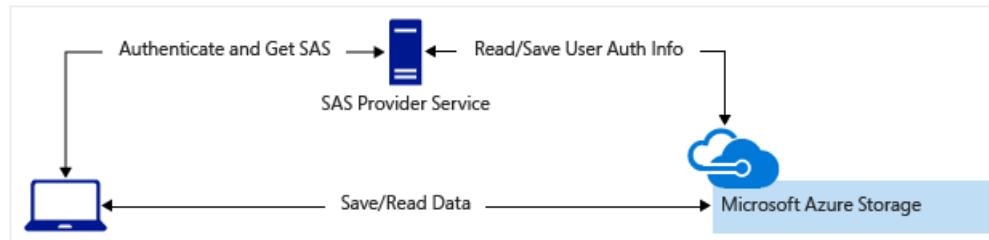
You can use a SAS when you want to provide access to resources in your storage account to a client that can't be trusted with the account key. Your storage account keys include both a primary and secondary key, both of which grant administrative access to your account and all of the resources in it. Exposing either of your account keys opens your account to the possibility of malicious or negligent use. Shared access signatures provide a safe alternative that allows other clients to read, write, and delete data in your storage account according to the permissions you've granted, and without need for the account key.

A common scenario where a SAS is useful is a service where users read and write their own data to your storage account. In a scenario where a storage account stores user data, there are two typical design patterns:

1. Clients upload and download data via a front-end proxy service, which performs authentication. This front-end proxy service has the advantage of allowing validation of business rules, but for large amounts of data or high-volume transactions, creating a service that can scale to match demand may be expensive or difficult.



2. A lightweight service authenticates the client as needed and then generates a SAS. Once the client receives the SAS, they can access storage account resources directly with the permissions defined by the SAS and for the interval allowed by the SAS. The SAS mitigates the need for routing all data through the front-end proxy service.



Many real-world services may use a hybrid of these two approaches, depending on the scenario involved, with some data processed and validated via the front-end proxy while other data is saved and/or read directly using SAS.

Additionally, you will need to use a SAS to authenticate the source object in a copy operation in certain scenarios:

- When you copy a blob to another blob that resides in a different storage account, you must use a SAS to authenticate the source blob. With version 2015-04-05, you can optionally use a SAS to authenticate the destination blob as well.
- When you copy a file to another file that resides in a different storage account, you must use a SAS to authenticate the source file. With version 2015-04-05, you can optionally use a SAS to authenticate the destination file as well.
- When you copy a blob to a file, or a file to a blob, you must use a SAS to authenticate the source object, even if the source and destination objects reside within the same storage account.

Types of shared access signatures

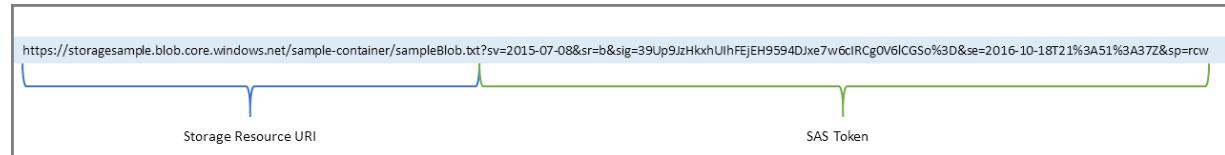
Version 2015-04-05 of Azure Storage introduces a new type of shared access signature, the account SAS. You can now create either of two types of shared access signatures:

- **Account SAS.** The account SAS delegates access to resources in one or more of the storage services. All of the operations available via a service SAS are also available via an account SAS. Additionally, with the account SAS, you can delegate access to operations that apply to a given service, such as **Get/Set Service Properties** and **Get Service Stats**. You can also delegate access to read, write, and delete operations on blob containers, tables, queues, and file shares that are not permitted with a service SAS. See [Constructing an Account SAS](#) for in-depth information about constructing the account SAS token.
- **Service SAS.** The service SAS delegates access to a resource in just one of the storage services: the Blob, Queue, Table, or File service. See [Constructing a Service SAS](#) and [Service SAS Examples](#) for in-depth information about constructing the service SAS token.

How a shared access signature works

A shared access signature is a signed URI that points to one or more storage resources and includes a token that contains a special set of query parameters. The token indicates how the resources may be accessed by the client. One of the query parameters, the signature, is constructed from the SAS parameters and signed with the account key. This signature is used by Azure Storage to authenticate the SAS.

Here's an example of a SAS URI, showing the resource URI and the SAS token:



Note that the SAS token is a string generated on the client side (see the [SAS examples](#) section below for code examples). The SAS token generated by the storage client library is not tracked by Azure Storage in any way. You can create an unlimited number of SAS tokens on the client side.

When a client provides a SAS URI to Azure Storage as part of a request, the service checks the SAS parameters and signature to verify that it is valid for authenticating the request. If the service verifies that the signature is valid, then the request is authenticated. Otherwise, the request is declined with error code 403 (Forbidden).

Shared access signature parameters

The account SAS and service SAS tokens include some common parameters, and also take a few parameters that are different.

Parameters common to account SAS and service SAS tokens

- **Api version** An optional parameter that specifies the storage service version to use to execute the request.
- **Service version** A required parameter that specifies the storage service version to use to authenticate the request.
- **Start time.** This is the time at which the SAS becomes valid. The start time for a shared access signature is optional; if omitted, the SAS is effective immediately. Must be expressed in UTC (Coordinated Universal Time), with a special UTC designator ("Z") i.e. 1994-11-05T13:15:30Z.
- **Expiry time.** This is the time after which the SAS is no longer valid. Best practices recommend that you either specify an expiry time for a SAS, or associate it with a stored access policy. Must be expressed in UTC (Coordinated Universal Time), with a special UTC designator ("Z") i.e. 1994-11-05T13:15:30Z (see more below).
- **Permissions.** The permissions specified on the SAS indicate what operations the client can perform against the storage resource using the SAS. Available permissions differ for an account SAS and a service SAS.
- **IP.** An optional parameter that specifies an IP address or a range of IP addresses outside of Azure (see the section [Routing session configuration state](#) for Express Route) from which to accept requests.
- **Protocol.** An optional parameter that specifies the protocol permitted for a request. Possible values are both HTTPS and HTTP (https,http), which is the default value, or HTTPS only (https). Note that HTTP only is not a permitted value.
- **Signature.** The signature is constructed from the other parameters specified as part token and then encrypted. It's used to authenticate the SAS.

Parameters for an account SAS token

- **Service or services.** An account SAS can delegate access to one or more of the storage services. For example, you can create an account SAS that delegates access to the Blob and File service. Or you can create a SAS that delegates access to all four services (Blob, Queue, Table, and File).
- **Storage resource types.** An account SAS applies to one or more classes of storage resources, rather than a specific resource. You can create an account SAS to delegate access to:

- Service-level APIs, which are called against the storage account resource. Examples include **Get/Set Service Properties**, **Get Service Stats**, and **List Containers/Queues/Tables/Shares**.
- Container-level APIs, which are called against the container objects for each service: blob containers, queues, tables, and file shares. Examples include **Create/Delete Container**, **Create/Delete Queue**, **Create/Delete Table**, **Create/Delete Share**, and **List Blobs/Files and Directories**.
- Object-level APIs, which are called against blobs, queue messages, table entities, and files. For example, **Put Blob**, **Query Entity**, **Get Messages**, and **Create File**.

Parameters for a service SAS token

- **Storage resource.** Storage resources for which you can delegate access with a service SAS include:
 - Containers and blobs
 - File shares and files
 - Queues
 - Tables and ranges of table entities.

Examples of SAS URLs

Here is an example of a service SAS URI that provides read and write permissions to a blob. The table breaks down each part of the URI to understand how it contributes to the SAS:

```
https://myaccount.blob.core.windows.net/sascontainer/sasblob.txt?sv=2015-04-05&st=2015-04-29T22%3A18%3A26Z&se=2015-04-30T02%3A23%3A26Z&sr=b&sp=rw&sip=168.1.5.60-168.1.5.70&spr=https&sig=Z%2FRHIX5Xcg0Mq2rqI301WTjEg2tYkboXr1P9ZUXDtkk%3D
```

NAME	SAS PORTION	DESCRIPTION
Blob URI	https://myaccount.blob.core.windows.net/sascontainer/sasblob.txt	The address of the blob. Note that using HTTPS is highly recommended.
Storage services version	sv=2015-04-05	For storage services version 2012-02-12 and later, this parameter indicates the version to use.
Start time	st=2015-04-29T22%3A18%3A26Z	Specified in UTC time. If you want the SAS to be valid immediately, omit the start time.
Expiry time	se=2015-04-30T02%3A23%3A26Z	Specified in UTC time.
Resource	sr=b	The resource is a blob.
Permissions	sp=rw	The permissions granted by the SAS include Read (r) and Write (w).
IP range	sip=168.1.5.60-168.1.5.70	The range of IP addresses from which a request will be accepted.
Protocol	spr=https	Only requests using HTTPS are permitted.

NAME	SAS PORTION	DESCRIPTION
Signature	sig=Z%2FRHIX5Xcg0Mq2rl3OlWTjEg2tYkboXr1P9ZUXDtkk%3D	Used to authenticate access to the blob. The signature is an HMAC computed over a string-to-sign and key using the SHA256 algorithm, and then encoded using Base64 encoding.

And here is an example of an account SAS that uses the same common parameters on the token. Since these parameters are described above, they are not described here. Only the parameters that are specific to account SAS are described in the table below.

```
https://myaccount.blob.core.windows.net/?restype=service&comp=properties&sv=2015-04-05&ss=bf&srt=s&st=2015-04-29T22%3A18%3A26Z&se=2015-04-30T02%3A23%3A26Z&sr=b&sp=rw&ip=168.1.5.60-168.1.5.70&spr=https&sig=F%6GRVAZ5Cdj2Pw4tgU7I1STkWgn7bUkkAg8P6HESXwmf%4B
```

NAME	SAS PORTION	DESCRIPTION
Resource URI	https://myaccount.blob.core.windows.net/ ?restype=service&comp=properties	The Blob service endpoint, with parameters for getting service properties (when called with GET) or setting service properties (when called with SET).
Services	ss=bf	The SAS applies to the Blob and File services
Resource types	srt=s	The SAS applies to service-level operations.
Permissions	sp=rw	The permissions grant access to read and write operations.

Given that permissions are restricted to the service level, accessible operations with this SAS are **Get Blob Service Properties** (read) and **Set Blob Service Properties** (write). However, with a different resource URI, the same SAS token could also be used to delegate access to **Get Blob Service Stats** (read).

Controlling a SAS with a stored access policy

A shared access signature can take one of two forms:

- **Ad hoc SAS:** When you create an ad hoc SAS, the start time, expiry time, and permissions for the SAS are all specified on the SAS URI (or implied, in the case where start time is omitted). This type of SAS may be created as an account SAS or a service SAS.
- **SAS with stored access policy:** A stored access policy is defined on a resource container - a blob container, table, queue, or file share - and can be used to manage constraints for one or more shared access signatures. When you associate a SAS with a stored access policy, the SAS inherits the constraints - the start time, expiry time, and permissions - defined for the stored access policy.

NOTE

Currently, an account SAS must be an ad hoc SAS. Stored access policies are not yet supported for account SAS.

The difference between the two forms is important for one key scenario: revocation. A SAS is a URL, so anyone who obtains the SAS can use it, regardless of who requested it to begin with. If a SAS is published publicly, it

can be used by anyone in the world. A SAS that is distributed is valid until one of four things happens:

1. The expiry time specified on the SAS is reached.
2. The expiry time specified on the stored access policy referenced by the SAS is reached (if a stored access policy is referenced, and if it specifies an expiry time). This can either occur because the interval elapses, or because you have modified the stored access policy to have an expiry time in the past, which is one way to revoke the SAS.
3. The stored access policy referenced by the SAS is deleted, which is another way to revoke the SAS. Note that if you recreate the stored access policy with exactly the same name, all existing SAS tokens will again be valid according to the permissions associated with that stored access policy (assuming that the expiry time on the SAS has not passed). If you are intending to revoke the SAS, be sure to use a different name if you recreate the access policy with an expiry time in the future.
4. The account key that was used to create the SAS is regenerated. Note that doing this will cause all application components using that account key to fail to authenticate until they are updated to use either the other valid account key or the newly regenerated account key.

IMPORTANT

A shared access signature URI is associated with the account key used to create the signature, and the associated stored access policy (if any). If no stored access policy is specified, the only way to revoke a shared access signature is to change the account key.

Authenticating from a client application with a SAS

A client who is in possession of a SAS can use the SAS to authenticate a request against a storage account for which they do not possess the account keys. A SAS can be included in a connection string, or used directly from the appropriate constructor or method.

Using a SAS in a connection string

If you possess a shared access signature (SAS) URL that grants you access to resources in a storage account, you can use the SAS in a connection string. Because the SAS includes on the URI the information required to authenticate the request, the SAS URI provides the protocol, the service endpoint, and the necessary credentials to access the resource.

To create a connection string that includes a shared access signature, specify the string in the following format:

```
BlobEndpoint=myBlobEndpoint;  
QueueEndpoint=myQueueEndpoint;  
TableEndpoint=myTableEndpoint;  
FileEndpoint=myFileEndpoint;  
SharedAccessSignature=sasToken
```

Each service endpoint is optional, although the connection string must contain at least one.

NOTE

Using HTTPS with a SAS is recommended as a best practice.

If you are specifying a SAS in a connection string in a configuration file, you may need to encode special characters in the URL.

Service SAS example

Here's an example of a connection string that includes a service SAS for Blob storage:

```
BlobEndpoint=https://storagesample.blob.core.windows.net;SharedAccessSignature=sv=2015-04-05&sr=b&si=tutorial-policy-635959936145100803&sig=9aCzs76n0E7y5BpEi2GvsSv433BZa22leD0ZXX%2BXXIU%3D
```

And here's an example of the same connection string with encoding of special characters:

```
BlobEndpoint=https://storagesample.blob.core.windows.net;SharedAccessSignature=sv=2015-04-05&sr=b&si=tutorial-policy-635959936145100803&sig=9aCzs76n0E7y5BpEi2GvsSv433BZa22leD0ZXX%2BXXIU%3D
```

Account SAS example

Here's an example of a connection string that includes an account SAS for Blob and File storage. Note that endpoints for both services are specified:

```
BlobEndpoint=https://storagesample.blob.core.windows.net;
FileEndpoint=https://storagesample.file.core.windows.net;
SharedAccessSignature=sv=2015-07-08&sig=iCvQmdZngZNW%2F4vw43j6%2BVz6fndHF5LI639QJba4r8o%3D&spr=https&st=2016-04-12T03%3A24%3A31Z&se=2016-04-13T03%3A29%3A31Z&srt=s&ss=bf&sp=rwl
```

And here's an example of the same connection string with URL encoding:

```
BlobEndpoint=https://storagesample.blob.core.windows.net;
FileEndpoint=https://storagesample.file.core.windows.net;
SharedAccessSignature=sv=2015-07-08&sig=iCvQmdZngZNW%2F4vw43j6%2BVz6fndHF5LI639QJba4r8o%3D&spr=https&st=2016-04-12T03%3A24%3A31Z&se=2016-04-13T03%3A29%3A31Z&srt=s&ss=bf&sp=rwl
```

Using a SAS in a constructor or method

Several Azure Storage client library constructors and method overloads offer a SAS parameter, so that you can authenticate a request to the service with a SAS.

For example, here a SAS URI is used to create a reference to a block blob. The SAS provides the only credentials needed for the request. The block blob reference is then used for a write operation:

```

string sasUri = "https://storagesample.blob.core.windows.net/sample-container/" +
    "sampleBlob.txt?sv=2015-07-08&sr=b&sig=39Up9JzHkxhUIhFEjEH9594DJxe7w6cIRCg0V6lCGSo%3D" +
    "&se=2016-10-18T21%3A51%3A37Z&sp=rcw";

CloudBlockBlob blob = new CloudBlockBlob(new Uri(sasUri));

// Create operation: Upload a blob with the specified name to the container.
// If the blob does not exist, it will be created. If it does exist, it will be overwritten.
try
{
    MemoryStream msWrite = new MemoryStream(Encoding.UTF8.GetBytes(blobContent));
    msWrite.Position = 0;
    using (msWrite)
    {
        await blob.UploadFromStreamAsync(msWrite);
    }

    Console.WriteLine("Create operation succeeded for SAS {0}", sasUri);
    Console.WriteLine();
}
catch (StorageException e)
{
    if (e.RequestInformation.HttpStatusCode == 403)
    {
        Console.WriteLine("Create operation failed for SAS {0}", sasUri);
        Console.WriteLine("Additional error information: " + e.Message);
        Console.WriteLine();
    }
    else
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
        throw;
    }
}
}

```

Best practices for using SAS

When you use shared access signatures in your applications, you need to be aware of two potential risks:

- If a SAS is leaked, it can be used by anyone who obtains it, which can potentially compromise your storage account.
- If a SAS provided to a client application expires and the application is unable to retrieve a new SAS from your service, then the application's functionality may be hindered.

The following recommendations for using shared access signatures will help balance these risks:

1. **Always use HTTPS** to create a SAS or to distribute a SAS. If a SAS is passed over HTTP and intercepted, an attacker performing a man-in-the-middle attack will be able to read the SAS and then use it just as the intended user could have, potentially compromising sensitive data or allowing for data corruption by the malicious user.
2. **Reference stored access policies where possible.** Stored access policies give you the option to revoke permissions without having to regenerate the storage account keys. Set the expiration on these to be a very long time (or infinite) and make sure that it is regularly updated to move it farther into the future.
3. **Use near-term expiration times on an ad hoc SAS.** In this way, even if a SAS is compromised unknowingly, it will only be viable for a short time duration. This practice is especially important if you cannot reference a stored access policy. This practice also helps limit the amount of data that can be written to a blob by limiting the time available to upload to it.
4. **Have clients automatically renew the SAS if necessary.** Clients should renew the SAS well before the expiration, in order to allow time for retries if the service providing the SAS is unavailable. If your SAS is

meant to be used for a small number of immediate, short-lived operations that are expected to be completed within the expiration period, then this may be unnecessary as the SAS is not expected to be renewed. However, if you have client that is routinely making requests via SAS, then the possibility of expiration comes into play. The key consideration is to balance the need for the SAS to be short-lived (as stated above) with the need to ensure that the client is requesting renewal early enough to avoid disruption due to the SAS expiring prior to successful renewal.

5. **Be careful with SAS start time.** If you set the start time for a SAS to **now**, then due to clock skew (differences in current time according to different machines), failures may be observed intermittently for the first few minutes. In general, set the start time to be at least 15 minutes ago, or don't set it at all, which will make it valid immediately in all cases. The same generally applies to expiry time as well - remember that you may observe up to 15 minutes of clock skew in either direction on any request. Note for clients using a REST version prior to 2012-02-12, the maximum duration for a SAS that does not reference a stored access policy is 1 hour, and any policies specifying longer term than that will fail.
6. **Be specific with the resource to be accessed.** A typical security best practice is to provide a user with the minimum required privileges. If a user only needs read access to a single entity, then grant them read access to that single entity, and not read/write/delete access to all entities. This also helps mitigate the threat of the SAS being compromised, as the SAS has less power in the hands of an attacker.
7. **Understand that your account will be billed for any usage, including that done with SAS.** If you provide write access to a blob, a user may choose to upload a 200GB blob. If you've given them read access as well, they may choose to download it 10 times, incurring 2TB in egress costs for you. Again, provide limited permissions, to help mitigate the potential of malicious users. Use short-lived SAS to reduce this threat (but be mindful of clock skew on the end time).
8. **Validate data written using SAS.** When a client application writes data to your storage account, keep in mind that there can be problems with that data. If your application requires that that data be validated or authorized before it is ready to use, you should perform this validation after the data is written and before it is used by your application. This practice also protects against corrupt or malicious data being written to your account, either by a user who properly acquired the SAS, or by a user exploiting a leaked SAS.
9. **Don't always use SAS.** Sometimes the risks associated with a particular operation against your storage account outweigh the benefits of SAS. For such operations, create a middle-tier service that writes to your storage account after performing business rule validation, authentication, and auditing. Also, sometimes it's simpler to manage access in other ways. For example, if you want to make all blobs in a container publically readable, you can make the container Public, rather than providing a SAS to every client for access.
10. **Use Storage Analytics to monitor your application.** You can use logging and metrics to observe any spike in authentication failures due to an outage in your SAS provider service or to the inadvertent removal of a stored access policy. See the [Azure Storage Team Blog](#) for additional information.

SAS examples

Below are some examples of both types of shared access signatures, account SAS and service SAS.

To run these examples, you'll need to download and reference these packages:

- [Azure Storage Client Library for .NET](#), version 6.x or later (to use account SAS).
- [Azure Configuration Manager](#)

For additional examples that show how to create and test a SAS, see [Azure Code Samples for Storage](#).

Example: Create and use an account SAS

The following code example creates an account SAS that is valid for the Blob and File services, and gives the client permissions read, write, and list permissions to access service-level APIs. The account SAS restricts the protocol to HTTPS, so the request must be made with HTTPS.

```

static string GetAccountSASToken()
{
    // To create the account SAS, you need to use your shared key credentials. Modify for your account.
    const string ConnectionString = "DefaultEndpointsProtocol=https;AccountName=account-
name;AccountKey=account-key";
    CloudStorageAccount storageAccount = CloudStorageAccount.Parse(ConnectionString);

    // Create a new access policy for the account.
    SharedAccessAccountPolicy policy = new SharedAccessAccountPolicy()
    {
        Permissions = SharedAccessAccountPermissions.Read | SharedAccessAccountPermissions.Write | 
SharedAccessAccountPermissions.List,
        Services = SharedAccessAccountServices.Blob | SharedAccessAccountServices.File,
        ResourceTypes = SharedAccessAccountResourceTypes.Service,
        SharedAccessExpiryTime = DateTime.UtcNow.AddHours(24),
        Protocols = SharedAccessProtocol.HttpsOnly
    };

    // Return the SAS token.
    return storageAccount.GetSharedAccessSignature(policy);
}

```

To use the account SAS to access service-level APIs for the Blob service, construct a Blob client object using the SAS and the Blob storage endpoint for your storage account.

```

static void UseAccountSAS(string sasToken)
{
    // Create new storage credentials using the SAS token.
    StorageCredentials accountSAS = new StorageCredentials(sasToken);
    // Use these credentials and the account name to create a Blob service client.
    CloudStorageAccount accountWithSAS = new CloudStorageAccount(accountSAS, "account-name", endpointSuffix:
null, useHttps: true);
    CloudBlobClient blobClientWithSAS = accountWithSAS.CreateCloudBlobClient();

    // Now set the service properties for the Blob client created with the SAS.
    blobClientWithSAS.SetServiceProperties(new ServiceProperties()
    {
        HourMetrics = new MetricsProperties()
        {
            MetricsLevel = MetricsLevel.ServiceAndApi,
            RetentionDays = 7,
            Version = "1.0"
        },
        MinuteMetrics = new MetricsProperties()
        {
            MetricsLevel = MetricsLevel.ServiceAndApi,
            RetentionDays = 7,
            Version = "1.0"
        },
        Logging = new LoggingProperties()
        {
            LoggingOperations = LoggingOperations.All,
            RetentionDays = 14,
            Version = "1.0"
        }
    });

    // The permissions granted by the account SAS also permit you to retrieve service properties.
    ServiceProperties serviceProperties = blobClientWithSAS.GetServiceProperties();
    Console.WriteLine(serviceProperties.HourMetrics.MetricsLevel);
    Console.WriteLine(serviceProperties.HourMetrics.RetentionDays);
    Console.WriteLine(serviceProperties.HourMetrics.Version);
}

```

Example: Create a stored access policy

The following code creates a stored access policy on a container. You can use the access policy to specify constraints for a service SAS on the container or its blobs.

```
private static async Task CreateSharedAccessPolicyAsync(CloudBlobContainer container, string policyName)
{
    // Create a new shared access policy and define its constraints.
    // The access policy provides create, write, read, list, and delete permissions.
    SharedAccessBlobPolicy sharedPolicy = new SharedAccessBlobPolicy()
    {
        // When the start time for the SAS is omitted, the start time is assumed to be the time when the
        // storage service receives the request.
        // Omitting the start time for a SAS that is effective immediately helps to avoid clock skew.
        SharedAccessExpiryTime = DateTime.UtcNow.AddHours(24),
        Permissions = SharedAccessBlobPermissions.Read | SharedAccessBlobPermissions.List |
                      SharedAccessBlobPermissions.Write | SharedAccessBlobPermissions.Create |
                      SharedAccessBlobPermissions.Delete
    };

    // Get the container's existing permissions.
    BlobContainerPermissions permissions = await container.GetPermissionsAsync();

    // Add the new policy to the container's permissions, and set the container's permissions.
    permissions.SharedAccessPolicies.Add(policyName, sharedPolicy);
    await container.SetPermissionsAsync(permissions);
}
```

Example: Create a service SAS on a container

The following code creates a SAS on a container. If the name of an existing stored access policy is provided, that policy is associated with the SAS. If no stored access policy is provided, then the code creates an ad-hoc SAS on the container.

```

private static string GetContainerSasUri(CloudBlobContainer container, string storedPolicyName = null)
{
    string sasContainerToken;

    // If no stored policy is specified, create a new access policy and define its constraints.
    if (storedPolicyName == null)
    {
        // Note that the SharedAccessBlobPolicy class is used both to define the parameters of an ad-hoc
        SAS, and
        // to construct a shared access policy that is saved to the container's shared access policies.
        SharedAccessBlobPolicy adHocPolicy = new SharedAccessBlobPolicy()
        {
            // When the start time for the SAS is omitted, the start time is assumed to be the time when the
            storage service receives the request.
            // Omitting the start time for a SAS that is effective immediately helps to avoid clock skew.
            SharedAccessExpiryTime = DateTime.UtcNow.AddHours(24),
            Permissions = SharedAccessBlobPermissions.Write | SharedAccessBlobPermissions.List
        };

        // Generate the shared access signature on the container, setting the constraints directly on the
        signature.
        sasContainerToken = container.GetSharedAccessSignature(adHocPolicy, null);

        Console.WriteLine("SAS for blob container (ad hoc): {0}", sasContainerToken);
        Console.WriteLine();
    }
    else
    {
        // Generate the shared access signature on the container. In this case, all of the constraints for
        the
        // shared access signature are specified on the stored access policy, which is provided by name.
        // It is also possible to specify some constraints on an ad-hoc SAS and others on the stored access
        policy.
        sasContainerToken = container.GetSharedAccessSignature(null, storedPolicyName);

        Console.WriteLine("SAS for blob container (stored access policy): {0}", sasContainerToken);
        Console.WriteLine();
    }

    // Return the URI string for the container, including the SAS token.
    return container.Uri + sasContainerToken;
}

```

Example: Create a service SAS on a blob

The following code creates a SAS on a blob. If the name of an existing stored access policy is provided, that policy is associated with the SAS. If no stored access policy is provided, then the code creates an ad-hoc SAS on the blob.

```

private static string GetBlobSasUri(CloudBlobContainer container, string blobName, string policyName = null)
{
    string sasBlobToken;

    // Get a reference to a blob within the container.
    // Note that the blob may not exist yet, but a SAS can still be created for it.
    CloudBlockBlob blob = container.GetBlockBlobReference(blobName);

    if (policyName == null)
    {
        // Create a new access policy and define its constraints.
        // Note that the SharedAccessBlobPolicy class is used both to define the parameters of an ad-hoc
        SAS, and
        // to construct a shared access policy that is saved to the container's shared access policies.
        SharedAccessBlobPolicy adHocSAS = new SharedAccessBlobPolicy()
        {
            // When the start time for the SAS is omitted, the start time is assumed to be the time when the
            storage service receives the request.
            // Omitting the start time for a SAS that is effective immediately helps to avoid clock skew.
            SharedAccessExpiryTime = DateTime.UtcNow.AddHours(24),
            Permissions = SharedAccessBlobPermissions.Read | SharedAccessBlobPermissions.Write |
            SharedAccessBlobPermissions.Create
        };

        // Generate the shared access signature on the blob, setting the constraints directly on the
        // signature.
        sasBlobToken = blob.GetSharedAccessSignature(adHocSAS);

        Console.WriteLine("SAS for blob (ad hoc): {0}", sasBlobToken);
        Console.WriteLine();
    }
    else
    {
        // Generate the shared access signature on the blob. In this case, all of the constraints for the
        // shared access signature are specified on the container's stored access policy.
        sasBlobToken = blob.GetSharedAccessSignature(null, policyName);

        Console.WriteLine("SAS for blob (stored access policy): {0}", sasBlobToken);
        Console.WriteLine();
    }

    // Return the URI string for the container, including the SAS token.
    return blob.Uri + sasBlobToken;
}

```

Conclusion

Shared access signatures are useful for providing limited permissions to your storage account to clients that should not have the account key. As such, they are a vital part of the security model for any application using Azure Storage. If you follow the best practices listed here, you can use SAS to provide greater flexibility of access to resources in your storage account, without compromising the security of your application.

Next Steps

- [Get Started with Azure File storage on Windows](#)
- [Manage anonymous read access to containers and blobs](#)
- [Delegating Access with a Shared Access Signature](#)
- [Introducing Table and Queue SAS](#)

Tutorial: Encrypt and decrypt blobs in Microsoft Azure Storage using Azure Key Vault

1/17/2017 • 7 min to read • [Edit on GitHub](#)

Introduction

This tutorial covers how to make use of client-side storage encryption with Azure Key Vault. It walks you through how to encrypt and decrypt a blob in a console application using these technologies.

Estimated time to complete: 20 minutes

For overview information about Azure Key Vault, see [What is Azure Key Vault?](#).

For overview information about client-side encryption for Azure Storage, see [Client-Side Encryption and Azure Key Vault for Microsoft Azure Storage](#).

Prerequisites

To complete this tutorial, you must have the following:

- An Azure Storage account
- Visual Studio 2013 or later
- Azure PowerShell

Overview of client-side encryption

For an overview of client-side encryption for Azure Storage, see [Client-Side Encryption and Azure Key Vault for Microsoft Azure Storage](#)

Here is a brief description of how client side encryption works:

1. The Azure Storage client SDK generates a content encryption key (CEK), which is a one-time-use symmetric key.
2. Customer data is encrypted using this CEK.
3. The CEK is then wrapped (encrypted) using the key encryption key (KEK). The KEK is identified by a key identifier and can be an asymmetric key pair or a symmetric key and can be managed locally or stored in Azure Key Vault. The Storage client itself never has access to the KEK. It just invokes the key wrapping algorithm that is provided by Key Vault. Customers can choose to use custom providers for key wrapping/unwrapping if they want.
4. The encrypted data is then uploaded to the Azure Storage service.

Set up your Azure Key Vault

In order to proceed with this tutorial, you need to do the following steps, which are outlined in the tutorial [Get started with Azure Key Vault](#):

- Create a key vault.
- Add a key or secret to the key vault.
- Register an application with Azure Active Directory.
- Authorize the application to use the key or secret.

Make note of the ClientID and ClientSecret that were generated when registering an application with Azure Active

Directory.

Create both keys in the key vault. We assume for the rest of the tutorial that you have used the following names: ContosoKeyVault and TestRSAKey1.

Create a console application with packages and AppSettings

In Visual Studio, create a new console application.

Add necessary nuget packages in the Package Manager Console.

```
Install-Package WindowsAzure.Storage

// This is the latest stable release for ADAL.
Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory -Version 2.16.204221202

Install-Package Microsoft.Azure.KeyVault
Install-Package Microsoft.Azure.KeyVault.Extensions
```

Add AppSettings to the App.Config.

```
<appSettings>
    <add key="accountName" value="myaccount"/>
    <add key="accountKey" value="theaccountkey"/>
    <add key="clientId" value="theclientid"/>
    <add key="clientSecret" value="theclientsecret"/>
    <add key="container" value="stuff"/>
</appSettings>
```

Add the following `using` statements and make sure to add a reference to System.Configuration to the project.

```
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using System.Configuration;
using Microsoft.WindowsAzure.Storage.Auth;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Blob;
using Microsoft.Azure.KeyVault;
using System.Threading;
using System.IO;
```

Add a method to get a token to your console application

The following method is used by Key Vault classes that need to authenticate for access to your key vault.

```
private async static Task<string> GetToken(string authority, string resource, string scope)
{
    var authContext = new AuthenticationContext(authority);
    ClientCredential clientCred = new ClientCredential(
        ConfigurationManager.AppSettings["clientId"],
        ConfigurationManager.AppSettings["clientSecret"]);
    AuthenticationResult result = await authContext.AcquireTokenAsync(resource, clientCred);

    if (result == null)
        throw new InvalidOperationException("Failed to obtain the JWT token");

    return result.AccessToken;
}
```

Access Storage and Key Vault in your program

In the Main function, add the following code.

```
// This is standard code to interact with Blob storage.  
StorageCredentials creds = new StorageCredentials(  
    ConfigurationManager.AppSettings["accountName"],  
    ConfigurationManager.AppSettings["accountKey"]);  
CloudStorageAccount account = new CloudStorageAccount(creds, useHttps: true);  
CloudBlobClient client = account.CreateCloudBlobClient();  
CloudBlobContainer contain = client.GetContainerReference(ConfigurationManager.AppSettings["container"]);  
contain.CreateIfNotExists();  
  
// The Resolver object is used to interact with Key Vault for Azure Storage.  
// This is where the GetToken method from above is used.  
KeyVaultKeyResolver cloudResolver = new KeyVaultKeyResolver(GetToken);
```

NOTE

Key Vault Object Models

It is important to understand that there are actually two Key Vault object models to be aware of: one is based on the REST API (KeyVault namespace) and the other is an extension for client-side encryption.

The Key Vault Client interacts with the REST API and understands JSON Web Keys and secrets for the two kinds of things that are contained in Key Vault.

The Key Vault Extensions are classes that seem specifically created for client-side encryption in Azure Storage. They contain an interface for keys (IKey) and classes based on the concept of a Key Resolver. There are two implementations of IKey that you need to know: RSAKey and SymmetricKey. Now they happen to coincide with the things that are contained in a Key Vault, but at this point they are independent classes (so the Key and Secret retrieved by the Key Vault Client do not implement IKey).

Encrypt blob and upload

Add the following code to encrypt a blob and upload it to your Azure storage account. The **ResolveKeyAsync** method that is used returns an IKey.

```
// Retrieve the key that you created previously.  
// The IKey that is returned here is an RsaKey.  
// Remember that we used the names contosokeyvault and testrsakey1.  
var rsa = cloudResolver.ResolveKeyAsync("https://contosokeyvault.vault.azure.net/keys/TestRSAKey1",  
CancellationToken.None).GetAwaiter().GetResult();  
  
// Now you simply use the RSA key to encrypt by setting it in the BlobEncryptionPolicy.  
BlobEncryptionPolicy policy = new BlobEncryptionPolicy(rsa, null);  
BlobRequestOptions options = new BlobRequestOptions() { EncryptionPolicy = policy };  
  
// Reference a block blob.  
CloudBlockBlob blob = contain.GetBlockBlobReference("MyFile.txt");  
  
// Upload using the UploadFromStream method.  
using (var stream = System.IO.File.OpenRead(@"C:\data\MyFile.txt"))  
    blob.UploadFromStream(stream, stream.Length, null, options, null);
```

Following is a screenshot from the [Azure Classic Portal](#) for a blob that has been encrypted by using client-side encryption with a key stored in Key Vault. The **KeyId** property is the URI for the key in Key Vault that acts as the KEK. The **EncryptedKey** property contains the encrypted version of the CEK.

The screenshot shows the Azure Storage Blob properties page for a file named 'MyFile'. The 'Edit blob properties and metadata' dialog is open. In the 'PROPERTIES' section, the 'EncryptionMode' is listed as 'FullBlob' and the 'WrappedContentKey' is associated with a specific Key ID. In the 'METADATA' section, there is a key-value pair for 'encryptiondata' which contains a JSON object representing the encryption policy.

PROPERTY	VALUE
Name	MyFile.txt
Absolute URI	https://[REDACTED].blob.core.windows.net/MyFile.txt
Blob Type	Block Blob
Cache Control	EMPTY
Content Encoding	EMPTY
Content Language	EMPTY
Content MD5	Z0RmifbSl4BuUPxkVleiiig==
Content Type	application/octet-stream
ETag	"0x8D27353AA4D8146"
Last Modified Time (UTC)	6/12/2015 2:20:58 PM
Lease Status	Unlocked
Size	15.86 KB

METADATA

METADATA	VALUE
encryptiondata	{"EncryptionMode": "FullBlob", "WrappedContentKey": {"KeyId": "https://[REDACTED].vault.azure.net/keys/7e9bdcb977746979f4038e6642a6f73", "EncryptedKey": "cmjwHnOn6nwQMsLzjHmCS5C7oxtU1fAAAtX8uAO3BbEbsHCms3s3DQ/b1xaEKJxdxpPCP0AjJBCWILAmCDuuFw3y8wbutBYfS4au4Dk1+Sx60IpFT2ANprlw4JLz1va3lQEgHvAqVQLU85bPdDpa19I6cDFOjbVSeoyj6C6yXBkZfu94yaUcdatfW02cgB5Jx/J/y/te8j6a5vyphToyCYEBSeM0FpUhL7YwC6Hq8TSapOITACfW94eGd1w==", "Algorithm": "RSA-OAEP"}, "EncryptionAgent": {"Protocol": "1.0", "EncryptionAlgorithm": "AES_CBC_256"}, "ContentEncryptor": "AES_CBC_256"}}

NOTE

If you look at the `BlobEncryptionPolicy` constructor, you will see that it can accept a key and/or a resolver. Be aware that right now you cannot use a resolver for encryption because it does not currently support a default key.

Decrypt blob and download

Decryption is really when using the Resolver classes make sense. The ID of the key used for encryption is associated with the blob in its metadata, so there is no reason for you to retrieve the key and remember the association between key and blob. You just have to make sure that the key remains in Key Vault.

The private key of an RSA Key remains in Key Vault, so for decryption to occur, the Encrypted Key from the blob metadata that contains the CEK is sent to Key Vault for decryption.

Add the following to decrypt the blob that you just uploaded.

```
// In this case, we will not pass a key and only pass the resolver because
// this policy will only be used for downloading / decrypting.
BlobEncryptionPolicy policy = new BlobEncryptionPolicy(null, cloudResolver);
BlobRequestOptions options = new BlobRequestOptions() { EncryptionPolicy = policy };

using (var np = File.Open(@"C:\data\MyFileDecrypted.txt", FileMode.Create))
    blob.DownloadToStream(np, null, options, null);
```

NOTE

There are a couple of other kinds of resolvers to make key management easier, including: AggregateKeyResolver and CachingKeyResolver.

Use Key Vault secrets

The way to use a secret with client-side encryption is via the SymmetricKey class because a secret is essentially a symmetric key. But, as noted above, a secret in Key Vault does not map exactly to a SymmetricKey. There are a few things to understand:

- The key in a SymmetricKey has to be a fixed length: 128, 192, 256, 384, or 512 bits.
- The key in a SymmetricKey should be Base64 encoded.
- A Key Vault secret that will be used as a SymmetricKey needs to have a Content Type of "application/octet-stream" in Key Vault.

Here is an example in PowerShell of creating a secret in Key Vault that can be used as a SymmetricKey. Please note that the hard coded value, \$key, is for demonstration purpose only. In your own code you'll want to generate this key.

```
// Here we are making a 128-bit key so we have 16 characters.  
//      The characters are in the ASCII range of UTF8 so they are  
//      each 1 byte. 16 x 8 = 128.  
$key = "qwertyuiopasdfgh"  
$b = [System.Text.Encoding]::UTF8.GetBytes($key)  
$enc = [System.Convert]::ToBase64String($b)  
$secretvalue = ConvertTo-SecureString $enc -AsPlainText -Force  
  
// Substitute the VaultName and Name in this command.  
$secret = Set-AzureKeyVaultSecret -VaultName 'ContosoKeyVault' -Name 'TestSecret2' -SecretValue $secretvalue -  
ContentType "application/octet-stream"
```

In your console application, you can use the same call as before to retrieve this secret as a SymmetricKey.

```
SymmetricKey sec = (SymmetricKey) cloudResolver.ResolveKeyAsync(  
    "https://contosokeyvault.vault.azure.net/secrets/TestSecret2/",  
    CancellationToken.None).GetAwaiter().GetResult();
```

That's it. Enjoy!

Next steps

For more information about using Microsoft Azure Storage with C#, see [Microsoft Azure Storage Client Library for .NET](#).

For more information about the Blob REST API, see [Blob Service REST API](#).

For the latest information on Microsoft Azure Storage, go to the [Microsoft Azure Storage Team Blog](#).

Client-Side Encryption and Azure Key Vault for Microsoft Azure Storage

1/17/2017 • 13 min to read • [Edit on GitHub](#)

Overview

The [Azure Storage Client Library for .NET Nuget package](#) supports encrypting data within client applications before uploading to Azure Storage, and decrypting data while downloading to the client. The library also supports integration with [Azure Key Vault](#) for storage account key management.

For a step-by-step tutorial that leads you through the process of encrypting blobs using client-side encryption and Azure Key Vault, see [Encrypt and decrypt blobs in Microsoft Azure Storage using Azure Key Vault](#).

For client-side encryption with Java, see [Client-Side Encryption with Java for Microsoft Azure Storage](#).

Encryption and decryption via the envelope technique

The processes of encryption and decryption follow the envelope technique.

Encryption via the envelope technique

Encryption via the envelope technique works in the following way:

1. The Azure storage client library generates a content encryption key (CEK), which is a one-time-use symmetric key.
2. User data is encrypted using this CEK.
3. The CEK is then wrapped (encrypted) using the key encryption key (KEK). The KEK is identified by a key identifier and can be an asymmetric key pair or a symmetric key and can be managed locally or stored in Azure Key Vaults.

The storage client library itself never has access to KEK. The library invokes the key wrapping algorithm that is provided by Key Vault. Users can choose to use custom providers for key wrapping/unwrapping if desired.

4. The encrypted data is then uploaded to the Azure Storage service. The wrapped key along with some additional encryption metadata is either stored as metadata (on a blob) or interpolated with the encrypted data (queue messages and table entities).

Decryption via the envelope technique

Decryption via the envelope technique works in the following way:

1. The client library assumes that the user is managing the key encryption key (KEK) either locally or in Azure Key Vaults. The user does not need to know the specific key that was used for encryption. Instead, a key resolver which resolves different key identifiers to keys can be set up and used.
2. The client library downloads the encrypted data along with any encryption material that is stored on the service.
3. The wrapped content encryption key (CEK) is then unwrapped (decrypted) using the key encryption key (KEK). Here again, the client library does not have access to KEK. It simply invokes the custom or Key Vault provider's unwrapping algorithm.
4. The content encryption key (CEK) is then used to decrypt the encrypted user data.

Encryption Mechanism

The storage client library uses [AES](#) in order to encrypt user data. Specifically, [Cipher Block Chaining \(CBC\)](#) mode with AES. Each service works somewhat differently, so we will discuss each of them here.

Blobs

The client library currently supports encryption of whole blobs only. Specifically, encryption is supported when users use the **UploadFrom*** methods or the **OpenWrite** method. For downloads, both complete and range downloads are supported.

During encryption, the client library will generate a random Initialization Vector (IV) of 16 bytes, together with a random content encryption key (CEK) of 32 bytes, and perform envelope encryption of the blob data using this information. The wrapped CEK and some additional encryption metadata are then stored as blob metadata along with the encrypted blob on the service.

WARNING

If you are editing or uploading your own metadata for the blob, you need to ensure that this metadata is preserved. If you upload new metadata without this metadata, the wrapped CEK, IV, and other metadata will be lost and the blob content will never be retrievable again.

Downloading an encrypted blob involves retrieving the content of the entire blob using the **DownloadTo/BlobReadStream*** convenience methods. The wrapped CEK is unwrapped and used together with the IV (stored as blob metadata in this case) to return the decrypted data to the users.

Downloading an arbitrary range (**DownloadRange*** methods) in the encrypted blob involves adjusting the range provided by users in order to get a small amount of additional data that can be used to successfully decrypt the requested range.

All blob types (block blobs, page blobs, and append blobs) can be encrypted/decrypted using this scheme.

Queues

Since queue messages can be of any format, the client library defines a custom format that includes the Initialization Vector (IV) and the encrypted content encryption key (CEK) in the message text.

During encryption, the client library generates a random IV of 16 bytes along with a random CEK of 32 bytes and performs envelope encryption of the queue message text using this information. The wrapped CEK and some additional encryption metadata are then added to the encrypted queue message. This modified message (shown below) is stored on the service.

```
<MessageText>
{"EncryptedMessageContents": "6k0u8Rq1C3+M1Q04a1KLmWthWXSmHV3mEfxBAgP9QGTU++MKn2uPq3t2UjF1D06w", "EncryptionData"
": {...}}</MessageText>
```

During decryption, the wrapped key is extracted from the queue message and unwrapped. The IV is also extracted from the queue message and used along with the unwrapped key to decrypt the queue message data. Note that the encryption metadata is small (under 500 bytes), so while it does count toward the 64KB limit for a queue message, the impact should be manageable.

Tables

The client library supports encryption of entity properties for insert and replace operations.

NOTE

Merge is not currently supported. Since a subset of properties may have been encrypted previously using a different key, simply merging the new properties and updating the metadata will result in data loss. Merging either requires making extra service calls to read the pre-existing entity from the service, or using a new key per property, both of which are not suitable for performance reasons.

Table data encryption works as follows:

1. Users specify the properties to be encrypted.
2. The client library generates a random Initialization Vector (IV) of 16 bytes along with a random content encryption key (CEK) of 32 bytes for every entity, and performs envelope encryption on the individual properties to be encrypted by deriving a new IV per property. The encrypted property is stored as binary data.
3. The wrapped CEK and some additional encryption metadata are then stored as two additional reserved properties. The first reserved property (`_ClientEncryptionMetadata1`) is a string property that holds the information about IV, version, and wrapped key. The second reserved property (`_ClientEncryptionMetadata2`) is a binary property that holds the information about the properties that are encrypted. The information in this second property (`_ClientEncryptionMetadata2`) is itself encrypted.
4. Due to these additional reserved properties required for encryption, users may now have only 250 custom properties instead of 252. The total size of the entity must be less than 1 MB.

Note that only string properties can be encrypted. If other types of properties are to be encrypted, they must be converted to strings. The encrypted strings are stored on the service as binary properties, and they are converted back to strings after decryption.

For tables, in addition to the encryption policy, users must specify the properties to be encrypted. This can be done by either specifying an `[EncryptProperty]` attribute (for POCO entities that derive from `TableEntity`) or an encryption resolver in request options. An encryption resolver is a delegate that takes a partition key, row key, and property name and returns a Boolean that indicates whether that property should be encrypted. During encryption, the client library will use this information to decide whether a property should be encrypted while writing to the wire. The delegate also provides for the possibility of logic around how properties are encrypted. (For example, if X, then encrypt property A; otherwise encrypt properties A and B.) Note that it is not necessary to provide this information while reading or querying entities.

Batch Operations

In batch operations, the same KEK will be used across all the rows in that batch operation because the client library only allows one options object (and hence one policy/KEK) per batch operation. However, the client library will internally generate a new random IV and random CEK per row in the batch. Users can also choose to encrypt different properties for every operation in the batch by defining this behavior in the encryption resolver.

Queries

To perform query operations, you must specify a key resolver that is able to resolve all the keys in the result set. If an entity contained in the query result cannot be resolved to a provider, the client library will throw an error. For any query that performs server-side projections, the client library will add the special encryption metadata properties (`_ClientEncryptionMetadata1` and `_ClientEncryptionMetadata2`) by default to the selected columns.

Azure Key Vault

Azure Key Vault helps safeguard cryptographic keys and secrets used by cloud applications and services. By using Azure Key Vault, users can encrypt keys and secrets (such as authentication keys, storage account keys, data encryption keys, .PFX files, and passwords) by using keys that are protected by hardware security modules (HSMs). For more information, see [What is Azure Key Vault?](#).

The storage client library uses the Key Vault core library in order to provide a common framework across Azure

for managing keys. Users also get the additional benefit of using the Key Vault extensions library. The extensions library provides useful functionality around simple and seamless Symmetric/RSA local and cloud key providers as well as with aggregation and caching.

Interface and dependencies

There are three Key Vault packages:

- Microsoft.Azure.KeyVault.Core contains the IKey and IKeyResolver. It is a small package with no dependencies. The storage client library for .NET defines it as a dependency.
- Microsoft.Azure.KeyVault contains the Key Vault REST client.
- Microsoft.Azure.KeyVault.Extensions contains extension code that includes implementations of cryptographic algorithms and an RSAKey and a SymmetricKey. It depends on the Core and KeyVault namespaces and provides functionality to define an aggregate resolver (when users want to use multiple key providers) and a caching key resolver. Although the storage client library does not directly depend on this package, if users wish to use Azure Key Vault to store their keys or to use the Key Vault extensions to consume the local and cloud cryptographic providers, they will need this package.

Key Vault is designed for high-value master keys, and throttling limits per Key Vault are designed with this in mind. When performing client-side encryption with Key Vault, the preferred model is to use symmetric master keys stored as secrets in Key Vault and cached locally. Users must do the following:

1. Create a secret offline and upload it to Key Vault.
2. Use the secret's base identifier as a parameter to resolve the current version of the secret for encryption and cache this information locally. Use CachingKeyResolver for caching; users are not expected to implement their own caching logic.
3. Use the caching resolver as an input while creating the encryption policy.

More information regarding Key Vault usage can be found in the [encryption code samples](#).

Best practices

Encryption support is available only in the storage client library for .NET. Windows Phone and Windows Runtime do not currently support encryption.

IMPORTANT

Be aware of these important points when using client-side encryption:

- When reading from or writing to an encrypted blob, use whole blob upload commands and range/whole blob download commands. Avoid writing to an encrypted blob using protocol operations such as Put Block, Put Block List, Write Pages, Clear Pages, or Append Block; otherwise you may corrupt the encrypted blob and make it unreadable.
- For tables, a similar constraint exists. Be careful to not update encrypted properties without updating the encryption metadata.
- If you set metadata on the encrypted blob, you may overwrite the encryption-related metadata required for decryption, since setting metadata is not additive. This is also true for snapshots; avoid specifying metadata while creating a snapshot of an encrypted blob. If metadata must be set, be sure to call the **FetchAttributes** method first to get the current encryption metadata, and avoid concurrent writes while metadata is being set.
- Enable the **RequireEncryption** property in the default request options for users that should work only with encrypted data. See below for more info.

Client API / Interface

While creating an EncryptionPolicy object, users can provide only a Key (implementing IKey), only a resolver (implementing IKeyResolver), or both. IKey is the basic key type that is identified using a key identifier and that

provides the logic for wrapping/unwrapping. IKeyResolver is used to resolve a key during the decryption process. It defines a ResolveKey method that returns an IKey given a key identifier. This provides users the ability to choose between multiple keys that are managed in multiple locations.

- For encryption, the key is used always and the absence of a key will result in an error.
- For decryption:
 - The key resolver is invoked if specified to get the key. If the resolver is specified but does not have a mapping for the key identifier, an error is thrown.
 - If resolver is not specified but a key is specified, the key is used if its identifier matches the required key identifier. If the identifier does not match, an error is thrown.

The [encryption samples](#) demonstrate a more detailed end-to-end scenario for blobs, queues and tables, along with Key Vault integration.

RequireEncryption mode

Users can optionally enable a mode of operation where all uploads and downloads must be encrypted. In this mode, attempts to upload data without an encryption policy or download data that is not encrypted on the service will fail on the client. The **RequireEncryption** property of the request options object controls this behavior. If your application will encrypt all objects stored in Azure Storage, then you can set the **RequireEncryption** property on the default request options for the service client object. For example, set **CloudBlobClient.DefaultRequestOptions.RequireEncryption** to **true** to require encryption for all blob operations performed through that client object.

Blob service encryption

Create a **BlobEncryptionPolicy** object and set it in the request options (per API or at a client level by using **DefaultRequestOptions**). Everything else will be handled by the client library internally.

```
// Create the IKey used for encryption.  
RsaKey key = new RsaKey("private:key1" /* key identifier */);  
  
// Create the encryption policy to be used for upload and download.  
BlobEncryptionPolicy policy = new BlobEncryptionPolicy(key, null);  
  
// Set the encryption policy on the request options.  
BlobRequestOptions options = new BlobRequestOptions() { EncryptionPolicy = policy };  
  
// Upload the encrypted contents to the blob.  
blob.UploadFromStream(stream, size, null, options, null);  
  
// Download and decrypt the encrypted contents from the blob.  
MemoryStream outputStream = new MemoryStream();  
blob.DownloadToStream(outputStream, null, options, null);
```

Queue service encryption

Create a **QueueEncryptionPolicy** object and set it in the request options (per API or at a client level by using **DefaultRequestOptions**). Everything else will be handled by the client library internally.

```

// Create the IKey used for encryption.
RsaKey key = new RsaKey("private:key1" /* key identifier */);

// Create the encryption policy to be used for upload and download.
QueueEncryptionPolicy policy = new QueueEncryptionPolicy(key, null);

// Add message
QueueRequestOptions options = new QueueRequestOptions() { EncryptionPolicy = policy };
queue.AddMessage(message, null, null, options, null);

// Retrieve message
CloudQueueMessage retrMessage = queue.GetMessage(null, options, null);

```

Table service encryption

In addition to creating an encryption policy and setting it on request options, you must either specify an **EncryptionResolver** in **TableRequestOptions**, or set the [EncryptProperty] attribute on the entity.

Using the resolver

```

// Create the IKey used for encryption.
RsaKey key = new RsaKey("private:key1" /* key identifier */);

// Create the encryption policy to be used for upload and download.
TableEncryptionPolicy policy = new TableEncryptionPolicy(key, null);

TableRequestOptions options = new TableRequestOptions()
{
    EncryptionResolver = (pk, rk, propName) =>
    {
        if (propName == "foo")
        {
            return true;
        }
        return false;
    },
    EncryptionPolicy = policy
};

// Insert Entity
currentTable.Execute(TableOperation.Insert(ent), options, null);

// Retrieve Entity
// No need to specify an encryption resolver for retrieve
TableRequestOptions retrieveOptions = new TableRequestOptions()
{
    EncryptionPolicy = policy
};

TableOperation operation = TableOperation.Retrieve(ent.PartitionKey, ent.RowKey);
TableResult result = currentTable.Execute(operation, retrieveOptions, null);

```

Using attributes

As mentioned above, if the entity implements **TableEntity**, then the properties can be decorated with the [EncryptProperty] attribute instead of specifying the **EncryptionResolver**.

```

[EncryptProperty]
public string EncryptedProperty1 { get; set; }

```

Encryption and performance

Note that encrypting your storage data results in additional performance overhead. The content key and IV must be generated, the content itself must be encrypted, and additional meta-data must be formatted and uploaded.

This overhead will vary depending on the quantity of data being encrypted. We recommend that customers always test their applications for performance during development.

Next steps

- [Tutorial: Encrypt and decrypt blobs in Microsoft Azure Storage using Azure Key Vault](#)
- Download the [Azure Storage Client Library for .NET NuGet package](#)
- Download the Azure Key Vault NuGet [Core](#), [Client](#), and [Extensions](#) packages
- Visit the [Azure Key Vault Documentation](#)

Client-Side Encryption and Azure Key Vault with Java for Microsoft Azure Storage

1/17/2017 • 13 min to read • [Edit on GitHub](#)

Overview

The [Azure Storage Client Library for Java](#) supports encrypting data within client applications before uploading to Azure Storage, and decrypting data while downloading to the client. The library also supports integration with [Azure Key Vault](#) for storage account key management.

Encryption and decryption via the envelope technique

The processes of encryption and decryption follow the envelope technique.

Encryption via the envelope technique

Encryption via the envelope technique works in the following way:

1. The Azure storage client library generates a content encryption key (CEK), which is a one-time-use symmetric key.
2. User data is encrypted using this CEK.
3. The CEK is then wrapped (encrypted) using the key encryption key (KEK). The KEK is identified by a key identifier and can be an asymmetric key pair or a symmetric key and can be managed locally or stored in Azure Key Vaults.
The storage client library itself never has access to KEK. The library invokes the key wrapping algorithm that is provided by Key Vault. Users can choose to use custom providers for key wrapping/unwrapping if desired.
4. The encrypted data is then uploaded to the Azure Storage service. The wrapped key along with some additional encryption metadata is either stored as metadata (on a blob) or interpolated with the encrypted data (queue messages and table entities).

Decryption via the envelope technique

Decryption via the envelope technique works in the following way:

1. The client library assumes that the user is managing the key encryption key (KEK) either locally or in Azure Key Vaults. The user does not need to know the specific key that was used for encryption. Instead, a key resolver which resolves different key identifiers to keys can be set up and used.
2. The client library downloads the encrypted data along with any encryption material that is stored on the service.
3. The wrapped content encryption key (CEK) is then unwrapped (decrypted) using the key encryption key (KEK). Here again, the client library does not have access to KEK. It simply invokes the custom or Key Vault provider's unwrapping algorithm.
4. The content encryption key (CEK) is then used to decrypt the encrypted user data.

Encryption Mechanism

The storage client library uses [AES](#) in order to encrypt user data. Specifically, [Cipher Block Chaining \(CBC\)](#) mode with AES. Each service works somewhat differently, so we will discuss each of them here.

Blobs

The client library currently supports encryption of whole blobs only. Specifically, encryption is supported when users use the **upload*** methods or the **openOutputStream** method. For downloads, both complete and range

downloads are supported.

During encryption, the client library will generate a random Initialization Vector (IV) of 16 bytes, together with a random content encryption key (CEK) of 32 bytes, and perform envelope encryption of the blob data using this information. The wrapped CEK and some additional encryption metadata are then stored as blob metadata along with the encrypted blob on the service.

WARNING

If you are editing or uploading your own metadata for the blob, you need to ensure that this metadata is preserved. If you upload new metadata without this metadata, the wrapped CEK, IV and other metadata will be lost and the blob content will never be retrievable again.

Downloading an encrypted blob involves retrieving the content of the entire blob using the **download*/openInputStream** convenience methods. The wrapped CEK is unwrapped and used together with the IV (stored as blob metadata in this case) to return the decrypted data to the users.

Downloading an arbitrary range (**downloadRange*** methods) in the encrypted blob involves adjusting the range provided by users in order to get a small amount of additional data that can be used to successfully decrypt the requested range.

All blob types (block blobs, page blobs, and append blobs) can be encrypted/decrypted using this scheme.

Queues

Since queue messages can be of any format, the client library defines a custom format that includes the Initialization Vector (IV) and the encrypted content encryption key (CEK) in the message text.

During encryption, the client library generates a random IV of 16 bytes along with a random CEK of 32 bytes and performs envelope encryption of the queue message text using this information. The wrapped CEK and some additional encryption metadata are then added to the encrypted queue message. This modified message (shown below) is stored on the service.

```
<MessageText>
{"EncryptedMessageContents": "6kOu8Rq1C3+M1Q04a1KLmWthWXSmHV3mEfxbAgP9QGTU++MKn2uPq3t2UjF1D06w", "EncryptionData": ...
}</MessageText>
```

During decryption, the wrapped key is extracted from the queue message and unwrapped. The IV is also extracted from the queue message and used along with the unwrapped key to decrypt the queue message data. Note that the encryption metadata is small (under 500 bytes), so while it does count toward the 64KB limit for a queue message, the impact should be manageable.

Tables

The client library supports encryption of entity properties for insert and replace operations.

NOTE

Merge is not currently supported. Since a subset of properties may have been encrypted previously using a different key, simply merging the new properties and updating the metadata will result in data loss. Merging either requires making extra service calls to read the pre-existing entity from the service, or using a new key per property, both of which are not suitable for performance reasons.

Table data encryption works as follows:

1. Users specify the properties to be encrypted.
2. The client library generates a random Initialization Vector (IV) of 16 bytes along with a random content

encryption key (CEK) of 32 bytes for every entity, and performs envelope encryption on the individual properties to be encrypted by deriving a new IV per property. The encrypted property is stored as binary data.

3. The wrapped CEK and some additional encryption metadata are then stored as two additional reserved properties. The first reserved property (`_ClientEncryptionMetadata1`) is a string property that holds the information about IV, version, and wrapped key. The second reserved property (`_ClientEncryptionMetadata2`) is a binary property that holds the information about the properties that are encrypted. The information in this second property (`_ClientEncryptionMetadata2`) is itself encrypted.
4. Due to these additional reserved properties required for encryption, users may now have only 250 custom properties instead of 252. The total size of the entity must be less than 1MB.

Note that only string properties can be encrypted. If other types of properties are to be encrypted, they must be converted to strings. The encrypted strings are stored on the service as binary properties, and they are converted back to strings after decryption.

For tables, in addition to the encryption policy, users must specify the properties to be encrypted. This can be done by either specifying an [Encrypt] attribute (for POCO entities that derive from `TableEntity`) or an encryption resolver in request options. An encryption resolver is a delegate that takes a partition key, row key, and property name and returns a boolean that indicates whether that property should be encrypted. During encryption, the client library will use this information to decide whether a property should be encrypted while writing to the wire. The delegate also provides for the possibility of logic around how properties are encrypted. (For example, if X, then encrypt property A; otherwise encrypt properties A and B.) Note that it is not necessary to provide this information while reading or querying entities.

Batch Operations

In batch operations, the same KEK will be used across all the rows in that batch operation because the client library only allows one options object (and hence one policy/KEK) per batch operation. However, the client library will internally generate a new random IV and random CEK per row in the batch. Users can also choose to encrypt different properties for every operation in the batch by defining this behavior in the encryption resolver.

Queries

To perform query operations, you must specify a key resolver that is able to resolve all the keys in the result set. If an entity contained in the query result cannot be resolved to a provider, the client library will throw an error. For any query that performs server side projections, the client library will add the special encryption metadata properties (`_ClientEncryptionMetadata1` and `_ClientEncryptionMetadata2`) by default to the selected columns.

Azure Key Vault

Azure Key Vault helps safeguard cryptographic keys and secrets used by cloud applications and services. By using Azure Key Vault, users can encrypt keys and secrets (such as authentication keys, storage account keys, data encryption keys, .PFX files, and passwords) by using keys that are protected by hardware security modules (HSMs). For more information, see [What is Azure Key Vault?](#).

The storage client library uses the Key Vault core library in order to provide a common framework across Azure for managing keys. Users also get the additional benefit of using the Key Vault extensions library. The extensions library provides useful functionality around simple and seamless Symmetric/RSA local and cloud key providers as well as with aggregation and caching.

Interface and dependencies

There are three Key Vault packages:

- `azure-keyvault-core` contains the `IKey` and `IKeyResolver`. It is a small package with no dependencies. The storage client library for Java defines it as a dependency.
- `azure-keyvault` contains the Key Vault REST client.
- `azure-keyvault-extensions` contains extension code that includes implementations of cryptographic

algorithms and an RSAKey and a SymmetricKey. It depends on the Core and KeyVault namespaces and provides functionality to define an aggregate resolver (when users want to use multiple key providers) and a caching key resolver. Although the storage client library does not directly depend on this package, if users wish to use Azure Key Vault to store their keys or to use the Key Vault extensions to consume the local and cloud cryptographic providers, they will need this package.

Key Vault is designed for high-value master keys, and throttling limits per Key Vault are designed with this in mind. When performing client-side encryption with Key Vault, the preferred model is to use symmetric master keys stored as secrets in Key Vault and cached locally. Users must do the following:

1. Create a secret offline and upload it to Key Vault.
2. Use the secret's base identifier as a parameter to resolve the current version of the secret for encryption and cache this information locally. Use CachingKeyResolver for caching; users are not expected to implement their own caching logic.
3. Use the caching resolver as an input while creating the encryption policy. More information regarding Key Vault usage can be found in the encryption code samples.

Best practices

Encryption support is available only in the storage client library for Java.

IMPORTANT

Be aware of these important points when using client-side encryption:

- When reading from or writing to an encrypted blob, use whole blob upload commands and range/whole blob download commands. Avoid writing to an encrypted blob using protocol operations such as Put Block, Put Block List, Write Pages, Clear Pages, or Append Block; otherwise you may corrupt the encrypted blob and make it unreadable.
- For tables, a similar constraint exists. Be careful to not update encrypted properties without updating the encryption metadata.
- If you set metadata on the encrypted blob, you may overwrite the encryption-related metadata required for decryption, since setting metadata is not additive. This is also true for snapshots; avoid specifying metadata while creating a snapshot of an encrypted blob. If metadata must be set, be sure to call the **downloadAttributes** method first to get the current encryption metadata, and avoid concurrent writes while metadata is being set.
- Enable the **requireEncryption** flag in the default request options for users that should work only with encrypted data. See below for more info.

Client API / Interface

While creating an EncryptionPolicy object, users can provide only a Key (implementing IKey), only a resolver (implementing IKeyResolver), or both. IKey is the basic key type that is identified using a key identifier and that provides the logic for wrapping/unwrapping. IKeyResolver is used to resolve a key during the decryption process. It defines a ResolveKey method that returns an IKey given a key identifier. This provides users the ability to choose between multiple keys that are managed in multiple locations.

- For encryption, the key is used always and the absence of a key will result in an error.
- For decryption:
 - The key resolver is invoked if specified to get the key. If the resolver is specified but does not have a mapping for the key identifier, an error is thrown.
 - If resolver is not specified but a key is specified, the key is used if its identifier matches the required key identifier. If the identifier does not match, an error is thrown.

The [encryption samples](#) demonstrate a more detailed end-to-end scenario for blobs, queues and tables, along with Key Vault integration.

RequireEncryption mode

Users can optionally enable a mode of operation where all uploads and downloads must be encrypted. In this mode, attempts to upload data without an encryption policy or download data that is not encrypted on the service will fail on the client. The **requireEncryption** flag of the request options object controls this behavior. If your application will encrypt all objects stored in Azure Storage, then you can set the **requireEncryption** property on the default request options for the service client object.

For example, use **CloudBlobClient.getDefaultRequestOptions().setRequireEncryption(true)** to require encryption for all blob operations performed through that client object.

Blob service encryption

Create a **BlobEncryptionPolicy** object and set it in the request options (per API or at a client level by using **DefaultRequestOptions**). Everything else will be handled by the client library internally.

```
// Create the IKey used for encryption.  
RsaKey key = new RsaKey("private:key1" /* key identifier */);  
  
// Create the encryption policy to be used for upload and download.  
BlobEncryptionPolicy policy = new BlobEncryptionPolicy(key, null);  
  
// Set the encryption policy on the request options.  
BlobRequestOptions options = new BlobRequestOptions();  
options.setEncryptionPolicy(policy);  
  
// Upload the encrypted contents to the blob.  
blob.upload(stream, size, null, options, null);  
  
// Download and decrypt the encrypted contents from the blob.  
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();  
blob.download(outputStream, null, options, null);
```

Queue service encryption

Create a **QueueEncryptionPolicy** object and set it in the request options (per API or at a client level by using **DefaultRequestOptions**). Everything else will be handled by the client library internally.

```
// Create the IKey used for encryption.  
RsaKey key = new RsaKey("private:key1" /* key identifier */);  
  
// Create the encryption policy to be used for upload and download.  
QueueEncryptionPolicy policy = new QueueEncryptionPolicy(key, null);  
  
// Add message  
QueueRequestOptions options = new QueueRequestOptions();  
options.setEncryptionPolicy(policy);  
  
queue.addMessage(message, 0, 0, options, null);  
  
// Retrieve message  
CloudQueueMessage retrMessage = queue.retrieveMessage(30, options, null);
```

Table service encryption

In addition to creating an encryption policy and setting it on request options, you must either specify an **EncryptionResolver** in **TableRequestOptions**, or set the [Encrypt] attribute on the entity's getter and setter.

Using the resolver

```

// Create the IKey used for encryption.
RsaKey key = new RsaKey("private:key1" /* key identifier */);

// Create the encryption policy to be used for upload and download.
TableEncryptionPolicy policy = new TableEncryptionPolicy(key, null);

TableRequestOptions options = new TableRequestOptions()
options.setEncryptionPolicy(policy);
options.setEncryptionResolver(new EncryptionResolver() {
    public boolean encryptionResolver(String pk, String rk, String key) {
        if (key == "foo")
        {
            return true;
        }
        return false;
    }
});

// Insert Entity
currentTable.execute(TableOperation.insert(ent), options, null);

// Retrieve Entity
// No need to specify an encryption resolver for retrieve
TableRequestOptions retrieveOptions = new TableRequestOptions()
retrieveOptions.setEncryptionPolicy(policy);

TableOperation operation = TableOperation.retrieve(ent.PartitionKey, ent.RowKey, DynamicTableEntity.class);
TableResult result = currentTable.execute(operation, retrieveOptions, null);

```

Using attributes

As mentioned above, if the entity implements TableEntity, then the properties getter and setter can be decorated with the [Encrypt] attribute instead of specifying the **EncryptionResolver**.

```

private string encryptedProperty1;

@Encrypt
public String getEncryptedProperty1 () {
    return this.encryptedProperty1;
}

@Encrypt
public void setEncryptedProperty1(final String encryptedProperty1) {
    this.encryptedProperty1 = encryptedProperty1;
}

```

Encryption and performance

Note that encrypting your storage data results in additional performance overhead. The content key and IV must be generated, the content itself must be encrypted, and additional meta-data must be formatted and uploaded. This overhead will vary depending on the quantity of data being encrypted. We recommend that customers always test their applications for performance during development.

Next steps

- Download the [Azure Storage Client Library for Java Maven package](#)
- Download the [Azure Storage Client Library for Java Source Code from GitHub](#)
- Download the Azure Key Vault Maven Library for Java Maven packages:
 - [Core](#) package
 - [Client](#) package

- Visit the [Azure Key Vault Documentation](#)

Client-Side Encryption with Python for Microsoft Azure Storage

1/17/2017 • 12 min to read • [Edit on GitHub](#)

Overview

The [Azure Storage Client Library for Python](#) supports encrypting data within client applications before uploading to Azure Storage, and decrypting data while downloading to the client.

NOTE

The Azure Storage Python library is in preview.

Encryption and decryption via the envelope technique

The processes of encryption and decryption follow the envelope technique.

Encryption via the envelope technique

Encryption via the envelope technique works in the following way:

1. The Azure storage client library generates a content encryption key (CEK), which is a one-time-use symmetric key.
2. User data is encrypted using this CEK.
3. The CEK is then wrapped (encrypted) using the key encryption key (KEK). The KEK is identified by a key identifier and can be an asymmetric key pair or a symmetric key, which is managed locally. The storage client library itself never has access to KEK. The library invokes the key wrapping algorithm that is provided by the KEK. Users can choose to use custom providers for key wrapping/unwrapping if desired.
4. The encrypted data is then uploaded to the Azure Storage service. The wrapped key along with some additional encryption metadata is either stored as metadata (on a blob) or interpolated with the encrypted data (queue messages and table entities).

Decryption via the envelope technique

Decryption via the envelope technique works in the following way:

1. The client library assumes that the user is managing the key encryption key (KEK) locally. The user does not need to know the specific key that was used for encryption. Instead, a key resolver, which resolves different key identifiers to keys, can be set up and used.
2. The client library downloads the encrypted data along with any encryption material that is stored on the service.
3. The wrapped content encryption key (CEK) is then unwrapped (decrypted) using the key encryption key (KEK). Here again, the client library does not have access to KEK. It simply invokes the custom provider's unwrapping algorithm.
4. The content encryption key (CEK) is then used to decrypt the encrypted user data.

Encryption Mechanism

The storage client library uses [AES](#) in order to encrypt user data. Specifically, [Cipher Block Chaining \(CBC\)](#) mode with AES. Each service works somewhat differently, so we will discuss each of them here.

Blobs

The client library currently supports encryption of whole blobs only. Specifically, encryption is supported when users use the **create*** methods. For downloads, both complete and range downloads are supported, and parallelization of both upload and download is available.

During encryption, the client library will generate a random Initialization Vector (IV) of 16 bytes, together with a random content encryption key (CEK) of 32 bytes, and perform envelope encryption of the blob data using this information. The wrapped CEK and some additional encryption metadata are then stored as blob metadata along with the encrypted blob on the service.

WARNING

If you are editing or uploading your own metadata for the blob, you need to ensure that this metadata is preserved. If you upload new metadata without this metadata, the wrapped CEK, IV and other metadata will be lost and the blob content will never be retrievable again.

Downloading an encrypted blob involves retrieving the content of the entire blob using the **get*** convenience methods. The wrapped CEK is unwrapped and used together with the IV (stored as blob metadata in this case) to return the decrypted data to the users.

Downloading an arbitrary range (**get*** methods with range parameters passed in) in the encrypted blob involves adjusting the range provided by users in order to get a small amount of additional data that can be used to successfully decrypt the requested range.

Block blobs and page blobs only can be encrypted/decrypted using this scheme. There is currently no support for encrypting append blobs.

Queues

Since queue messages can be of any format, the client library defines a custom format that includes the Initialization Vector (IV) and the encrypted content encryption key (CEK) in the message text.

During encryption, the client library generates a random IV of 16 bytes along with a random CEK of 32 bytes and performs envelope encryption of the queue message text using this information. The wrapped CEK and some additional encryption metadata are then added to the encrypted queue message. This modified message (shown below) is stored on the service.

```
<MessageText>
{"EncryptedMessageContents": "6k0u8Rq1C3+M1Q04a1KLmWthWXSmHV3mEfxbAgP9QGTU++MKn2uPq3t2UjF1D06w", "EncryptionData":
{...}}</MessageText>
```

During decryption, the wrapped key is extracted from the queue message and unwrapped. The IV is also extracted from the queue message and used along with the unwrapped key to decrypt the queue message data. Note that the encryption metadata is small (under 500 bytes), so while it does count toward the 64KB limit for a queue message, the impact should be manageable.

Tables

The client library supports encryption of entity properties for insert and replace operations.

NOTE

Merge is not currently supported. Since a subset of properties may have been encrypted previously using a different key, simply merging the new properties and updating the metadata will result in data loss. Merging either requires making extra service calls to read the pre-existing entity from the service, or using a new key per property, both of which are not suitable for performance reasons.

Table data encryption works as follows:

1. Users specify the properties to be encrypted.
2. The client library generates a random Initialization Vector (IV) of 16 bytes along with a random content encryption key (CEK) of 32 bytes for every entity, and performs envelope encryption on the individual properties to be encrypted by deriving a new IV per property. The encrypted property is stored as binary data.
3. The wrapped CEK and some additional encryption metadata are then stored as two additional reserved properties. The first reserved property (`_ClientEncryptionMetadata1`) is a string property that holds the information about IV, version, and wrapped key. The second reserved property (`_ClientEncryptionMetadata2`) is a binary property that holds the information about the properties that are encrypted. The information in this second property (`_ClientEncryptionMetadata2`) is itself encrypted.
4. Due to these additional reserved properties required for encryption, users may now have only 250 custom properties instead of 252. The total size of the entity must be less than 1MB.

Note that only string properties can be encrypted. If other types of properties are to be encrypted, they must be converted to strings. The encrypted strings are stored on the service as binary properties, and they are converted back to strings (raw strings, not EntityProperties with type `EdmType.STRING`) after decryption.

For tables, in addition to the encryption policy, users must specify the properties to be encrypted. This can be done by either storing these properties in `TableEntity` objects with the type set to `EdmType.STRING` and `encrypt` set to true or setting the `encryption_resolver_function` on the `tableservice` object. An encryption resolver is a function that takes a partition key, row key, and property name and returns a boolean that indicates whether that property should be encrypted. During encryption, the client library will use this information to decide whether a property should be encrypted while writing to the wire. The delegate also provides for the possibility of logic around how properties are encrypted. (For example, if X, then encrypt property A; otherwise encrypt properties A and B.) Note that it is not necessary to provide this information while reading or querying entities.

Batch Operations

One encryption policy applies to all rows in the batch. The client library will internally generate a new random IV and random CEK per row in the batch. Users can also choose to encrypt different properties for every operation in the batch by defining this behavior in the encryption resolver. If a batch is created as a context manager through the `tableservice batch()` method, the `tableservice`'s encryption policy will automatically be applied to the batch. If a batch is created explicitly by calling the constructor, the encryption policy must be passed as a parameter and left unmodified for the lifetime of the batch. Note that entities are encrypted as they are inserted into the batch using the batch's encryption policy (entities are NOT encrypted at the time of committing the batch using the `tableservice`'s encryption policy).

Queries

To perform query operations, you must specify a key resolver that is able to resolve all the keys in the result set. If an entity contained in the query result cannot be resolved to a provider, the client library will throw an error. For any query that performs server side projections, the client library will add the special encryption metadata properties (`_ClientEncryptionMetadata1` and `_ClientEncryptionMetadata2`) by default to the selected columns.

IMPORTANT

Be aware of these important points when using client-side encryption:

- When reading from or writing to an encrypted blob, use whole blob upload commands and range/whole blob download commands. Avoid writing to an encrypted blob using protocol operations such as Put Block, Put Block List, Write Pages, or Clear Pages; otherwise you may corrupt the encrypted blob and make it unreadable.
- For tables, a similar constraint exists. Be careful to not update encrypted properties without updating the encryption metadata.
- If you set metadata on the encrypted blob, you may overwrite the encryption-related metadata required for decryption, since setting metadata is not additive. This is also true for snapshots; avoid specifying metadata while creating a snapshot of an encrypted blob. If metadata must be set, be sure to call the **get_blob_metadata** method first to get the current encryption metadata, and avoid concurrent writes while metadata is being set.
- Enable the **require_encryption** flag on the service object for users that should work only with encrypted data. See below for more info.

The storage client library expects the provided KEK and key resolver to implement the following interface. [Azure Key Vault](#) support for Python KEK management is pending and will be integrated into this library when completed.

Client API / Interface

After a storage service object (i.e. `blockblobservice`) has been created, the user may assign values to the fields that constitute an encryption policy: `key_encryption_key`, `key_resolver_function`, and `require_encryption`. Users can provide only a KEK, only a resolver, or both. `key_encryption_key` is the basic key type that is identified using a key identifier and that provides the logic for wrapping/unwrapping. `key_resolver_function` is used to resolve a key during the decryption process. It returns a valid KEK given a key identifier. This provides users the ability to choose between multiple keys that are managed in multiple locations.

The KEK must implement the following methods to successfully encrypt data:

- `wrap_key(cek)`: Wraps the specified CEK (bytes) using an algorithm of the user's choice. Returns the wrapped key.
- `get_key_wrap_algorithm()`: Returns the algorithm used to wrap keys.
- `get_kid()`: Returns the string key id for this KEK. The KEK must implement the following methods to successfully decrypt data:
 - `unwrap_key(cek, algorithm)`: Returns the unwrapped form of the specified CEK using the string-specified algorithm.
 - `get_kid()`: Returns a string key id for this KEK.

The key resolver must at least implement a method that, given a key id, returns the corresponding KEK implementing the interface above. Only this method is to be assigned to the `key_resolver_function` property on the service object.

- For encryption, the key is used always and the absence of a key will result in an error.
- For decryption:
 - The key resolver is invoked if specified to get the key. If the resolver is specified but does not have a mapping for the key identifier, an error is thrown.
 - If resolver is not specified but a key is specified, the key is used if its identifier matches the required key identifier. If the identifier does not match, an error is thrown.

The encryption samples in `azure.storage.samples` demonstrate a more detailed end-to-end scenario for blobs, queues and tables. Sample implementations of the KEK and key resolver are provided in the sample files as `KeyWrapper` and `KeyResolver` respectively.

RequireEncryption mode

Users can optionally enable a mode of operation where all uploads and downloads must be encrypted. In this mode, attempts to upload data without an encryption policy or download data that is not encrypted on the service will fail on the client. The **require_encryption** flag on the service object controls this behavior.

Blob service encryption

Set the encryption policy fields on the blockblobservice object. Everything else will be handled by the client library internally.

```
# Create the KEK used for encryption.  
# KeyWrapper is the provided sample implementation, but the user may use their own object as long as it  
implements the interface above.  
kek = KeyWrapper('local:key1') # Key identifier  
  
# Create the key resolver used for decryption.  
# KeyResolver is the provided sample implementation, but the user may use whatever implementation they choose so  
long as the function set on the service object behaves appropriately.  
key_resolver = KeyResolver()  
key_resolver.put_key(kek)  
  
# Set the KEK and key resolver on the service object.  
my_block_blob_service.key_encryption_key = kek  
my_block_blob_service.key_resolver_funcion = key_resolver.resolve_key  
  
# Upload the encrypted contents to the blob.  
my_block_blob_service.create_blob_from_stream(container_name, blob_name, stream)  
  
# Download and decrypt the encrypted contents from the blob.  
blob = my_block_blob_service.get_blob_to_bytes(container_name, blob_name)
```

Queue service encryption

Set the encryption policy fields on the queueservice object. Everything else will be handled by the client library internally.

```
# Create the KEK used for encryption.  
# KeyWrapper is the provided sample implementation, but the user may use their own object as long as it  
implements the interface above.  
kek = KeyWrapper('local:key1') # Key identifier  
  
# Create the key resolver used for decryption.  
# KeyResolver is the provided sample implementation, but the user may use whatever implementation they choose so  
long as the function set on the service object behaves appropriately.  
key_resolver = KeyResolver()  
key_resolver.put_key(kek)  
  
# Set the KEK and key resolver on the service object.  
my_queue_service.key_encryption_key = kek  
my_queue_service.key_resolver_funcion = key_resolver.resolve_key  
  
# Add message  
my_queue_service.put_message(queue_name, content)  
  
# Retrieve message  
retrieved_message_list = my_queue_service.get_messages(queue_name)
```

Table service encryption

In addition to creating an encryption policy and setting it on request options, you must either specify an **encryption_resolver_function** on the **tableservice**, or set the encrypt attribute on the EntityProperty.

Using the resolver

```

# Create the KEK used for encryption.
# KeyWrapper is the provided sample implementation, but the user may use their own object as long as it
implements the interface above.
kek = KeyWrapper('local:key1') # Key identifier

# Create the key resolver used for decryption.
# KeyResolver is the provided sample implementation, but the user may use whatever implementation they choose so
long as the function set on the service object behaves appropriately.
key_resolver = KeyResolver()
key_resolver.put_key(kek)

# Define the encryption resolver_function.
def my_encryption_resolver(pk, rk, property_name):
    if property_name == 'foo':
        return True
    return False

# Set the KEK and key resolver on the service object.
my_table_service.key_encryption_key = kek
my_table_service.key_resolver_funcion = key_resolver.resolve_key
my_table_service.encryption_resolver_function = my_encryption_resolver

# Insert Entity
my_table_service.insert_entity(table_name, entity)

# Retrieve Entity
# Note: No need to specify an encryption resolver for retrieve, but it is harmless to leave the property set.
my_table_service.get_entity(table_name, entity['PartitionKey'], entity['RowKey'])

```

Using attributes

As mentioned above, a property may be marked for encryption by storing it in an EntityProperty object and setting the encrypt field.

```
encrypted_property_1 = EntityProperty(EdmType.STRING, value, encrypt=True)
```

Encryption and performance

Note that encrypting your storage data results in additional performance overhead. The content key and IV must be generated, the content itself must be encrypted, and additional metadata must be formatted and uploaded. This overhead will vary depending on the quantity of data being encrypted. We recommend that customers always test their applications for performance during development.

Next steps

- Download the [Azure Storage Client Library for Java PyPi package](#)
- Download the [Azure Storage Client Library for Python Source Code from GitHub](#)

Storage Analytics

1/17/2017 • 10 min to read • [Edit on GitHub](#)

Overview

Azure Storage Analytics performs logging and provides metrics data for a storage account. You can use this data to trace requests, analyze usage trends, and diagnose issues with your storage account.

To use Storage Analytics, you must enable it individually for each service you want to monitor. You can enable it from the [Azure Portal](#). For details, see [Monitor a storage account in the Azure Portal](#). You can also enable Storage Analytics programmatically via the REST API or the client library. Use the [Get Blob Service Properties](#), [Get Queue Service Properties](#), [Get Table Service Properties](#), and [Get File Service Properties](#) operations to enable Storage Analytics for each service.

The aggregated data is stored in a well-known blob (for logging) and in well-known tables (for metrics), which may be accessed using the Blob service and Table service APIs.

Storage Analytics has a 20 TB limit on the amount of stored data that is independent of the total limit for your storage account. For more information about billing and data retention policies, see [Storage Analytics and Billing](#). For more information about storage account limits, see [Azure Storage Scalability and Performance Targets](#).

For an in-depth guide on using Storage Analytics and other tools to identify, diagnose, and troubleshoot Azure Storage-related issues, see [Monitor, diagnose, and troubleshoot Microsoft Azure Storage](#).

About Storage Analytics logging

Storage Analytics logs detailed information about successful and failed requests to a storage service. This information can be used to monitor individual requests and to diagnose issues with a storage service. Requests are logged on a best-effort basis.

Log entries are created only if there is storage service activity. For example, if a storage account has activity in its Blob service but not in its Table or Queue services, only logs pertaining to the Blob service will be created.

Storage Analytics Logging is not available for the Azure File Service.

Logging authenticated requests

The following types of authenticated requests are logged:

- Successful requests.
- Failed requests, including timeout, throttling, network, authorization, and other errors.
- Requests using a Shared Access Signature (SAS), including failed and successful requests.
- Requests to analytics data.

Requests made by Storage Analytics itself, such as log creation or deletion, are not logged. A full list of the logged data is documented in the [Storage Analytics Logged Operations and Status Messages](#) and [Storage Analytics Log Format](#) topics.

Logging anonymous requests

The following types of anonymous requests are logged:

- Successful requests.
- Server errors.
- Timeout errors for both client and server.

- Failed GET requests with error code 304 (Not Modified).

All other failed anonymous requests are not logged. A full list of the logged data is documented in the [Storage Analytics Logged Operations and Status Messages](#) and [Storage Analytics Log Format](#) topics.

How logs are stored

All logs are stored in block blobs in a container named \$logs, which is automatically created when Storage Analytics is enabled for a storage account. The \$logs container is located in the blob namespace of the storage account, for example: `http://<accountname>.blob.core.windows.net/$logs`. This container cannot be deleted once Storage Analytics has been enabled, though its contents can be deleted.

NOTE

The \$logs container is not displayed when a container listing operation is performed, such as the [ListContainers](#) method. It must be accessed directly. For example, you can use the [ListBlobs](#) method to access the blobs in the `$logs` container. As requests are logged, Storage Analytics will upload intermediate results as blocks. Periodically, Storage Analytics will commit these blocks and make them available as a blob.

Duplicate records may exist for logs created in the same hour. You can determine if a record is a duplicate by checking the **RequestId** and **Operation** number.

Log naming conventions

Each log will be written in the following format.

```
<service-name>/YYYY/MM/DD/hhmm/<counter>.log
```

The following table describes each attribute in the log name.

ATTRIBUTE	DESCRIPTION
	The name of the storage service. For example: blob, table, or queue.
YYYY	The four-digit year for the log. For example: 2011.
MM	The two-digit month for the log. For example: 07.
DD	The two-digit month for the log. For example: 07.
hh	The two-digit hour that indicates the starting hour for the logs, in 24-hour UTC format. For example: 18.
mm	The two-digit number that indicates the starting minute for the logs. This value is unsupported in the current version of Storage Analytics, and its value will always be 00.
	A zero-based counter with six digits that indicates the number of log blobs generated for the storage service in an hour time period. This counter starts at 000000. For example: 000001.

The following is a complete sample log name that combines the previous examples.

```
blob/2011/07/31/1800/000001.log
```

The following is a sample URI that can be used to access the previous log.

```
https://<accountname>.blob.core.windows.net/$logs/blob/2011/07/31/1800/000001.log
```

When a storage request is logged, the resulting log name correlates to the hour when the requested operation completed. For example, if a GetBlob request was completed at 6:30 P.M. on 7/31/2011, the log would be written with the following prefix: `blob/2011/07/31/1800/`

Log metadata

All log blobs are stored with metadata that can be used to identify what logging data the blob contains. The following table describes each metadata attribute.

ATTRIBUTE	DESCRIPTION
LogType	Describes whether the log contains information pertaining to read, write, or delete operations. This value can include one type or a combination of all three, separated by commas. Example 1: write; Example 2: read,write; Example 3: read,write,delete.
StartTime	The earliest time of an entry in the log, in the form of YYYY-MM-DDThh:mm:ssZ. For example: 2011-07-31T18:21:46Z.
EndTime	The latest time of an entry in the log, in the form of YYYY-MM-DDThh:mm:ssZ. For example: 2011-07-31T18:22:09Z.
LogVersion	The version of the log format. Currently the only supported value is 1.0.

The following list displays complete sample metadata using the previous examples.

- LogType=write
- StartTime=2011-07-31T18:21:46Z
- EndTime=2011-07-31T18:22:09Z
- LogVersion=1.0

Accessing logging data

All data in the `/$logs` container can be accessed by using the Blob service APIs, including the .NET APIs provided by the Azure managed library. The storage account administrator can read and delete logs, but cannot create or update them. Both the log's metadata and log name can be used when querying for a log. It is possible for a given hour's logs to appear out of order, but the metadata always specifies the timespan of log entries in a log. Therefore, you can use a combination of log names and metadata when searching for a particular log.

About Storage Analytics metrics

Storage Analytics can store metrics that include aggregated transaction statistics and capacity data about requests to a storage service. Transactions are reported at both the API operation level as well as at the storage service level, and capacity is reported at the storage service level. Metrics data can be used to analyze storage service usage, diagnose issues with requests made against the storage service, and to improve the performance of applications that use a service.

To use Storage Analytics, you must enable it individually for each service you want to monitor. You can enable it from the [Azure Portal](#). For details, see [Monitor a storage account in the Azure Portal](#). You can also enable Storage Analytics programmatically via the REST API or the client library. Use the **Get Service Properties** operations to enable Storage Analytics for each service.

Transaction metrics

A robust set of data is recorded at hourly or minute intervals for each storage service and requested API operation, including ingress/egress, availability, errors, and categorized request percentages. You can see a complete list of the transaction details in the [Storage Analytics Metrics Table Schema](#) topic.

Transaction data is recorded at two levels – the service level and the API operation level. At the service level, statistics summarizing all requested API operations are written to a table entity every hour even if no requests were made to the service. At the API operation level, statistics are only written to an entity if the operation was requested within that hour.

For example, if you perform a **GetBlob** operation on your Blob service, Storage Analytics Metrics will log the request and include it in the aggregated data for both the Blob service as well as the **GetBlob** operation. However, if no **GetBlob** operation is requested during the hour, an entity will not be written to the `$MetricsTransactionsBlob` for that operation.

Transaction metrics are recorded for both user requests and requests made by Storage Analytics itself. For example, requests by Storage Analytics to write logs and table entities are recorded. For more information about how these requests are billed, see [Storage Analytics and Billing](#).

Capacity metrics

NOTE

Currently, capacity metrics are only available for the Blob service. Capacity metrics for the Table service and Queue service will be available in future versions of Storage Analytics.

Capacity data is recorded daily for a storage account's Blob service, and two table entities are written. One entity provides statistics for user data, and the other provides statistics about the `$logs` blob container used by Storage Analytics. The `$MetricsCapacityBlob` table includes the following statistics:

- **Capacity**: The amount of storage used by the storage account's Blob service, in bytes.
- **ContainerCount**: The number of blob containers in the storage account's Blob service.
- **ObjectCount**: The number of committed and uncommitted block or page blobs in the storage account's Blob service.

For more information about the capacity metrics, see [Storage Analytics Metrics Table Schema](#).

How metrics are stored

All metrics data for each of the storage services is stored in three tables reserved for that service: one table for transaction information, one table for minute transaction information, and another table for capacity information. Transaction and minute transaction information consists of request and response data, and capacity information consists of storage usage data. Hour metrics, minute metrics, and capacity for a storage account's Blob service can be accessed in tables that are named as described in the following table.

METRICS LEVEL	TABLE NAMES	SUPPORTED VERSIONS
Hourly metrics, primary location	<code>\$MetricsTransactionsBlob</code> <code>\$MetricsTransactionsTable</code> <code>\$MetricsTransactionsQueue</code>	Versions prior to 2013-08-15 only. While these names are still supported, it's recommended that you switch to using the tables listed below.
Hourly metrics, primary location	<code>\$MetricsHourPrimaryTransactionsBlob</code> <code>\$MetricsHourPrimaryTransactionsTable</code> <code>\$MetricsHourPrimaryTransactionsQueue</code>	All versions, including 2013-08-15.

METRICS LEVEL	TABLE NAMES	SUPPORTED VERSIONS
Minute metrics, primary location	\$MetricsMinutePrimaryTransactionsBlob \$MetricsMinutePrimaryTransactionsTable \$MetricsMinutePrimaryTransactionsQueue	All versions, including 2013-08-15.
Hourly metrics, secondary location	\$MetricsHourSecondaryTransactionsBlob \$MetricsHourSecondaryTransactionsTable \$MetricsHourSecondaryTransactionsQueue	All versions, including 2013-08-15. Read-access geo-redundant replication must be enabled.
Minute metrics, secondary location	\$MetricsMinuteSecondaryTransactionsBlob \$MetricsMinuteSecondaryTransactionsTable \$MetricsMinuteSecondaryTransactionsQueue	All versions, including 2013-08-15. Read-access geo-redundant replication must be enabled.
Capacity (Blob service only)	\$MetricsCapacityBlob	All versions, including 2013-08-15.

These tables are automatically created when Storage Analytics is enabled for a storage account. They are accessed via the namespace of the storage account, for example:

```
https://<accountname>.table.core.windows.net/Tables("&quot;$MetricsTransactionsBlob&quot;")
```

Accessing metrics data

All data in the metrics tables can be accessed by using the Table service APIs, including the .NET APIs provided by the Azure managed library. The storage account administrator can read and delete table entities, but cannot create or update them.

Billing for Storage Analytics

Storage Analytics is enabled by a storage account owner; it is not enabled by default. All metrics data is written by the services of a storage account. As a result, each write operation performed by Storage Analytics is billable. Additionally, the amount of storage used by metrics data is also billable.

The following actions performed by Storage Analytics are billable:

- Requests to create blobs for logging.
- Requests to create table entities for metrics.

If you have configured a data retention policy, you are not charged for delete transactions when Storage Analytics deletes old logging and metrics data. However, delete transactions from a client are billable. For more information about retention policies, see [Setting a Storage Analytics Data Retention Policy](#).

Understanding billable requests

Every request made to an account's storage service is either billable or non-billable. Storage Analytics logs each individual request made to a service, including a status message that indicates how the request was handled. Similarly, Storage Analytics stores metrics for both a service and the API operations of that service, including the percentages and count of certain status messages. Together, these features can help you analyze your billable requests, make improvements on your application, and diagnose issues with requests to your services. For more information about billing, see [Understanding Azure Storage Billing - Bandwidth, Transactions, and Capacity](#).

When looking at Storage Analytics data, you can use the tables in the [Storage Analytics Logged Operations and Status Messages](#) topic to determine what requests are billable. Then you can compare your logs and metrics data to the status messages to see if you were charged for a particular request. You can also use the tables in the previous topic to investigate availability for a storage service or individual API operation.

Next steps

Setting up Storage Analytics

- [Monitor a storage account in the Azure Portal](#)
- [Enabling and Configuring Storage Analytics](#)

Storage Analytics logging

- [About Storage Analytics Logging](#)
- [Storage Analytics Log Format](#)
- [Storage Analytics Logged Operations and Status Messages](#)

Storage Analytics metrics

- [About Storage Analytics Metrics](#)
- [Storage Analytics Metrics Table Schema](#)
- [Storage Analytics Logged Operations and Status Messages](#)

Enabling Azure Storage metrics and viewing metrics data

1/17/2017 • 8 min to read • [Edit on GitHub](#)

Overview

By default, Storage Metrics is not enabled for your storage services. You can enable monitoring via the [Azure Portal](#) or Windows PowerShell, or programmatically via the storage client library.

When you enable Storage Metrics, you must choose a retention period for the data: this period determines for how long the storage service keeps the metrics and charges you for the space required to store them. Typically, you should use a shorter retention period for minute metrics than hourly metrics because of the significant extra space required for minute metrics. You should choose a retention period such that you have sufficient time to analyze the data and download any metrics you wish to keep for off-line analysis or reporting purposes. Remember that you will also be billed for downloading metrics data from your storage account.

How to enable metrics using the Azure Portal

Follow these steps to enable metrics in the [Azure Portal](#):

1. Navigate to your storage account.
2. Open the **Settings** blade and select **Diagnostics**.
3. Ensure that **Status** is set to **On**.
4. Select the metrics for the services you wish to monitor.
5. Specify a retention policy to indicate how long to retain metrics and log data.

Note that the [Azure Portal](#) does not currently enable you to configure minute metrics in your storage account; you must enable minute metrics using PowerShell or programmatically.

How to enable metrics using PowerShell

You can use PowerShell on your local machine to configure Storage Metrics in your storage account by using the Azure PowerShell cmdlet `Get-AzureStorageServiceMetricsProperty` to retrieve the current settings, and the cmdlet `Set-AzureStorageServiceMetricsProperty` to change the current settings.

The cmdlets that control Storage Metrics use the following parameters:

- `MetricsType`: possible values are Hour and Minute.
- `ServiceType`: possible values are Blob, Queue, and Table.
- `MetricsLevel`: possible values are None, Service, and ServiceAndApi.

For example, the following command switches on minute metrics for the Blob service in your default storage account with the retention period set to five days:

```
Set-AzureStorageServiceMetricsProperty -MetricsType Minute -ServiceType Blob -MetricsLevel ServiceAndApi -RetentionDays 5`
```

The following command retrieves the current hourly metrics level and retention days for the blob service in your default storage account:

```
Get-AzureStorageServiceMetricsProperty -MetricsType Hour -ServiceType Blob
```

For information about how to configure the Azure PowerShell cmdlets to work with your Azure subscription and how to select the default storage account to use, see: [How to install and configure Azure PowerShell](#).

How to enable Storage metrics programmatically

The following C# snippet shows how to enable metrics and logging for the Blob service using the storage client library for .NET:

```
//Parse the connection string for the storage account.
const string ConnectionString = "DefaultEndpointsProtocol=https;AccountName=account-name;AccountKey=account-key";
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(ConnectionString);

// Create service client for credentialled access to the Blob service.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Enable Storage Analytics logging and set retention policy to 10 days.
ServiceProperties properties = new ServiceProperties();
properties.Logging.LoggingOperations = LoggingOperations.All;
properties.Logging.RetentionDays = 10;
properties.Logging.Version = "1.0";

// Configure service properties for metrics. Both metrics and logging must be set at the same time.
properties.HourMetrics.MetricsLevel = MetricsLevel.ServiceAndApi;
properties.HourMetrics.RetentionDays = 10;
properties.HourMetrics.Version = "1.0";

properties.MinuteMetrics.MetricsLevel = MetricsLevel.ServiceAndApi;
properties.MinuteMetrics.RetentionDays = 10;
properties.MinuteMetrics.Version = "1.0";

// Set the default service version to be used for anonymous requests.
properties.DefaultServiceVersion = "2015-04-05";

// Set the service properties.
blobClient.SetServiceProperties(properties);
```

Viewing Storage metrics

After you configure Storage Analytics metrics to monitor your storage account, Storage Analytics records the metrics in a set of well-known tables in your storage account. You can configure charts to view hourly metrics in the [Azure Portal](#):

1. Navigate to your storage account in the [Azure Portal](#).
2. In the **Monitoring** section, click **Add Tiles** to add a new chart. In the **Tile Gallery**, select the metric you want to view, and drag it to the **Monitoring** section.
3. To edit which metrics are displayed in a chart, click the **Edit** link. You can add or remove individual metrics by selecting or deselecting them.
4. Click **Save** when you've finished editing metrics.

If you want to download the metrics for long-term storage or to analyze them locally, you will need to:

- Use a tool that is aware of these tables and will allow you to view and download them.
- Write a custom application or script to read and store the tables.

Many third-party storage-browsing tools are aware of these tables and enable you to view them directly. Please see [Azure Storage Explorers](#) for a list of available tools.

NOTE

Starting with version 0.8.0 of the [Microsoft Azure Storage Explorer](#), you will now be able to view and download the analytics metrics tables.

In order to access the analytics tables programmatically, do note that the analytics tables do not appear if you list all the tables in your storage account. You can either access them directly by name, or use the [CloudAnalyticsClient API](#) in the .NET client library to query the table names.

Hourly metrics

- \$MetricsHourPrimaryTransactionsBlob
- \$MetricsHourPrimaryTransactionsTable
- \$MetricsHourPrimaryTransactionsQueue

Minute metrics

- \$MetricsMinutePrimaryTransactionsBlob
- \$MetricsMinutePrimaryTransactionsTable
- \$MetricsMinutePrimaryTransactionsQueue

Capacity

- \$MetricsCapacityBlob

You can find full details of the schemas for these tables at [Storage Analytics Metrics Table Schema](#). The sample rows below show only a subset of the columns available, but illustrate some important features of the way Storage Metrics saves these metrics:

PARTITIONKEY	ROWKEY	TIMESTAMP	TOTALREQUESTS	TOTALBILLABLEREQUESTS	TOTALINGRESS	TOTALEGRESS	AVAILABILITY	AVERAGEE2ELATENCY	AVERAGESERVEABILITY	PERCENTSUCCESS
20140522T1100	user;All	2014-05-22T11:01:16.7650250Z	7	7	4003	46801	100	104.4286	6.857143	100
20140522T1100	user;QueryEntities	2014-05-22T11:01:16.7640250Z	5	5	2694	45951	100	143.8	7.8	100
20140522T1100	user;QueryEntity	2014-05-22T11:01:16.7650250Z	1	1	538	633	100	3	3	100

PARTITIONKEY	ROWKEY	TIMESTAMP	TOTALREQUESTS	TOTALBILLABLEREQUESTS	TOTALINGRESS	TOTALEGRESS	AVAILABILITY	AVERAGEE2ELATENCY	AVERAGESERVEURLATENCY	PERCENTSUCCESS
20140522T1100	user;UpdateEntity	2014-05-22T11:01:16.7650250Z	1	1	771	217	100	9	6	100

In this example minute metrics data, the partition key uses the time at minute resolution. The row key identifies the type of information that is stored in the row and this is composed of two pieces of information, the access type, and the request type:

- The access type is either user or system, where user refers to all user requests to the storage service, and system refers to requests made by Storage Analytics.
- The request type is either all in which case it is a summary line, or it identifies the specific API such as QueryEntity or UpdateEntity.

The sample data above shows all the records for a single minute (starting at 11:00AM), so the number of QueryEntities requests plus the number of QueryEntity requests plus the number of UpdateEntity requests add up to seven, which is the total shown on the user>All row. Similarly, you can derive the average end-to-end latency 104.4286 on the user>All row by calculating $((143.8 * 5) + 3 + 9)/7$.

You should consider setting up alerts in the [Azure Portal](#) on the Monitor page so that Storage Metrics can automatically notify you of any important changes in the behavior of your storage services. If you use a storage explorer tool to download this metrics data in a delimited format, you can use Microsoft Excel to analyze the data. See the blog post [Microsoft Azure Storage Explorers](#) for a list of available storage explorer tools.

Accessing metrics data programmatically

The following listing shows sample C# code that accesses the minute metrics for a range of minutes and displays the results in a console Window. It uses the Azure Storage Library version 4 that includes the CloudAnalyticsClient class that simplifies accessing the metrics tables in storage.

```

private static void PrintMinuteMetrics(CloudAnalyticsClient analyticsClient, DateTimeOffset startDateTime,
DateTimeOffset endDateTime)
{
    // Convert the dates to the format used in the PartitionKey
    var start = startDateTime.UnixTime("yyyyMMdd'T'HHmm");
    var end = endDateTime.UnixTime("yyyyMMdd'T'HHmm");

    var services = Enum.GetValues(typeof(StorageService));
    foreach (StorageService service in services)
    {
        Console.WriteLine("Minute Metrics for Service {0} from {1} to {2} UTC", service, start, end);
        var metricsQuery = analyticsClient.CreateMinuteMetricsQuery(service, StorageLocation.Primary);
        var t = analyticsClient.GetMinuteMetricsTable(service);
        var opContext = new OperationContext();
        var query =
            from entity in metricsQuery
            // Note, you can't filter using the entity properties Time, AccessType, or TransactionType
            // because they are calculated fields in the MetricsEntity class.
            // The PartitionKey identifies the DateTime of the metrics.
            where entity.PartitionKey.CompareTo(start) >= 0 && entity.PartitionKey.CompareTo(end) <= 0
            select entity;

        // Filter on "user" transactions after fetching the metrics from Table Storage.
        // (StartsWith is not supported using LINQ with Azure table storage)
        var results = query.ToList().Where(m => m.RowKey.StartsWith("user"));
        var resultString = results.Aggregate(new StringBuilder(), (builder, metrics) =>
builder.AppendLine(MetricsString(metrics, opContext))).ToString();
        Console.WriteLine(resultString);
    }
}

private static string MetricsString(MetricsEntity entity, OperationContext opContext)
{
    var entityProperties = entity.WriteEntity(opContext);
    var entityString =
        string.Format("Time: {0}, ", entity.Time) +
        string.Format("AccessType: {0}, ", entity.AccessType) +
        string.Format("TransactionType: {0}, ", entity.TransactionType) +
        string.Join(", ", entityProperties.Select(e => new KeyValuePair<string, string>(e.Key.ToString(),
e.Value.PropertyAsObject.ToString())));
    return entityString;
}

```

What charges do you incur when you enable storage metrics?

Write requests to create table entities for metrics are charged at the standard rates applicable to all Azure Storage operations.

Read and delete requests by a client to metrics data are also billable at standard rates. If you have configured a data retention policy, you are not charged when Azure Storage deletes old metrics data. However, if you delete analytics data, your account is charged for the delete operations.

The capacity used by the metrics tables is also billable: you can use the following to estimate the amount of capacity used for storing metrics data:

- If each hour a service utilizes every API in every service, then approximately 148KB of data is stored every hour in the metrics transaction tables if you have enabled both service and API level summary.
- If each hour a service utilizes every API in every service, then approximately 12KB of data is stored every hour in the metrics transaction tables if you have enabled just service level summary.
- The capacity table for blobs has two rows added each day (provided user has opted in for logs): this implies that every day the size of this table increases by up to approximately 300 bytes.

Next-steps:

[Enabling Storage Logging and Accessing Log Data](#)

Monitor, diagnose, and troubleshoot Microsoft Azure Storage

1/17/2017 • 56 min to read • [Edit on GitHub](#)

Overview

Diagnosing and troubleshooting issues in a distributed application hosted in a cloud environment can be more complex than in traditional environments. Applications can be deployed in a PaaS or IaaS infrastructure, on premises, on a mobile device, or in some combination of these. Typically, your application's network traffic may traverse public and private networks and your application may use multiple storage technologies such as Microsoft Azure Storage Tables, Blobs, Queues, or Files in addition to other data stores such as relational and document databases.

To manage such applications successfully you should monitor them proactively and understand how to diagnose and troubleshoot all aspects of them and their dependent technologies. As a user of Azure Storage services, you should continuously monitor the Storage services your application uses for any unexpected changes in behavior (such as slower than usual response times), and use logging to collect more detailed data and to analyze a problem in depth. The diagnostics information you obtain from both monitoring and logging will help you to determine the root cause of the issue your application encountered. Then you can troubleshoot the issue and determine the appropriate steps you can take to remediate it. Azure Storage is a core Azure service, and forms an important part of the majority of solutions that customers deploy to the Azure infrastructure. Azure Storage includes capabilities to simplify monitoring, diagnosing, and troubleshooting storage issues in your cloud-based applications.

NOTE

Storage accounts with a replication type of Zone-Redundant Storage (ZRS) do not have the metrics or logging capability enabled at this time. Also, the Azure Files Service does not support logging at this time.

For a hands-on guide to end-to-end troubleshooting in Azure Storage applications, see [End-to-End Troubleshooting using Azure Storage Metrics and Logging, AzCopy, and Message Analyzer](#).

- [Introduction](#)
 - [How this guide is organized](#)
- [Monitoring your storage service](#)
 - [Monitoring service health](#)
 - [Monitoring capacity](#)
 - [Monitoring availability](#)
 - [Monitoring performance](#)
- [Diagnosing storage issues](#)
 - [Service health issues](#)
 - [Performance issues](#)
 - [Diagnosing errors](#)
 - [Storage emulator issues](#)
 - [Storage logging tools](#)
 - [Using network logging tools](#)
- [End-to-end tracing](#)

- Correlating log data
- Client request ID
- Server request ID
- Timestamps
- Troubleshooting guidance
 - Metrics show high AverageE2ELatency and low AverageServerLatency
 - Metrics show low AverageE2ELatency and low AverageServerLatency but the client is experiencing high latency
 - Metrics show high AverageServerLatency
 - You are experiencing unexpected delays in message delivery on a queue
 - Metrics show an increase in PercentThrottlingError
 - Metrics show an increase in PercentTimeoutError
 - Metrics show an increase in PercentNetworkError
 - The client is receiving HTTP 403 (Forbidden) messages
 - The client is receiving HTTP 404 (Not found) messages
 - The client is receiving HTTP 409 (Conflict) messages
 - Metrics show low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors
 - Capacity metrics show an unexpected increase in storage capacity usage
 - You are experiencing unexpected reboots of Virtual Machines that have a large number of attached VHDs
 - Your issue arises from using the storage emulator for development or test
 - You are encountering problems installing the Azure SDK for .NET
 - You have a different issue with a storage service
 - Troubleshooting Azure Files issues with Windows and Linux
- Appendices
 - Appendix 1: Using Fiddler to capture HTTP and HTTPS traffic
 - Appendix 2: Using Wireshark to capture network traffic
 - Appendix 3: Using Microsoft Message Analyzer to capture network traffic
 - Appendix 4: Using Excel to view metrics and log data
 - Appendix 5: Monitoring with Application Insights for Visual Studio Team Services

Introduction

This guide shows you how to use features such as Azure Storage Analytics, client-side logging in the Azure Storage Client Library, and other third-party tools to identify, diagnose, and troubleshoot Azure Storage related issues.

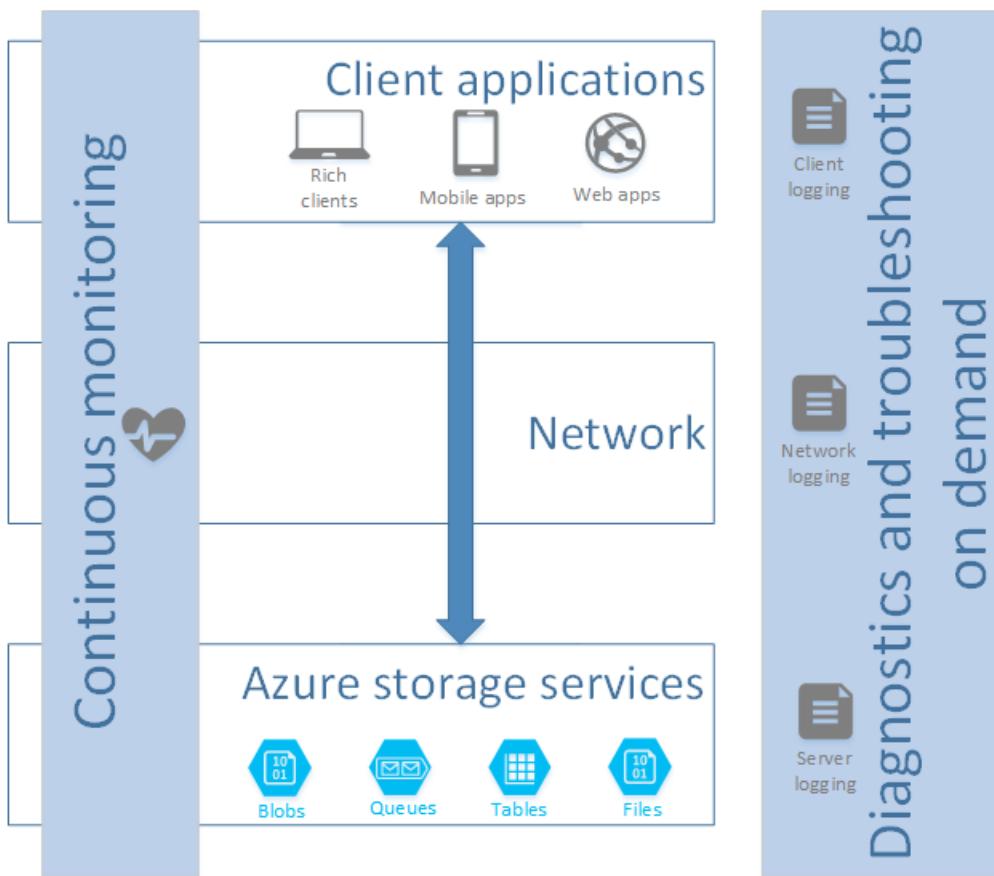


Figure 1 Monitoring, Diagnostics, and Troubleshooting

This guide is intended to be read primarily by developers of online services that use Azure Storage Services and IT Pros responsible for managing such online services. The goals of this guide are:

- To help you maintain the health and performance of your Azure Storage accounts.
- To provide you with the necessary processes and tools to help you decide if an issue or problem in an application relates to Azure Storage.
- To provide you with actionable guidance for resolving problems related to Azure Storage.

How this guide is organized

The section "[Monitoring your storage service](#)" describes how to monitor the health and performance of your Azure Storage services using Azure Storage Analytics Metrics (Storage Metrics).

The section "[Diagnosing storage issues](#)" describes how to diagnose issues using Azure Storage Analytics Logging (Storage Logging). It also describes how to enable client-side logging using the facilities in one of the client libraries such as the Storage Client Library for .NET or the Azure SDK for Java.

The section "[End-to-end tracing](#)" describes how you can correlate the information contained in various log files and metrics data.

The section "[Troubleshooting guidance](#)" provides troubleshooting guidance for some of the common storage-related issues you might encounter.

The "[Appendices](#)" include information about using other tools such as Wireshark and Netmon for analyzing network packet data, Fiddler for analyzing HTTP/HTTPS messages, and Microsoft Message Analyzer for correlating log data.

Monitoring your storage service

If you are familiar with Windows performance monitoring, you can think of Storage Metrics as being an Azure

Storage equivalent of Windows Performance Monitor counters. In Storage Metrics you will find a comprehensive set of metrics (counters in Windows Performance Monitor terminology) such as service availability, total number of requests to service, or percentage of successful requests to service. For a full list of the available metrics, see [Storage Analytics Metrics Table Schema](#). You can specify whether you want the storage service to collect and aggregate metrics every hour or every minute. For more information about how to enable metrics and monitor your storage accounts, see [Enabling storage metrics and viewing metrics data](#).

You can choose which hourly metrics you want to display in the [Azure Portal](#) and configure rules that notify administrators by email whenever an hourly metric exceeds a particular threshold. For more information, see [Receive Alert Notifications](#).

The storage service collects metrics using a best effort, but may not record every storage operation.

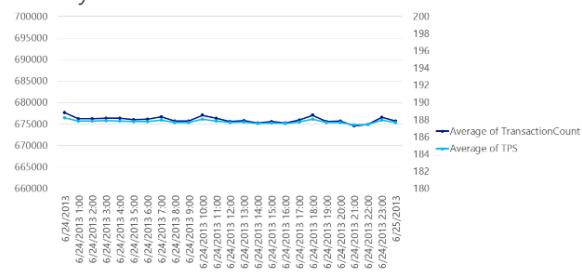
In the Azure Portal, you can view metrics such as availability, total requests, and average latency numbers for a storage account. A notification rule has also been set up to alert an administrator if availability drops below a certain level. From viewing this data, one possible area for investigation is the table service success percentage being below 100% (for more information, see the section "[Metrics show low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors](#)").

You should continuously monitor your Azure applications to ensure they are healthy and performing as expected by:

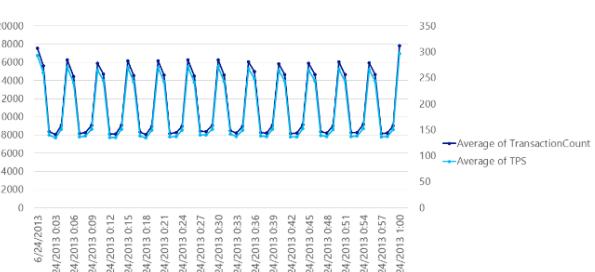
- Establishing some baseline metrics for application that will enable you to compare current data and identify any significant changes in the behavior of Azure storage and your application. The values of your baseline metrics will, in many cases, be application specific and you should establish them when you are performance testing your application.
- Recording minute metrics and using them to monitor actively for unexpected errors and anomalies such as spikes in error counts or request rates.
- Recording hourly metrics and using them to monitor average values such as average error counts and request rates.
- Investigating potential issues using diagnostics tools as discussed later in the section "[Diagnosing storage issues](#)."

The charts in Figure 3 below illustrate how the averaging that occurs for hourly metrics can hide spikes in activity. The hourly metrics appear to show a steady rate of requests, while the minute metrics reveal the fluctuations that are really taking place.

Hourly metrics - TPS



Minute Metrics - TPS



The remainder of this section describes what metrics you should monitor and why.

Monitoring service health

You can use the [Azure Portal](#) to view the health of the Storage service (and other Azure services) in all the Azure regions around the world. This enables you to see immediately if an issue outside of your control is affecting the Storage service in the region you use for your application.

The [Azure Portal](#) can also provide notifications of incidents that affect the various Azure services. Note: This information was previously available, along with historical data, on the [Azure Service Dashboard](#).

While the [Azure Portal](#) collects health information from inside the Azure datacenters (inside-out monitoring), you could also consider adopting an outside-in approach to generate synthetic transactions that periodically access your Azure-hosted web application from multiple locations. The services offered by [Dynatrace](#) and Application Insights for Visual Studio Team Services are examples of this outside-in approach. For more information about Application Insights for Visual Studio Team Services, see the appendix "[Appendix 5: Monitoring with Application Insights for Visual Studio Team Services](#)."

Monitoring capacity

Storage Metrics only stores capacity metrics for the blob service because blobs typically account for the largest proportion of stored data (at the time of writing, it is not possible to use Storage Metrics to monitor the capacity of your tables and queues). You can find this data in the **\$MetricsCapacityBlob** table if you have enabled monitoring for the Blob service. Storage Metrics records this data once per day, and you can use the value of the **RowKey** to determine whether the row contains an entity that relates to user data (value **data**) or analytics data (value **analytics**). Each stored entity contains information about the amount of storage used (**Capacity** measured in bytes) and the current number of containers (**ContainerCount**) and blobs (**ObjectCount**) in use in the storage account. For more information about the capacity metrics stored in the **\$MetricsCapacityBlob** table, see [Storage Analytics Metrics Table Schema](#).

NOTE

You should monitor these values for an early warning that you are approaching the capacity limits of your storage account. In the Azure Portal, you can add alert rules to notify you if aggregate storage use exceeds or falls below thresholds that you specify.

For help estimating the size of various storage objects such as blobs, see the blog post [Understanding Azure Storage Billing – Bandwidth, Transactions, and Capacity](#).

Monitoring availability

You should monitor the availability of the storage services in your storage account by monitoring the value in the **Availability** column in the hourly or minute metrics tables — **\$MetricsHourPrimaryTransactionsBlob**, **\$MetricsHourPrimaryTransactionsTable**, **\$MetricsHourPrimaryTransactionsQueue**, **\$MetricsMinutePrimaryTransactionsBlob**, **\$MetricsMinutePrimaryTransactionsTable**, **\$MetricsMinutePrimaryTransactionsQueue**, **\$MetricsCapacityBlob**. The **Availability** column contains a percentage value that indicates the availability of the service or the API operation represented by the row (the **RowKey** shows if the row contains metrics for the service as a whole or for a specific API operation).

Any value less than 100% indicates that some storage requests are failing. You can see why they are failing by examining the other columns in the metrics data that show the numbers of requests with different error types such as **ServerTimeoutError**. You should expect to see **Availability** fall temporarily below 100% for reasons such as transient server timeouts while the service moves partitions to better load-balance request; the retry logic in your client application should handle such intermittent conditions. The article [Storage Analytics Logged Operations and Status Messages](#) lists the transaction types that Storage Metrics includes in its **Availability** calculation.

In the [Azure Portal](#), you can add alert rules to notify you if **Availability** for a service falls below a threshold that you specify.

The "Troubleshooting guidance" section of this guide describes some common storage service issues related to availability.

Monitoring performance

To monitor the performance of the storage services, you can use the following metrics from the hourly and minute metrics tables.

- The values in the **AverageE2ELatency** and **AverageServerLatency** columns show the average time the storage service or API operation type is taking to process requests. **AverageE2ELatency** is a measure of end-to-end latency that includes the time taken to read the request and send the response in addition to the time taken to process the request (therefore includes network latency once the request reaches the storage service); **AverageServerLatency** is a measure of just the processing time and therefore excludes any network latency related to communicating with the client. See the section "[Metrics show high AverageE2ELatency and low AverageServerLatency](#)" later in this guide for a discussion of why there might be a significant difference between these two values.
- The values in the **TotalIngress** and **TotalEgress** columns show the total amount of data, in bytes, coming in to and going out of your storage service or through a specific API operation type.
- The values in the **TotalRequests** column show the total number of requests that the storage service or API operation is receiving. **TotalRequests** is the total number of requests that the storage service receives.

Typically, you will monitor for unexpected changes in any of these values as an indicator that you have an issue that requires investigation.

In the [Azure Portal](#), you can add alert rules to notify you if any of the performance metrics for this service fall below or exceed a threshold that you specify.

The "[Troubleshooting guidance](#)" section of this guide describes some common storage service issues related to performance.

Diagnosing storage issues

There are a number of ways that you might become aware of a problem or issue in your application, these include:

- A major failure that causes the application to crash or to stop working.
- Significant changes from baseline values in the metrics you are monitoring as described in the previous section "[Monitoring your storage service](#)."
- Reports from users of your application that some particular operation didn't complete as expected or that some feature is not working.
- Errors generated within your application that appear in log files or through some other notification method.

Typically, issues related to Azure storage services fall into one of four broad categories:

- Your application has a performance issue, either reported by your users, or revealed by changes in the performance metrics.
- There is a problem with the Azure Storage infrastructure in one or more regions.
- Your application is encountering an error, either reported by your users, or revealed by an increase in one of the error count metrics you monitor.
- During development and test, you may be using the local storage emulator; you may encounter some issues that relate specifically to usage of the storage emulator.

The following sections outline the steps you should follow to diagnose and troubleshoot issues in each of these four categories. The section "[Troubleshooting guidance](#)" later in this guide provides more detail for some common issues you may encounter.

Service health issues

Service health issues are typically outside of your control. The [Azure Portal](#) provides information about any ongoing issues with Azure services including storage services. If you opted for Read-Access Geo-Redundant Storage when you created your storage account, then in the event of your data being unavailable in the primary location, your application could switch temporarily to the read-only copy in the secondary location. To do this, your application must be able to switch between using the primary and secondary storage locations, and be able to work in a reduced functionality mode with read-only data. The Azure Storage Client libraries allow you to define

a retry policy that can read from secondary storage in case a read from primary storage fails. Your application also needs to be aware that the data in the secondary location is eventually consistent. For more information, see the blog post [Azure Storage Redundancy Options and Read Access Geo Redundant Storage](#).

Performance issues

The performance of an application can be subjective, especially from a user perspective. Therefore, it is important to have baseline metrics available to help you identify where there might be a performance issue. Many factors might affect the performance of an Azure storage service from the client application perspective. These factors might operate in the storage service, in the client, or in the network infrastructure; therefore it is important to have a strategy for identifying the origin of the performance issue.

After you have identified the likely location of the cause of the performance issue from the metrics, you can then use the log files to find detailed information to diagnose and troubleshoot the problem further.

The section "[Troubleshooting guidance](#)" later in this guide provides more information about some common performance related issues you may encounter.

Diagnosing errors

Users of your application may notify you of errors reported by the client application. Storage Metrics also records counts of different error types from your storage services such as **NetworkError**, **ClientTimeoutError**, or

AuthorizationError. While Storage Metrics only records counts of different error types, you can obtain more detail about individual requests by examining server-side, client-side, and network logs. Typically, the HTTP status code returned by the storage service will give an indication of why the request failed.

NOTE

Remember that you should expect to see some intermittent errors: for example, errors due to transient network conditions, or application errors.

The following resources are useful for understanding storage-related status and error codes:

- [Common REST API Error Codes](#)
- [Blob Service Error Codes](#)
- [Queue Service Error Codes](#)
- [Table Service Error Codes](#)
- [File Service Error Codes](#)

Storage emulator issues

The Azure SDK includes a storage emulator you can run on a development workstation. This emulator simulates most of the behavior of the Azure storage services and is useful during development and test, enabling you to run applications that use Azure storage services without the need for an Azure subscription and an Azure storage account.

The "[Troubleshooting guidance](#)" section of this guide describes some common issues encountered using the storage emulator.

Storage logging tools

Storage Logging provides server-side logging of storage requests in your Azure storage account. For more information about how to enable server-side logging and access the log data, see [Enabling Storage Logging and Accessing Log Data](#).

The Storage Client Library for .NET enables you to collect client-side log data that relates to storage operations performed by your application. For more information, see [Client-side Logging with the .NET Storage Client Library](#).

NOTE

In some circumstances (such as SAS authorization failures), a user may report an error for which you can find no request data in the server-side Storage logs. You can use the logging capabilities of the Storage Client Library to investigate if the cause of the issue is on the client or use network monitoring tools to investigate the network.

Using network logging tools

You can capture the traffic between the client and server to provide detailed information about the data the client and server are exchanging and the underlying network conditions. Useful network logging tools include:

- [Fiddler](#) is a free web debugging proxy that enables you to examine the headers and payload data of HTTP and HTTPS request and response messages. For more information, see [Appendix 1: Using Fiddler to capture HTTP and HTTPS traffic](#).
- [Microsoft Network Monitor \(Netmon\)](#) and [Wireshark](#) are free network protocol analyzers that enable you to view detailed packet information for a wide range of network protocols. For more information about Wireshark, see "[Appendix 2: Using Wireshark to capture network traffic](#)".
- Microsoft Message Analyzer is a tool from Microsoft that supersedes Netmon and that in addition to capturing network packet data, helps you to view and analyze the log data captured from other tools. For more information, see "[Appendix 3: Using Microsoft Message Analyzer to capture network traffic](#)".
- If you want to perform a basic connectivity test to check that your client machine can connect to the Azure storage service over the network, you cannot do this using the standard **ping** tool on the client. However, you can use the **tcping** tool to check connectivity.

In many cases, the log data from Storage Logging and the Storage Client Library will be sufficient to diagnose an issue, but in some scenarios, you may need the more detailed information that these network logging tools can provide. For example, using Fiddler to view HTTP and HTTPS messages enables you to view header and payload data sent to and from the storage services, which would enable you to examine how a client application retries storage operations. Protocol analyzers such as Wireshark operate at the packet level enabling you to view TCP data, which would enable you to troubleshoot lost packets and connectivity issues. Message Analyzer can operate at both HTTP and TCP layers.

End-to-end tracing

End-to-end tracing using a variety of log files is a useful technique for investigating potential issues. You can use the date/time information from your metrics data as an indication of where to start looking in the log files for the detailed information that will help you troubleshoot the issue.

Correlating log data

When viewing logs from client applications, network traces, and server-side storage logging it is critical to be able to correlate requests across the different log files. The log files include a number of different fields that are useful as correlation identifiers. The client request id is the most useful field to use to correlate entries in the different logs. However sometimes, it can be useful to use either the server request id or timestamps. The following sections provide more details about these options.

Client request ID

The Storage Client Library automatically generates a unique client request id for every request.

- In the client-side log that the Storage Client Library creates, the client request id appears in the **Client Request ID** field of every log entry relating to the request.
- In a network trace such as one captured by Fiddler, the client request id is visible in request messages as the **x-ms-client-request-id** HTTP header value.
- In the server-side Storage Logging log, the client request id appears in the Client request ID column.

NOTE

It is possible for multiple requests to share the same client request id because the client can assign this value (although the Storage Client Library assigns a new value automatically). In the case of retries from the client, all attempts share the same client request id. In the case of a batch sent from the client, the batch has a single client request id.

Server request ID

The storage service automatically generates server request ids.

- In the server-side Storage Logging log, the server request id appears the **Request ID header** column.
- In a network trace such as one captured by Fiddler, the server request id appears in response messages as the **x-ms-request-id** HTTP header value.
- In the client-side log that the Storage Client Library creates, the server request id appears in the **Operation Text** column for the log entry showing details of the server response.

NOTE

The storage service always assigns a unique server request id to every request it receives, so every retry attempt from the client and every operation included in a batch has a unique server request id.

If the Storage Client Library throws a **StorageException** in the client, the **RequestInformation** property contains a **RequestResult** object that includes a **ServiceRequestId** property. You can also access a **RequestResult** object from an **OperationContext** instance.

The code sample below demonstrates how to set a custom **ClientRequestId** value by attaching an **OperationContext** object the request to the storage service. It also shows how to retrieve the **ServerRequestId** value from the response message.

```

//Parse the connection string for the storage account.
const string ConnectionString = "DefaultEndpointsProtocol=https;AccountName=account-name;AccountKey=account-key";
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(ConnectionString);
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Create an Operation Context that includes custom ClientRequestId string based on constants defined within the
application along with a Guid.
OperationContext oc = new OperationContext();
oc.ClientRequestId = String.Format("{0} {1} {2} {3}", HOSTNAME, APPNAME, USERID, Guid.NewGuid().ToString());

try
{
    CloudBlobContainer container = blobClient.GetContainerReference("democontainer");
    ICloudBlob blob = container.GetBlobReferenceFromServer("testImage.jpg", null, null, oc);
    var downloadToPath = string.Format("./{0}", blob.Name);
    using (var fs = File.OpenWrite(downloadToPath))
    {
        blob.DownloadToStream(fs, null, null, oc);
        Console.WriteLine("\t Blob downloaded to file: {0}", downloadToPath);
    }
}
catch (StorageException storageException)
{
    Console.WriteLine("Storage exception {0} occurred", storageException.Message);
    // Multiple results may exist due to client side retry logic - each retried operation will have a unique
ServiceRequestId
    foreach (var result in oc.RequestResults)
    {
        Console.WriteLine("HttpStatus: {0}, ServiceRequestId {1}", result.HttpStatusCode,
result.ServiceRequestID);
    }
}

```

Timestamps

You can also use timestamps to locate related log entries, but be careful of any clock skew between the client and server that may exist. You should search plus or minus 15 minutes for matching server-side entries based on the timestamp on the client. Remember that the blob metadata for the blobs containing metrics indicates the time range for the metrics stored in the blob; this is useful if you have many metrics blobs for the same minute or hour.

Troubleshooting guidance

This section will help you with the diagnosis and troubleshooting of some of the common issues your application may encounter when using the Azure storage services. Use the list below to locate the information relevant to your specific issue.

Troubleshooting Decision Tree

Does your issue relate to the performance of one of the storage services?

- Metrics show high AverageE2ELatency and low AverageServerLatency
- Metrics show low AverageE2ELatency and low AverageServerLatency but the client is experiencing high latency
- Metrics show high AverageServerLatency
- You are experiencing unexpected delays in message delivery on a queue

Does your issue relate to the availability of one of the storage services?

- Metrics show an increase in PercentThrottlingError
- Metrics show an increase in PercentTimeoutError
- Metrics show an increase in PercentNetworkError

Is your client application receiving an HTTP 4XX (such as 404) response from a storage service?

- [The client is receiving HTTP 403 \(Forbidden\) messages](#)
- [The client is receiving HTTP 404 \(Not found\) messages](#)
- [The client is receiving HTTP 409 \(Conflict\) messages](#)

Metrics show low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors

Capacity metrics show an unexpected increase in storage capacity usage

You are experiencing unexpected reboots of Virtual Machines that have a large number of attached VHDS

Your issue arises from using the storage emulator for development or test

You are encountering problems installing the Azure SDK for .NET

You have a different issue with a storage service

Metrics show high AverageE2ELatency and low AverageServerLatency

The illustration below from the [Azure Portal](#) monitoring tool shows an example where the **AverageE2ELatency** is significantly higher than the **AverageServerLatency**.

NAME	SOURCE	MIN	MAX	AVG	TOTAL	ALERT RULES
Average E2E Latency	queue	0 ms	139.65 ms	49.47 ms	---	Not Configured
Average Server Latency	queue	0 ms	4.59 ms	2.49 ms	---	Not Configured

Note that the storage service only calculates the metric **AverageE2ELatency** for successful requests and, unlike **AverageServerLatency**, includes the time the client takes to send the data and receive acknowledgement from the storage service. Therefore, a difference between **AverageE2ELatency** and **AverageServerLatency** could be either due to the client application being slow to respond, or due to conditions on the network.

NOTE

You can also view **E2ELatency** and **ServerLatency** for individual storage operations in the Storage Logging log data.

Investigating client performance issues

Possible reasons for the client responding slowly include having a limited number of available connections or threads, or being low on resources such as CPU, memory or network bandwidth. You may be able to resolve the issue by modifying the client code to be more efficient (for example by using asynchronous calls to the storage service), or by using a larger Virtual Machine (with more cores and more memory).

For the table and queue services, the Nagle algorithm can also cause high **AverageE2ELatency** as compared to **AverageServerLatency**: for more information see the post [Nagle's Algorithm is Not Friendly towards Small Requests](#). You can disable the Nagle algorithm in code by using the **ServicePointManager** class in the **System.Net** namespace. You should do this before you make any calls to the table or queue services in your application since this does not affect connections that are already open. The following example comes from the **Application_Start** method in a worker role.

```
var storageAccount = CloudStorageAccount.Parse(connStr);
ServicePoint tableServicePoint = ServicePointManager.FindServicePoint(storageAccount.TableEndpoint);
tableServicePoint.UseNagleAlgorithm = false;
ServicePoint queueServicePoint = ServicePointManager.FindServicePoint(storageAccount.QueueEndpoint);
queueServicePoint.UseNagleAlgorithm = false;
```

You should check the client-side logs to see how many requests your client application is submitting, and check for general .NET related performance bottlenecks in your client such as CPU, .NET garbage collection, network utilization, or memory. As a starting point for troubleshooting .NET client applications, see [Debugging, Tracing, and Profiling](#).

Investigating network latency issues

Typically, high end-to-end latency caused by the network is due to transient conditions. You can investigate both transient and persistent network issues such as dropped packets by using tools such as Wireshark or Microsoft Message Analyzer.

For more information about using Wireshark to troubleshoot network issues, see "[Appendix 2: Using Wireshark to capture network traffic](#)."

For more information about using Microsoft Message Analyzer to troubleshoot network issues, see "[Appendix 3: Using Microsoft Message Analyzer to capture network traffic](#)."

Metrics show low AverageE2ELatency and low AverageServerLatency but the client is experiencing high latency

In this scenario, the most likely cause is a delay in the storage requests reaching the storage service. You should investigate why requests from the client are not making it through to the blob service.

One possible reason for the client delaying sending requests is that there are a limited number of available connections or threads.

You should also check whether the client is performing multiple retries, and investigate the reason if this is the case. To determine whether the client is performing multiple retries, you can:

- Examine the Storage Analytics logs. If multiple retries are happening, you will see multiple operations with the same client request ID but with different server request IDs.
- Examine the client logs. Verbose logging will indicate that a retry has occurred.
- Debug your code, and check the properties of the **OperationContext** object associated with the request. If the operation has retried, the **RequestResults** property will include multiple unique server request IDs. You can also check the start and end times for each request. For more information, see the code sample in the section [Server request ID](#).

If there are no issues in the client, you should investigate potential network issues such as packet loss. You can use tools such as Wireshark or Microsoft Message Analyzer to investigate network issues.

For more information about using Wireshark to troubleshoot network issues, see "[Appendix 2: Using Wireshark to capture network traffic](#)."

For more information about using Microsoft Message Analyzer to troubleshoot network issues, see "[Appendix 3: Using Microsoft Message Analyzer to capture network traffic](#)."

Metrics show high AverageServerLatency

In the case of high **AverageServerLatency** for blob download requests, you should use the Storage Logging logs to see if there are repeated requests for the same blob (or set of blobs). For blob upload requests, you should investigate what block size the client is using (for example, blocks less than 64K in size can result in overheads unless the reads are also in less than 64K chunks), and if multiple clients are uploading blocks to the same blob in parallel. You should also check the per-minute metrics for spikes in the number of requests that result in exceeding the per second scalability targets: also see "[Metrics show an increase in PercentTimeoutError](#)."

If you are seeing high **AverageServerLatency** for blob download requests when there are repeated requests the same blob or set of blobs, then you should consider caching these blobs using Azure Cache or the Azure Content Delivery Network (CDN). For upload requests, you can improve the throughput by using a larger block size. For queries to tables, it is also possible to implement client-side caching on clients that perform the same query operations and where the data doesn't change frequently.

High **AverageServerLatency** values can also be a symptom of poorly designed tables or queries that result in scan operations or that follow the append/prepend anti-pattern. See "[Metrics show an increase in PercentThrottlingError](#)" for more information.

NOTE

You can find a comprehensive checklist performance checklist here: [Microsoft Azure Storage Performance and Scalability Checklist](#).

You are experiencing unexpected delays in message delivery on a queue

If you are experiencing a delay between the time an application adds a message to a queue and the time it becomes available to read from the queue, then you should take the following steps to diagnose the issue:

- Verify the application is successfully adding the messages to the queue. Check that the application is not retrying the **AddMessage** method several times before succeeding. The Storage Client Library logs will show any repeated retries of storage operations.
- Verify there is no clock skew between the worker role that adds the message to the queue and the worker role that reads the message from the queue that makes it appear as if there is a delay in processing.
- Check if the worker role that reads the messages from the queue is failing. If a queue client calls the **GetMessage** method but fails to respond with an acknowledgement, the message will remain invisible on the queue until the **InvisibilityTimeout** period expires. At this point, the message becomes available for processing again.
- Check if the queue length is growing over time. This can occur if you do not have sufficient workers available to process all of the messages that other workers are placing on the queue. You should also check the metrics to see if delete requests are failing and the dequeue count on messages, which might indicate repeated failed attempts to delete the message.
- Examine the Storage Logging logs for any queue operations that have higher than expected **E2ELatency** and **ServerLatency** values over a longer period of time than usual.

Metrics show an increase in PercentThrottlingError

Throttling errors occur when you exceed the scalability targets of a storage service. The storage service does this to ensure that no single client or tenant can use the service at the expense of others. For more information, see [Azure Storage Scalability and Performance Targets](#) for details on scalability targets for storage accounts and performance targets for partitions within storage accounts.

If the **PercentThrottlingError** metric shows an increase in the percentage of requests that are failing with a throttling error, you need to investigate one of two scenarios:

- [Transient increase in PercentThrottlingError](#)
- [Permanent increase in PercentThrottlingError error](#)

An increase in **PercentThrottlingError** often occurs at the same time as an increase in the number of storage requests, or when you are initially load testing your application. This may also manifest itself in the client as "503 Server Busy" or "500 Operation Timeout" HTTP status messages from storage operations.

Transient increase in PercentThrottlingError

If you are seeing spikes in the value of **PercentThrottlingError** that coincide with periods of high activity for the application, you should implement an exponential (not linear) back off strategy for retries in your client: this will

reduce the immediate load on the partition and help your application to smooth out spikes in traffic. For more information about how to implement retry policies using the Storage Client Library, see [Microsoft.WindowsAzure.Storage.RetryPolicies Namespace](#).

NOTE

You may also see spikes in the value of **PercentThrottlingError** that do not coincide with periods of high activity for the application: the most likely cause here is the storage service moving partitions to improve load balancing.

Permanent increase in PercentThrottlingError error

If you are seeing a consistently high value for **PercentThrottlingError** following a permanent increase in your transaction volumes, or when you are performing your initial load tests on your application, then you need to evaluate how your application is using storage partitions and whether it is approaching the scalability targets for a storage account. For example, if you are seeing throttling errors on a queue (which counts as a single partition), then you should consider using additional queues to spread the transactions across multiple partitions. If you are seeing throttling errors on a table, you need to consider using a different partitioning scheme to spread your transactions across multiple partitions by using a wider range of partition key values. One common cause of this issue is the prepend/append anti-pattern where you select the date as the partition key and then all data on a particular day is written to one partition: under load, this can result in a write bottleneck. You should either consider a different partitioning design or evaluate whether using blob storage might be a better solution. You should also check if the throttling is occurring as a result of spikes in your traffic and investigate ways of smoothing your pattern of requests.

If you distribute your transactions across multiple partitions, you must still be aware of the scalability limits set for the storage account. For example, if you used ten queues each processing the maximum of 2,000 1KB messages per second, you will be at the overall limit of 20,000 messages per second for the storage account. If you need to process more than 20,000 entities per second, you should consider using multiple storage accounts. You should also bear in mind that the size of your requests and entities has an impact on when the storage service throttles your clients: if you have larger requests and entities, you may be throttled sooner.

Inefficient query design can also cause you to hit the scalability limits for table partitions. For example, a query with a filter that only selects one percent of the entities in a partition but that scans all the entities in a partition will need to access each entity. Every entity read will count towards the total number of transactions in that partition; therefore, you can easily reach the scalability targets.

NOTE

Your performance testing should reveal any inefficient query designs in your application.

Metrics show an increase in PercentTimeoutError

Your metrics show an increase in **PercentTimeoutError** for one of your storage services. At the same time, the client receives a high volume of "500 Operation Timeout" HTTP status messages from storage operations.

NOTE

You may see timeout errors temporarily as the storage service load balances requests by moving a partition to a new server.

The **PercentTimeoutError** metric is an aggregation of the following metrics: **ClientTimeoutError**, **AnonymousClientTimeoutError**, **SASClientTimeoutError**, **ServerTimeoutError**, **AnonymousServerTimeoutError**, and **SASServerTimeoutError**.

The server timeouts are caused by an error on the server. The client timeouts happen because an operation on the server has exceeded the timeout specified by the client; for example, a client using the Storage Client Library can

set a timeout for an operation by using the **ServerTimeout** property of the **QueueRequestOptions** class.

Server timeouts indicate a problem with the storage service that requires further investigation. You can use metrics to see if you are hitting the scalability limits for the service and to identify any spikes in traffic that might be causing this problem. If the problem is intermittent, it may be due to load-balancing activity in the service. If the problem is persistent and is not caused by your application hitting the scalability limits of the service, you should raise a support issue. For client timeouts, you must decide if the timeout is set to an appropriate value in the client and either change the timeout value set in the client or investigate how you can improve the performance of the operations in the storage service, for example by optimizing your table queries or reducing the size of your messages.

Metrics show an increase in PercentNetworkError

Your metrics show an increase in **PercentNetworkError** for one of your storage services. The

PercentNetworkError metric is an aggregation of the following metrics: **NetworkError**,

AnonymousNetworkError, and **SASNetworkError**. These occur when the storage service detects a network error when the client makes a storage request.

The most common cause of this error is a client disconnecting before a timeout expires in the storage service. You should investigate the code in your client to understand why and when the client disconnects from the storage service. You can also use Wireshark, Microsoft Message Analyzer, or Tcping to investigate network connectivity issues from the client. These tools are described in the [Appendices](#).

The client is receiving HTTP 403 (Forbidden) messages

If your client application is throwing HTTP 403 (Forbidden) errors, a likely cause is that the client is using an expired Shared Access Signature (SAS) when it sends a storage request (although other possible causes include clock skew, invalid keys, and empty headers). If an expired SAS key is the cause, you will not see any entries in the server-side Storage Logging log data. The following table shows a sample from the client-side log generated by the Storage Client Library that illustrates this issue occurring:

SOURCE	VERBOSITY	VERBOSITY	CLIENT REQUEST ID	OPERATION TEXT
Microsoft.WindowsAzure.Storage	Information	3	85d077ab-...	Starting operation with location Primary per location mode PrimaryOnly.
Microsoft.WindowsAzure.Storage	Information	3	85d077ab -...	Starting synchronous request to https://domemaildist.blob.core.windows.net/azureimblobcontainer/blobCreatedViaSASTxt?sv=2014-02-14&sr=c&si=mypolicy&sig=OFnd4Rd7z01flvh%2BmcR6zbudlH2F5Ikm%2FyhNYZEmJNQ%3D&api-version=2014-02-14 .
Microsoft.WindowsAzure.Storage	Information	3	85d077ab -...	Waiting for response.

SOURCE	VERBOSITY	VERBOSITY	CLIENT REQUEST ID	OPERATION TEXT
Microsoft.WindowsAzure.Storage	Warning	2	85d077ab -...	Exception thrown while waiting for response: The remote server returned an error: (403) Forbidden..
Microsoft.WindowsAzure.Storage	Information	3	85d077ab -...	Response received. Status code = 403, Request ID = 9d67c64a-64ed-4b0d-9515-3b14bbcd63d, Content-MD5 = , ETag = .
Microsoft.WindowsAzure.Storage	Warning	2	85d077ab -...	Exception thrown during the operation: The remote server returned an error: (403) Forbidden..
Microsoft.WindowsAzure.Storage	Information	3	85d077ab -...	Checking if the operation should be retried. Retry count = 0, HTTP status code = 403, Exception = The remote server returned an error: (403) Forbidden..
Microsoft.WindowsAzure.Storage	Information	3	85d077ab -...	The next location has been set to Primary, based on the location mode.
Microsoft.WindowsAzure.Storage	Error	1	85d077ab -...	Retry policy did not allow for a retry. Failing with The remote server returned an error: (403) Forbidden.

In this scenario, you should investigate why the SAS token is expiring before the client sends the token to the server:

- Typically, you should not set a start time when you create a SAS for a client to use immediately. If there are small clock differences between the host generating the SAS using the current time and the storage service, then it is possible for the storage service to receive a SAS that is not yet valid.
- You should not set a very short expiry time on a SAS. Again, small clock differences between the host generating the SAS and the storage service can lead to a SAS apparently expiring earlier than anticipated.
- Does the version parameter in the SAS key (for example **sv=2015-04-05**) match the version of the Storage Client Library you are using? We recommend that you always use the latest version of the [Storage Client Library](#).
- If you regenerate your storage access keys, this can invalidate any existing SAS tokens. This may be an issue if you generate SAS tokens with a long expiry time for client applications to cache.

If you are using the Storage Client Library to generate SAS tokens, then it is easy to build a valid token. However, if you are using the Storage REST API and constructing the SAS tokens by hand you should carefully read the topic [Delegating Access with a Shared Access Signature](#).

The client is receiving HTTP 404 (Not found) messages

If the client application receives an HTTP 404 (Not found) message from the server, this implies that the object the client was attempting to use (such as an entity, table, blob, container, or queue) does not exist in the storage service. There are a number of possible reasons for this, such as:

- [The client or another process previously deleted the object](#)
- [A Shared Access Signature \(SAS\) authorization issue](#)
- [Client-side JavaScript code does not have permission to access the object](#)
- [Network failure](#)

The client or another process previously deleted the object

In scenarios where the client is attempting to read, update, or delete data in a storage service it is usually easy to identify in the server-side logs a previous operation that deleted the object in question from the storage service. Very often, the log data shows that another user or process deleted the object. In the server-side Storage Logging log, the operation-type and requested-object-key columns show when a client deleted an object.

In the scenario where a client is attempting to insert an object, it may not be immediately obvious why this results in an HTTP 404 (Not found) response given that the client is creating a new object. However, if the client is creating a blob it must be able to find the blob container, if the client is creating a message it must be able to find a queue, and if the client is adding a row it must be able to find the table.

You can use the client-side log from the Storage Client Library to gain a more detailed understanding of when the client sends specific requests to the storage service.

The following client-side log generated by the Storage Client library illustrates the problem when the client cannot find the container for the blob it is creating. This log includes details of the following storage operations:

REQUEST ID	OPERATION
07b26a5d...	DeleteIfExists method to delete the blob container. Note that this operation includes a HEAD request to check for the existence of the container.
e2d06d78...	CreateIfNotExists method to create the blob container. Note that this operation includes a HEAD request that checks for the existence of the container. The HEAD returns a 404 message but continues.
de8b1c3c...	UploadFromStream method to create the blob. The PUT request fails with a 404 message

Log entries:

REQUEST ID	OPERATION TEXT
07b26a5d...	Starting synchronous request to https://domemaildist.blob.core.windows.net/azuremmblobcontainer .

REQUEST ID	OPERATION TEXT
07b26a5d-...	StringToSign = HEAD.....x-ms-client-request-id:07b26a5d-....x-ms-date:Tue, 03 Jun 2014 10:33:11 GMT.x-ms-version:2014-02-14./domemaildist/azuremmblobcontainer.restype:container.
07b26a5d-...	Waiting for response.
07b26a5d-...	Response received. Status code = 200, Request ID = eeead849-...Content-MD5 = , ETag = "0x8D14D2DC63D059B".
07b26a5d-...	Response headers were processed successfully, proceeding with the rest of the operation.
07b26a5d-...	Downloading response body.
07b26a5d-...	Operation completed successfully.
07b26a5d-...	Starting synchronous request to https://domemaildist.blob.core.windows.net/azuremmblobcontainer .
07b26a5d-...	StringToSign = DELETE.....x-ms-client-request-id:07b26a5d-....x-ms-date:Tue, 03 Jun 2014 10:33:12 GMT.x-ms-version:2014-02-14./domemaildist/azuremmblobcontainer.restype:container.
07b26a5d-...	Waiting for response.
07b26a5d-...	Response received. Status code = 202, Request ID = 6ab2a4cf-..., Content-MD5 = , ETag = .
07b26a5d-...	Response headers were processed successfully, proceeding with the rest of the operation.
07b26a5d-...	Downloading response body.
07b26a5d-...	Operation completed successfully.
e2d06d78-...	Starting asynchronous request to https://domemaildist.blob.core.windows.net/azuremmblobcontainer .
e2d06d78-...	StringToSign = HEAD.....x-ms-client-request-id:e2d06d78-....x-ms-date:Tue, 03 Jun 2014 10:33:12 GMT.x-ms-version:2014-02-14./domemaildist/azuremmblobcontainer.restype:container.
e2d06d78-...	Waiting for response.
de8b1c3c-...	Starting synchronous request to https://domemaildist.blob.core.windows.net/azuremmblobcontainer/blobCreated.txt .

REQUEST ID	OPERATION TEXT
de8b1c3c-...	StringToSign = PUT...64.qCmF+TQLPhq/YYK50mP9ZQ==x-ms-blob-type:BlockBlob.x-ms-client-request-id:de8b1c3c-....x-ms-date:Tue, 03 Jun 2014 10:33:12 GMT.x-ms-version:2014-02-14./domemaildist/azuremmblobcontainer/blobCreated.txt.
de8b1c3c-...	Preparing to write request data.
e2d06d78-...	Exception thrown while waiting for response: The remote server returned an error: (404) Not Found..
e2d06d78-...	Response received. Status code = 404, Request ID = 353ae3bc-..., Content-MD5 = , ETag = .
e2d06d78-...	Response headers were processed successfully, proceeding with the rest of the operation.
e2d06d78-...	Downloading response body.
e2d06d78-...	Operation completed successfully.
e2d06d78-...	Starting asynchronous request to https://domemaildist.blob.core.windows.net/azuremmblobcontainer .
e2d06d78-...	StringToSign = PUT...0.....x-ms-client-request-id:e2d06d78-....x-ms-date:Tue, 03 Jun 2014 10:33:12 GMT.x-ms-version:2014-02-14./domemaildist/azuremmblobcontainer.restype:container.
e2d06d78-...	Waiting for response.
de8b1c3c-...	Writing request data.
de8b1c3c-...	Waiting for response.
e2d06d78-...	Exception thrown while waiting for response: The remote server returned an error: (409) Conflict..
e2d06d78-...	Response received. Status code = 409, Request ID = c27da20e-..., Content-MD5 = , ETag = .
e2d06d78-...	Downloading error response body.
de8b1c3c-...	Exception thrown while waiting for response: The remote server returned an error: (404) Not Found..
de8b1c3c-...	Response received. Status code = 404, Request ID = 0eaeb3e-..., Content-MD5 = , ETag = .
de8b1c3c-...	Exception thrown during the operation: The remote server returned an error: (404) Not Found..

REQUEST ID	OPERATION TEXT
de8b1c3c...	Retry policy did not allow for a retry. Failing with The remote server returned an error: (404) Not Found..
e2d06d78...	Retry policy did not allow for a retry. Failing with The remote server returned an error: (409) Conflict..

In this example, the log shows that the client is interleaving requests from the **CreateIfNotExists** method (request id e2d06d78...) with the requests from the **UploadFromStream** method (de8b1c3c...); this is happening because the client application is invoking these methods asynchronously. You should modify the asynchronous code in the client to ensure that it creates the container before attempting to upload any data to a blob in that container. Ideally, you should create all your containers in advance.

A Shared Access Signature (SAS) authorization issue

If the client application attempts to use a SAS key that does not include the necessary permissions for the operation, the storage service returns an HTTP 404 (Not found) message to the client. At the same time, you will also see a non-zero value for **SASAuthorizationError** in the metrics.

The following table shows a sample server-side log message from the Storage Logging log file:

NAME	VALUE
Request start time	2014-05-30T06:17:48.4473697Z
Operation type	GetBlobProperties
Request status	SASAuthorizationError
HTTP status code	404
Authentication type	Sas
Service type	Blob
Request URL	https://domemaildist.blob.core.windows.net/azureimblobcontainer/blobCreatedViaSAS.txt
	?sv=2014-02-14&sr=c&si=mypolicy&sig=XXXXX&api-version=2014-02-14
Request id header	a1f348d5-8032-4912-93ef-b393e5252a3b
Client request ID	2d064953-8436-4ee0-aa0c-65cb874f7929

You should investigate why your client application is attempting to perform an operation it has not been granted permissions for.

Client-side JavaScript code does not have permission to access the object

If you are using a JavaScript client and the storage service is returning HTTP 404 messages, you check for the following JavaScript errors in the browser:

SEC7120: Origin http://localhost:56309 not found in Access-Control-Allow-Origin header.
SCRIPT7002: XMLHttpRequest: Network Error 0x80070005, Access is denied.

NOTE

You can use the F12 Developer Tools in Internet Explorer to trace the messages exchanged between the browser and the storage service when you are troubleshooting client-side JavaScript issues.

These errors occur because the web browser implements the [same origin policy](#) security restriction that prevents a web page from calling an API in a different domain from the domain the page comes from.

To work around the JavaScript issue, you can configure Cross Origin Resource Sharing (CORS) for the storage service the client is accessing. For more information, see [Cross-Origin Resource Sharing \(CORS\) Support for Azure Storage Services](#).

The following code sample shows how to configure your blob service to allow JavaScript running in the Contoso domain to access a blob in your blob storage service:

```
CloudBlobClient client = new CloudBlobClient(blobEndpoint, new StorageCredentials(accountName, accountKey));
// Set the service properties.
ServiceProperties sp = client.GetServiceProperties();
sp.DefaultServiceVersion = "2013-08-15";
CorsRule cr = new CorsRule();
cr.AllowedHeaders.Add("*");
cr.AllowedMethods = CorsHttpMethods.Get | CorsHttpMethods.Put;
cr.AllowedOrigins.Add("http://www.contoso.com");
cr.ExposedHeaders.Add("x-ms-*");
cr.MaxAgeInSeconds = 5;
sp.Cors.CorsRules.Clear();
sp.Cors.CorsRules.Add(cr);
client.SetServiceProperties(sp);
```

Network Failure

In some circumstances, lost network packets can lead to the storage service returning HTTP 404 messages to the client. For example, when your client application is deleting an entity from the table service you see the client throw a storage exception reporting an "HTTP 404 (Not Found)" status message from the table service. When you investigate the table in the table storage service, you see that the service did delete the entity as requested.

The exception details in the client include the request id (7e84f12d...) assigned by the table service for the request: you can use this information to locate the request details in the server-side storage logs by searching in the **request-id-header** column in the log file. You could also use the metrics to identify when failures such as this occur and then search the log files based on the time the metrics recorded this error. This log entry shows that the delete failed with an "HTTP (404) Client Other Error" status message. The same log entry also includes the request id generated by the client in the **client-request-id** column (813ea74f...).

The server-side log also includes another entry with the same **client-request-id** value (813ea74f...) for a successful delete operation for the same entity, and from the same client. This successful delete operation took place very shortly before the failed delete request.

The most likely cause of this scenario is that the client sent a delete request for the entity to the table service, which succeeded, but did not receive an acknowledgement from the server (perhaps due to a temporary network issue). The client then automatically retried the operation (using the same **client-request-id**), and this retry failed because the entity had already been deleted.

If this problem occurs frequently, you should investigate why the client is failing to receive acknowledgements from the table service. If the problem is intermittent, you should trap the "HTTP (404) Not Found" error and log it in the client, but allow the client to continue.

The client is receiving HTTP 409 (Conflict) messages

The following table shows an extract from the server-side log for two client operations: **DeleteIfExists** followed

immediately by **CreateIfNotExists** using the same blob container name. Note that each client operation results in two requests sent to the server, first a **GetContainerProperties** request to check if the container exists, followed by the **DeleteContainer** or **CreateContainer** request.

TIMESTAMP	OPERATION	RESULT	CONTAINER NAME	CLIENT REQUEST ID
05:10:13.7167225	GetContainerProperties	200	mmcont	c9f52c89-...
05:10:13.8167325	DeleteContainer	202	mmcont	c9f52c89-...
05:10:13.8987407	GetContainerProperties	404	mmcont	bc881924-...
05:10:14.2147723	CreateContainer	409	mmcont	bc881924-...

The code in the client application deletes and then immediately recreates a blob container using the same name: the **CreateIfNotExists** method (Client request ID bc881924-...) eventually fails with the HTTP 409 (Conflict) error. When a client deletes blob containers, tables, or queues there is a brief period before the name becomes available again.

The client application should use unique container names whenever it creates new containers if the delete/recreate pattern is common.

Metrics show low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors

The **PercentSuccess** metric captures the percent of operations that were successful based on their HTTP Status Code. Operations with status codes of 2XX count as successful, whereas operations with status codes in 3XX, 4XX and 5XX ranges are counted as unsuccessful and lower the **PercentSuccess** metric value. In the server-side storage log files, these operations are recorded with a transaction status of **ClientOtherErrors**.

It is important to note that these operations have completed successfully and therefore do not affect other metrics such as availability. Some examples of operations that execute successfully but that can result in unsuccessful HTTP status codes include:

- **ResourceNotFound** (Not Found 404), for example from a GET request to a blob that does not exist.
- **ResouceAlreadyExists** (Conflict 409), for example from a **CreateIfNotExist** operation where the resource already exists.
- **ConditionNotMet** (Not Modified 304), for example from a conditional operation such as when a client sends an **ETag** value and an HTTP **If-None-Match** header to request an image only if it has been updated since the last operation.

You can find a list of common REST API error codes that the storage services return on the page [Common REST API Error Codes](#).

Capacity metrics show an unexpected increase in storage capacity usage

If you see sudden, unexpected changes in capacity usage in your storage account, you can investigate the reasons by first looking at your availability metrics; for example, an increase in the number of failed delete requests might lead to an increase in the amount of blob storage you are using as application specific cleanup operations you might have expected to be freeing up space may not be working as expected (for example, because the SAS tokens used for freeing up space have expired).

You are experiencing unexpected reboots of Azure Virtual Machines that have a large number of attached VHDS

If an Azure Virtual Machine (VM) has a large number of attached VHDS that are in the same storage account, you

could exceed the scalability targets for an individual storage account causing the VM to fail. You should check the minute metrics for the storage account (**TotalRequests/TotalIngress/TotalEgress**) for spikes that exceed the scalability targets for a storage account. See the section "[Metrics show an increase in PercentThrottlingError](#)" for assistance in determining if throttling has occurred on your storage account.

In general, each individual input or output operation on a VHD from a Virtual Machine translates to **Get Page** or **Put Page** operations on the underlying page blob. Therefore, you can use the estimated IOPS for your environment to tune how many VHDs you can have in a single storage account based on the specific behavior of your application. We do not recommend having more than 40 disks in a single storage account. See [Azure Storage Scalability and Performance Targets](#) for details of the current scalability targets for storage accounts, in particular the total request rate and total bandwidth for the type of storage account you are using. If you are exceeding the scalability targets for your storage account, you should place your VHDs in multiple different storage accounts to reduce the activity in each individual account.

Your issue arises from using the storage emulator for development or test

You typically use the storage emulator during development and test to avoid the requirement for an Azure storage account. The common issues that can occur when you are using the storage emulator are:

- [Feature "X" is not working in the storage emulator](#)
- [Error "The value for one of the HTTP headers is not in the correct format" when using the storage emulator](#)
- [Running the storage emulator requires administrative privileges](#)

Feature "X" is not working in the storage emulator

The storage emulator does not support all of the features of the Azure storage services such as the file service. For more information, see [Use the Azure Storage Emulator for Development and Testing](#).

For those features that the storage emulator does not support, use the Azure storage service in the cloud.

Error "The value for one of the HTTP headers is not in the correct format" when using the storage emulator

You are testing your application that use the Storage Client Library against the local storage emulator and method calls such as **CreateIfNotExists** fail with the error message "The value for one of the HTTP headers is not in the correct format." This indicates that the version of the storage emulator you are using does not support the version of the storage client library you are using. The Storage Client Library adds the header **x-ms-version** to all the requests it makes. If the storage emulator does not recognize the value in the **x-ms-version** header, it rejects the request.

You can use the Storage Library Client logs to see the value of the **x-ms-version header** it is sending. You can also see the value of the **x-ms-version header** if you use Fiddler to trace the requests from your client application.

This scenario typically occurs if you install and use the latest version of the Storage Client Library without updating the storage emulator. You should either install the latest version of the storage emulator, or use cloud storage instead of the emulator for development and test.

Running the storage emulator requires administrative privileges

You are prompted for administrator credentials when you run the storage emulator. This only occurs when you are initializing the storage emulator for the first time. After you have initialized the storage emulator, you do not need administrative privileges to run it again.

For more information, see [Use the Azure Storage Emulator for Development and Testing](#). Note that you can also initialize the storage emulator in Visual Studio, which will also require administrative privileges.

You are encountering problems installing the Azure SDK for .NET

When you try to install the SDK, it fails trying to install the storage emulator on your local machine. The installation log contains one of the following messages:

- CAQuietExec: Error: Unable to access SQL instance

- CAQuietExec: Error: Unable to create database

The cause is an issue with existing LocalDB installation. By default, the storage emulator uses LocalDB to persist data when it simulates the Azure storage services. You can reset your LocalDB instance by running the following commands in a command-prompt window before trying to install the SDK.

```
sqllocaldb stop v11.0
sqllocaldb delete v11.0
delete %USERPROFILE%\WAStorageEmulatorDb3*.*
sqllocaldb create v11.0
```

The **delete** command removes any old database files from previous installations of the storage emulator.

You have a different issue with a storage service

If the previous troubleshooting sections do not include the issue you are having with a storage service, you should adopt the following approach to diagnosing and troubleshooting your issue.

- Check your metrics to see if there is any change from your expected base-line behavior. From the metrics, you may be able to determine whether the issue is transient or permanent, and which storage operations the issue is affecting.
- You can use the metrics information to help you search your server-side log data for more detailed information about any errors that are occurring. This information may help you troubleshoot and resolve the issue.
- If the information in the server-side logs is not sufficient to troubleshoot the issue successfully, you can use the Storage Client Library client-side logs to investigate the behavior of your client application, and tools such as Fiddler, Wireshark, and Microsoft Message Analyzer to investigate your network.

For more information about using Fiddler, see "[Appendix 1: Using Fiddler to capture HTTP and HTTPS traffic](#)."

For more information about using Wireshark, see "[Appendix 2: Using Wireshark to capture network traffic](#)."

For more information about using Microsoft Message Analyzer, see "[Appendix 3: Using Microsoft Message Analyzer to capture network traffic](#)."

Appendices

The appendices describe several tools that you may find useful when you are diagnosing and troubleshooting issues with Azure Storage (and other services). These tools are not part of Azure Storage and some are third-party products. As such, the tools discussed in these appendices are not covered by any support agreement you may have with Microsoft Azure or Azure Storage, and therefore as part of your evaluation process you should examine the licensing and support options available from the providers of these tools.

Appendix 1: Using Fiddler to capture HTTP and HTTPS traffic

Fiddler is a useful tool for analyzing the HTTP and HTTPS traffic between your client application and the Azure storage service you are using.

NOTE

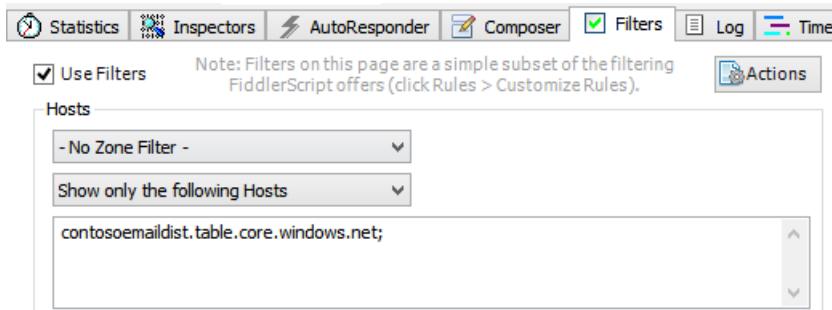
Fiddler can decode HTTPS traffic; you should read the Fiddler documentation carefully to understand how it does this, and to understand the security implications.

This appendix provides a brief walkthrough of how to configure Fiddler to capture traffic between the local machine where you have installed Fiddler and the Azure storage services.

After you have launched Fiddler, it will begin capturing HTTP and HTTPS traffic on your local machine. The following are some useful commands for controlling Fiddler:

- Stop and start capturing traffic. On the main menu, go to **File** and then click **Capture Traffic** to toggle capturing on and off.
- Save captured traffic data. On the main menu, go to **File**, click **Save**, and then click **All Sessions**: this enables you to save the traffic in a Session Archive file. You can reload a Session Archive later for analysis, or send it if requested to Microsoft support.

To limit the amount of traffic that Fiddler captures, you can use filters that you configure in the **Filters** tab. The following screenshot shows a filter that captures only traffic sent to the **contosoemaildist.table.core.windows.net** storage endpoint:

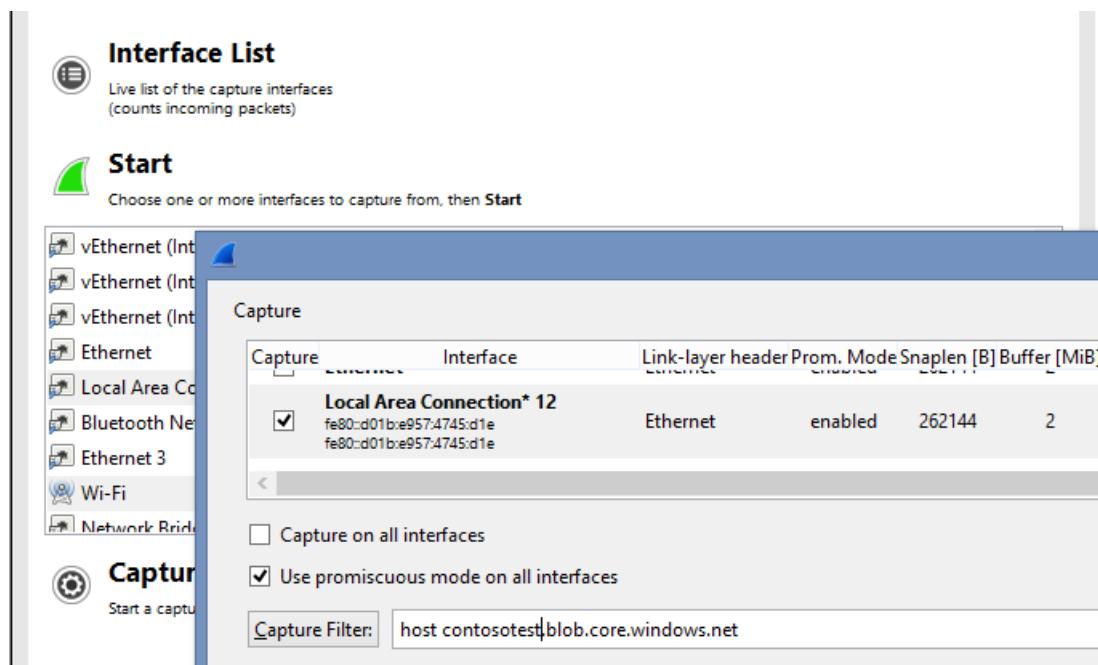


Appendix 2: Using Wireshark to capture network traffic

[Wireshark](#) is a network protocol analyzer that enables you to view detailed packet information for a wide range of network protocols.

The following procedure shows you how to capture detailed packet information for traffic from the local machine where you installed Wireshark to the table service in your Azure storage account.

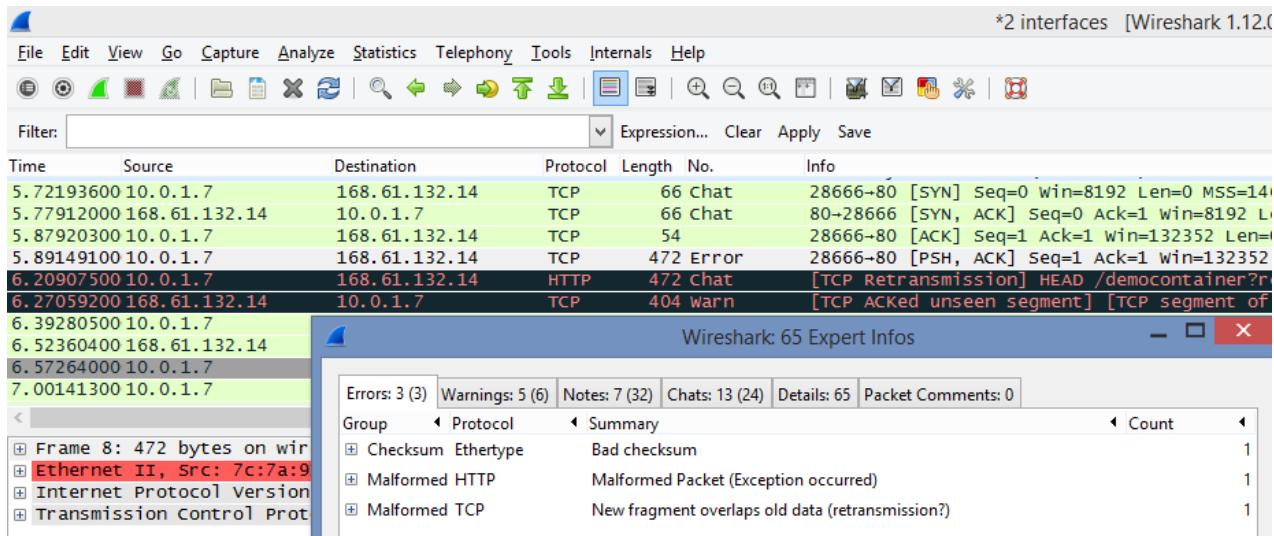
- Launch Wireshark on your local machine.
- In the **Start** section, select the local network interface or interfaces that are connected to the internet.
- Click **Capture Options**.
- Add a filter to the **Capture Filter** textbox. For example, **host contosoemaildist.table.core.windows.net** will configure Wireshark to capture only packets sent to or from the table service endpoint in the **contosoemaildist** storage account. Check out the [complete list of Capture Filters](#).



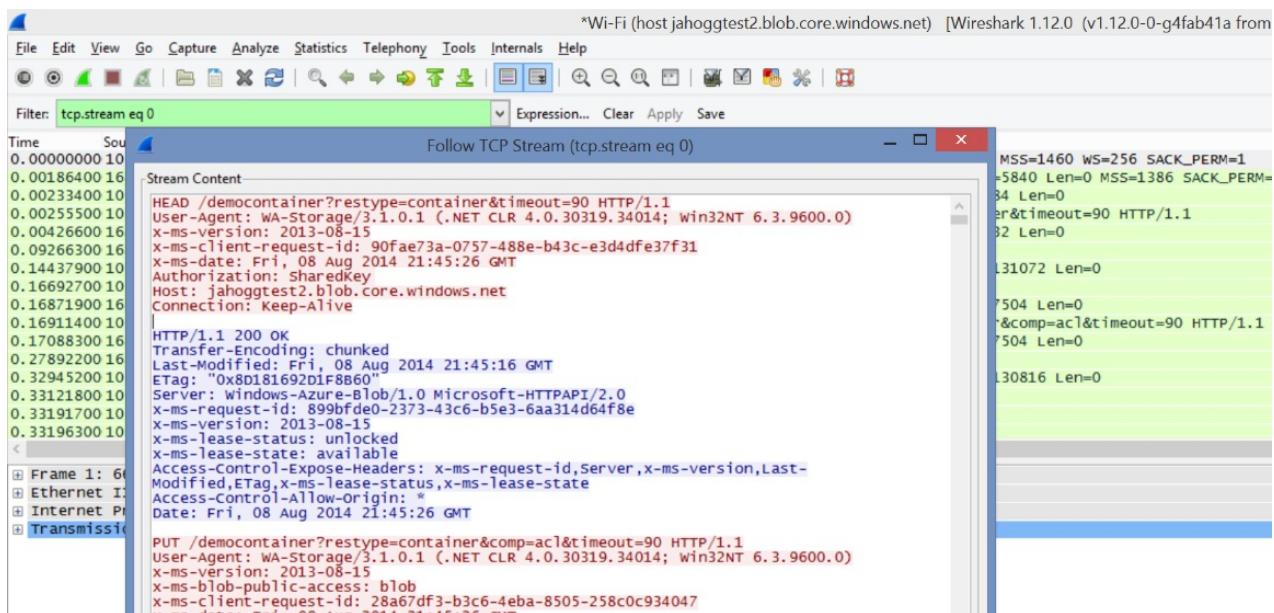
- Click **Start**. Wireshark will now capture all the packets sent to or from the table service endpoint as you use your client application on your local machine.
- When you have finished, on the main menu click **Capture** and then **Stop**.

7. To save the captured data in a Wireshark Capture File, on the main menu click **File** and then **Save**.

WireShark will highlight any errors that exist in the **packetlist** window. You can also use the **Expert Info** window (click **Analyze**, then **Expert Info**) to view a summary of errors and warnings.



You can also choose to view the TCP data as the application layer sees it by right-clicking on the TCP data and selecting **Follow TCP Stream**. This is particularly useful if you captured your dump without a capture filter. For more information, see [Following TCP Streams](#).



NOTE

For more information about using Wireshark, see the [Wireshark Users Guide](#).

Appendix 3: Using Microsoft Message Analyzer to capture network traffic

You can use Microsoft Message Analyzer to capture HTTP and HTTPS traffic in a similar way to Fiddler, and capture network traffic in a similar way to Wireshark.

Configure a web tracing session using Microsoft Message Analyzer

To configure a web tracing session for HTTP and HTTPS traffic using Microsoft Message Analyzer, run the Microsoft Message Analyzer application and then on the **File** menu, click **Capture/Trace**. In the list of available trace scenarios, select **Web Proxy**. Then in the **Trace Scenario Configuration** panel, in the **HostnameFilter** textbox, add the names of your storage endpoints (you can look up these names in the [Azure Portal](#)). For example, if the name of your Azure storage account is **contosodata**, you should add the following to the **HostnameFilter**

textbox:

```
contosodata.blob.core.windows.net contosodata.table.core.windows.net contosodata.queue.core.windows.net
```

NOTE

A space character separates the hostnames.

When you are ready to start collecting trace data, click the **Start With** button.

For more information about the Microsoft Message Analyzer **Web Proxy** trace, see [Microsoft-PEF-WebProxy Provider](#).

The built-in **Web Proxy** trace in Microsoft Message Analyzer is based on Fiddler; it can capture client-side HTTPS traffic and display unencrypted HTTPS messages. The **Web Proxy** trace works by configuring a local proxy for all HTTP and HTTPS traffic that gives it access to unencrypted messages.

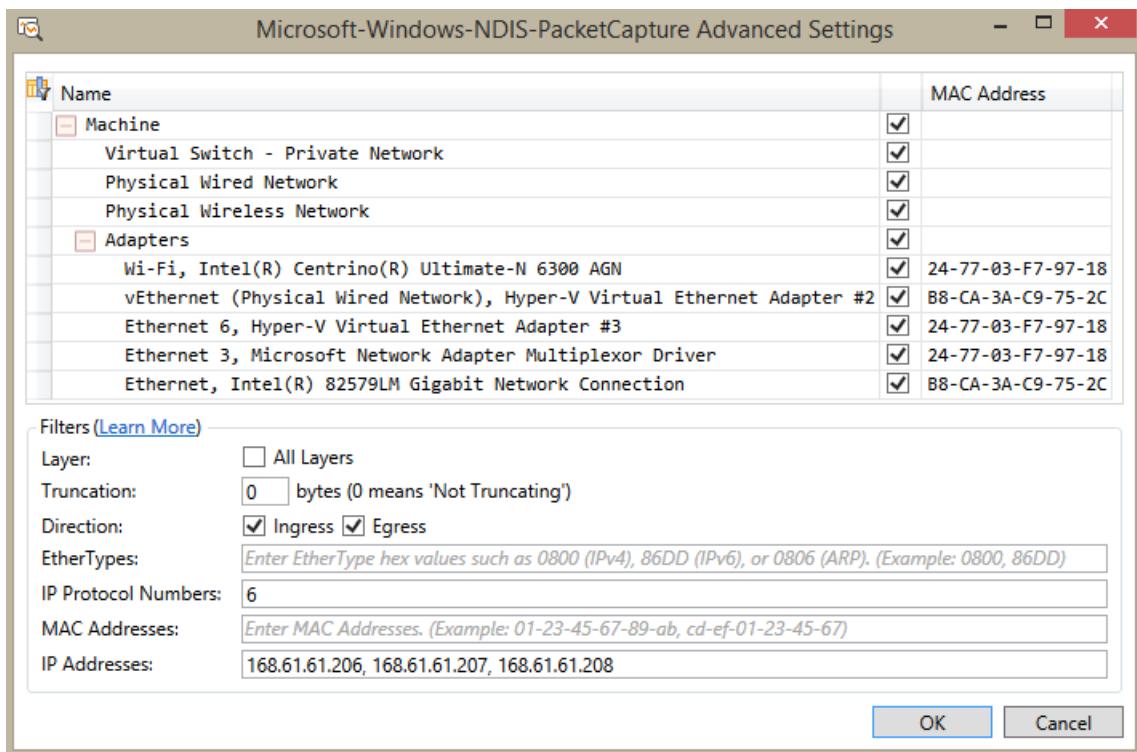
Diagnosing network issues using Microsoft Message Analyzer

In addition to using the Microsoft Message Analyzer **Web Proxy** trace to capture details of the HTTP/HTTPs traffic between the client application and the storage service, you can also use the built-in **Local Link Layer** trace to capture network packet information. This enables you to capture data similar to that which you can capture with Wireshark, and diagnose network issues such as dropped packets.

The following screenshot shows an example **Local Link Layer** trace with some **informational** messages in the **DiagnosisTypes** column. Clicking on an icon in the **DiagnosisTypes** column shows the details of the message. In this example, the server retransmitted message #305 because it did not receive an acknowledgement from the client:

MessageNumber	Type	Level	Message
302	i	Informational	
303	i	Informational	
304	i	Informational	
305	i	Informational	
306	i	Informational	
307	i	Informational	

When you create the trace session in Microsoft Message Analyzer, you can specify filters to reduce the amount of noise in the trace. On the **Capture / Trace** page where you define the trace, click on the **Configure** link next to **Microsoft-Windows-NDIS-PacketCapture**. The following screenshot shows a configuration that filters TCP traffic for the IP addresses of three storage services:



For more information about the Microsoft Message Analyzer Local Link Layer trace, see [Microsoft-PEF-NDIS-PacketCapture Provider](#).

Appendix 4: Using Excel to view metrics and log data

Many tools enable you to download the Storage Metrics data from Azure table storage in a delimited format that makes it easy to load the data into Excel for viewing and analysis. Storage Logging data from Azure blob storage is already in a delimited format that you can load into Excel. However, you will need to add appropriate column headings based in the information at [Storage Analytics Log Format](#) and [Storage Analytics Metrics Table Schema](#).

To import your Storage Logging data into Excel after you download it from blob storage:

- On the **Data** menu, click **From Text**.
- Browse to the log file you want to view and click **Import**.
- On step 1 of the **Text Import Wizard**, select **Delimited**.

On step 1 of the **Text Import Wizard**, select **Semicolon** as the only delimiter and choose double-quote as the **Text qualifier**. Then click **Finish** and choose where to place the data in your workbook.

Appendix 5: Monitoring with Application Insights for Visual Studio Team Services

You can also use the Application Insights feature for Visual Studio Team Services as part of your performance and availability monitoring. This tool can:

- Make sure your web service is available and responsive. Whether your app is a web site or a device app that uses a web service, it can test your URL every few minutes from locations around the world, and let you know if there's a problem.
- Quickly diagnose any performance issues or exceptions in your web service. Find out if CPU or other resources are being stretched, get stack traces from exceptions, and easily search through log traces. If the app's performance drops below acceptable limits, we can send you an email. You can monitor both .NET and Java web services.

You can find more information at [What is Application Insights?](#).

End-to-End Troubleshooting using Azure Storage Metrics and Logging, AzCopy, and Message Analyzer

1/17/2017 • 24 min to read • [Edit on GitHub](#)

Overview

Diagnosing and troubleshooting is a key skill for building and supporting client applications with Microsoft Azure Storage. Due to the distributed nature of an Azure application, diagnosing and troubleshooting errors and performance issues may be more complex than in traditional environments.

In this tutorial, we will demonstrate how to identify client certain errors that may affect performance, and troubleshoot those errors from end-to-end using tools provided by Microsoft and Azure Storage, in order to optimize the client application.

This tutorial provides a hands-on exploration of an end-to-end troubleshooting scenario. For an in-depth conceptual guide to troubleshooting Azure storage applications, see [Monitor, diagnose, and troubleshoot Microsoft Azure Storage](#).

Tools for troubleshooting Azure Storage applications

To troubleshoot client applications using Microsoft Azure Storage, you can use a combination of tools to determine when an issue has occurred and what the cause of the problem may be. These tools include:

- **Azure Storage Analytics.** [Azure Storage Analytics](#) provides metrics and logging for Azure Storage.
 - **Storage metrics** tracks transaction metrics and capacity metrics for your storage account. Using metrics, you can determine how your application is performing according to a variety of different measures. See [Storage Analytics Metrics Table Schema](#) for more information about the types of metrics tracked by Storage Analytics.
 - **Storage logging** logs each request to the Azure Storage services to a server-side log. The log tracks detailed data for each request, including the operation performed, the status of the operation, and latency information. See [Storage Analytics Log Format](#) for more information about the request and response data that is written to the logs by Storage Analytics.

NOTE

Storage accounts with a replication type of Zone-Redundant Storage (ZRS) do not have the metrics or logging capability enabled at this time.

- **Azure Portal.** You can configure metrics and logging for your storage account in the [Azure Portal](#). You can also view charts and graphs that show how your application is performing over time, and configure alerts to notify you if your application performs differently than expected for a specified metric.

See [Monitor a storage account in the Azure Portal](#) for information about configuring monitoring in the Azure Portal.

- **AzCopy.** Server logs for Azure Storage are stored as blobs, so you can use AzCopy to copy the log blobs to a local directory for analysis using Microsoft Message Analyzer. See [Transfer data with the AzCopy Command-Line Utility](#) for more information about AzCopy.
- **Microsoft Message Analyzer.** Message Analyzer is a tool that consumes log files and displays log data in a

visual format that makes it easy to filter, search, and group log data into useful sets that you can use to analyze errors and performance issues. See [Microsoft Message Analyzer Operating Guide](#) for more information about Message Analyzer.

About the sample scenario

For this tutorial, we'll examine a scenario where Azure Storage metrics indicates a low percent success rate for an application that calls Azure storage. The low percent success rate metric (shown as **PercentSuccess** in the [Azure Portal](#) and in the metrics tables) tracks operations that succeed, but that return an HTTP status code that is greater than 299. In the server-side storage log files, these operations are recorded with a transaction status of **ClientOtherErrors**. For more details about the low percent success metric, see [Metrics show low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors](#).

Azure Storage operations may return HTTP status codes greater than 299 as part of their normal functionality. But these errors in some cases indicate that you may be able to optimize your client application for improved performance.

In this scenario, we'll consider a low percent success rate to be anything below 100%. You can choose a different metric level, however, according to your needs. We recommend that during testing of your application, you establish a baseline tolerance for your key performance metrics. For example, you might determine, based on testing, that your application should have a consistent percent success rate of 90%, or 85%. If your metrics data shows that the application is deviating from that number, then you can investigate what may be causing the increase.

For our sample scenario, once we've established that the percent success rate metric is below 100%, we will examine the logs to find the errors that correlate to the metrics, and use them to figure out what is causing the lower percent success rate. We'll look specifically at errors in the 400 range. Then we'll more closely investigate 404 (Not Found) errors.

Some causes of 400-range errors

The examples below shows a sampling of some 400-range errors for requests against Azure Blob Storage, and their possible causes. Any of these errors, as well as errors in the 300 range and the 500 range, can contribute to a low percent success rate.

Note that the lists below are far from complete. See [Status and Error Codes](#) on MSDN for details about general Azure Storage errors and about errors specific to each of the storage services.

Status Code 404 (Not Found) Examples

Occurs when a read operation against a container or blob fails because the blob or container is not found.

- Occurs if a container or blob has been deleted by another client before this request.
- Occurs if you are using an API call that creates the container or blob after checking whether it exists. The `CreateIfNotExists` APIs make a HEAD call first to check for the existence of the container or blob; if it does not exist, a 404 error is returned, and then a second PUT call is made to write the container or blob.

Status Code 409 (Conflict) Examples

- Occurs if you use a Create API to create a new container or blob, without checking for existence first, and a container or blob with that name already exists.
- Occurs if a container is being deleted, and you attempt to create a new container with the same name before the deletion operation is complete.
- Occurs if you specify a lease on a container or blob, and there is already a lease present.

Status Code 412 (Precondition Failed) Examples

- Occurs when the condition specified by a conditional header has not been met.

- Occurs when the lease ID specified does not match the lease ID on the container or blob.

Generate log files for analysis

In this tutorial, we'll use Message Analyzer to work with three different types of log files, although you could choose to work with any one of these:

- The **server log**, which is created when you enable Azure Storage logging. The server log contains data about each operation called against one of the Azure Storage services - blob, queue, table, and file. The server log indicates which operation was called and what status code was returned, as well as other details about the request and response.
- The **.NET client log**, which is created when you enable client-side logging from within your .NET application. The client log includes detailed information about how the client prepares the request and receives and processes the response.
- The **HTTP network trace log**, which collects data on HTTP/HTTPS request and response data, including for operations against Azure Storage. In this tutorial, we'll generate the network trace via Message Analyzer.

Configure server-side logging and metrics

First, we'll need to configure Azure Storage logging and metrics, so that we have data from the client application to analyze. You can configure logging and metrics in a variety of ways - via the [Azure Portal](#), by using PowerShell, or programmatically. See [Enabling Storage Metrics and Viewing Metrics Data](#) and [Enabling Storage Logging and Accessing Log Data](#) on MSDN for details about configuring logging and metrics.

Via the Azure Portal

To configure logging and metrics for your storage account using the [Azure Portal](#), follow the instructions at [Monitor a storage account in the Azure Portal](#).

NOTE

It's not possible to set minute metrics using the Azure Portal. However, we recommend that you do set them for the purposes of this tutorial, and for investigating performance issues with your application. You can set minute metrics using PowerShell as shown below, or programmatically using the storage client library.

Note that the Azure Portal cannot display minute metrics, only hourly metrics.

Via PowerShell

To get started with PowerShell for Azure, see [How to install and configure Azure PowerShell](#).

1. Use the [Add-AzureAccount](#) cmdlet to add your Azure user account to the PowerShell window:

```
Add-AzureAccount
```

2. In the **Sign in to Microsoft Azure** window, type the email address and password associated with your account. Azure authenticates and saves the credential information, and then closes the window.
3. Set the default storage account to the storage account you are using for the tutorial by executing these commands in the PowerShell window:

```
$SubscriptionName = 'Your subscription name'
$StorageAccountName = 'yourstorageaccount'
Set-AzureSubscription -CurrentStorageAccountName $StorageAccountName -SubscriptionName $SubscriptionName
```

4. Enable storage logging for the Blob service:

```
Set-AzureStorageServiceLoggingProperty -ServiceType Blob -LoggingOperations Read,Write,Delete -PassThru  
-RetentionDays 7 -Version 1.0
```

5. Enable storage metrics for the Blob service, making sure to set **-MetricsType** to **Minute**:

```
Set-AzureStorageServiceMetricsProperty -ServiceType Blob -MetricsType Minute -MetricsLevel ServiceAndApi  
-PassThru -RetentionDays 7 -Version 1.0
```

Configure .NET client-side logging

To configure client-side logging for a .NET application, enable .NET diagnostics in the application's configuration file (web.config or app.config). See [Client-side Logging with the .NET Storage Client Library](#) and [Client-side Logging with the Microsoft Azure Storage SDK for Java](#) on MSDN for details.

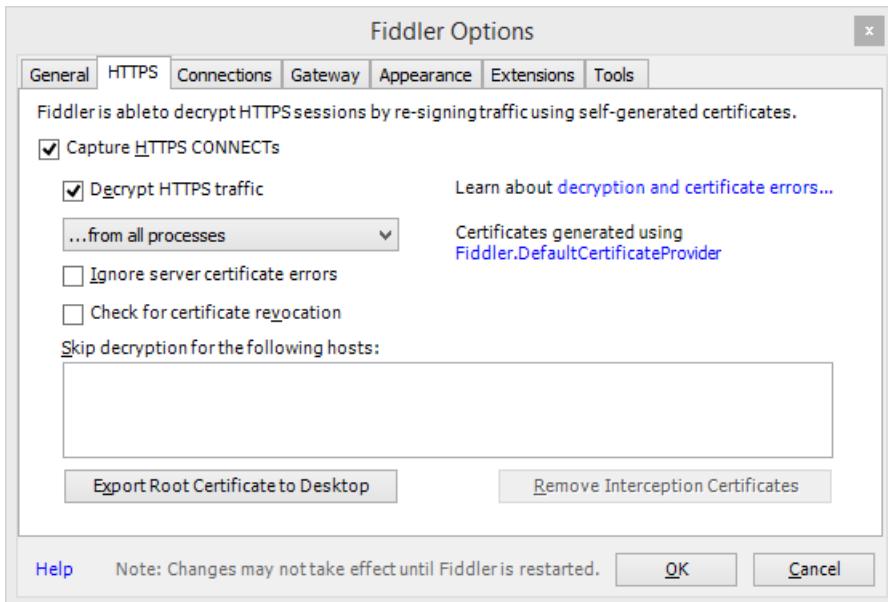
The client-side log includes detailed information about how the client prepares the request and receives and processes the response.

The Storage Client Library stores client-side log data in the location specified in the application's configuration file (web.config or app.config).

Collect a network trace

You can use Message Analyzer to collect an HTTP/HTTPS network trace while your client application is running. Message Analyzer uses [Fiddler](#) on the back end. Before you collect the network trace, we recommend that you configure Fiddler to record unencrypted HTTPS traffic:

1. Install [Fiddler](#).
2. Launch Fiddler.
3. Select **Tools | Fiddler Options**.
4. In the Options dialog, ensure that **Capture HTTPS CONNECTs** and **Decrypt HTTPS Traffic** are both selected, as shown below.



For the tutorial, collect and save a network trace first in Message Analyzer, then create an analysis session to analyze the trace and the logs. To collect a network trace in Message Analyzer:

1. In Message Analyzer, select **File | Quick Trace | Unencrypted HTTPS**.
2. The trace will begin immediately. Select **Stop** to stop the trace so that we can configure it to trace storage traffic only.
3. Select **Edit** to edit the tracing session.

4. Select the **Configure** link to the right of the **Microsoft-Pef-WebProxy** ETW provider.
5. In the **Advanced Settings** dialog, click the **Provider** tab.
6. In the **Hostname Filter** field, specify your storage endpoints, separated by spaces. For example, you can specify your endpoints as follows; change `<storagesample>` to the name of your storage account:

```
storagesample.blob.core.windows.net storagesample.queue.core.windows.net  
storagesample.table.core.windows.net
```

7. Exit the dialog, and click **Restart** to begin collecting the trace with the hostname filter in place, so that only Azure Storage network traffic is included in the trace.

NOTE

After you have finished collecting your network trace, we strongly recommend that you revert the settings that you may have changed in Fiddler to decrypt HTTPS traffic. In the Fiddler Options dialog, deselect the **Capture HTTPS CONNECTS** and **Decrypt HTTPS Traffic** checkboxes.

See [Using the Network Tracing Features](#) on Technet for more details.

Review metrics data in the Azure Portal

Once your application has been running for a period of time, you can review the metrics charts that appear in the [Azure Portal](#) to observe how your service has been performing. First, navigate to your storage account in the Azure Portal and add a chart for the **Success Percentage** metric.

In the Azure Portal, you'll now see **Success Percentage** in the monitoring chart, along with any other metrics you may have added. In the scenario we'll investigate next by analyzing the logs in Message Analyzer, the percent success rate is somewhat below 100%.

For more details on adding metrics to the Monitoring page, see [How to: Add metrics to the metrics table](#).

NOTE

It may take some time for your metrics data to appear in the Azure Portal after you enable storage metrics. This is because hourly metrics for the previous hour are not displayed in the Azure Portal until the current hour has elapsed. Also, minute metrics are not currently displayed in the Azure Portal. So depending on when you enable metrics, it may take up to two hours to see metrics data.

Use AzCopy to copy server logs to a local directory

Azure Storage writes server log data to blobs, while metrics are written to tables. Log blobs are available in the well-known `$logs` container for your storage account. Log blobs are named hierarchically by year, month, day, and hour, so that you can easily locate the range of time you wish to investigate. For example, in the `<storagesample>` account, the container for the log blobs for 01/02/2015, from 8-9 am, is `https://storagesample.blob.core.windows.net/$logs/blob/2015/01/08/0800`. The individual blobs in this container are named sequentially, beginning with `000000.log`.

You can use the AzCopy command-line tool to download these server-side log files to a location of your choice on your local machine. For example, you can use the following command to download the log files for blob operations that took place on January 2, 2015 to the folder `C:\Temp\Logs\Server`; replace `<storageaccountname>` with the name of your storage account, and `<storageaccountkey>` with your account access key:

```
AzCopy.exe /Source:http://<storageaccountname>.blob.core.windows.net/$logs /Dest:C:\Temp\Logs\Server  
/Pattern:"blob/2015/01/02" /SourceKey:<storageaccountkey> /S /V
```

AzCopy is available for download on the [Azure Downloads](#) page. For details about using AzCopy, see [Transfer data with the AzCopy Command-Line Utility](#).

For additional information about downloading server-side logs, see [Download Storage Logging log data](#).

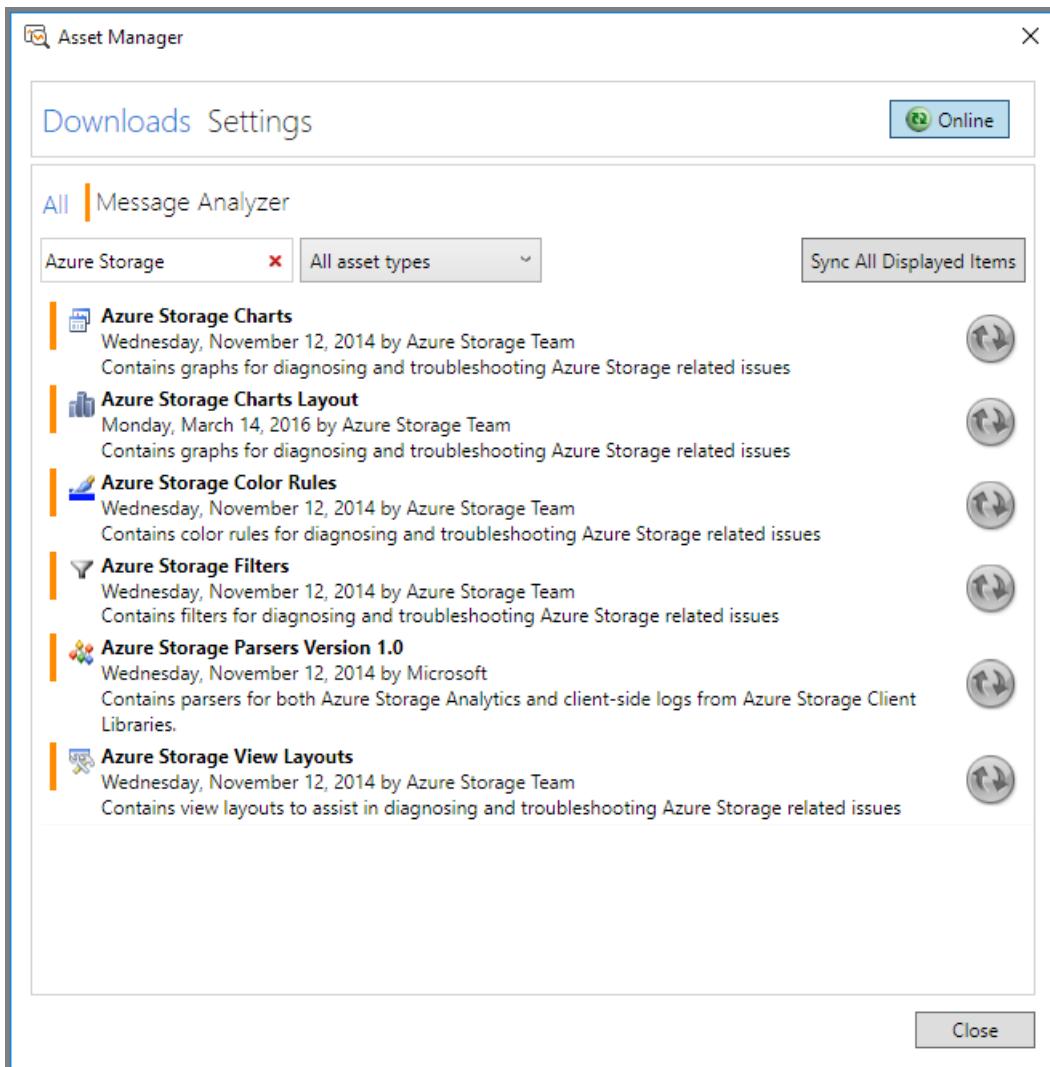
Use Microsoft Message Analyzer to analyze log data

Microsoft Message Analyzer is a tool for capturing, displaying, and analyzing protocol messaging traffic, events, and other system or application messages in troubleshooting and diagnostic scenarios. Message Analyzer also enables you to load, aggregate, and analyze data from log and saved trace files. For more information about Message Analyzer, see [Microsoft Message Analyzer Operating Guide](#).

Message Analyzer includes assets for Azure Storage that help you to analyze server, client, and network logs. In this section, we'll discuss how to use those tools to address the issue of low percent success in the storage logs.

Download and install Message Analyzer and the Azure Storage Assets

1. Download [Message Analyzer](#) from the Microsoft Download Center, and run the installer.
2. Launch Message Analyzer.
3. From the **Tools** menu, select **Asset Manager**. In the **Asset Manager** dialog, select **Downloads**, then filter on **Azure Storage**. You will see the Azure Storage Assets, as shown in the picture below.
4. Click **Sync All Displayed Items** to install the Azure Storage Assets. The available assets include:
 - **Azure Storage Color Rules:** Azure Storage color rules enable you to define special filters that use color, text, and font styles to highlight messages that contain specific information in a trace.
 - **Azure Storage Charts:** Azure Storage charts are predefined charts that graph server log data. Note that to use Azure Storage charts at this time, you may only load the server log into the Analysis Grid.
 - **Azure Storage Parsers:** The Azure Storage parsers parse the Azure Storage client, server, and HTTP logs in order to display them in the Analysis Grid.
 - **Azure Storage Filters:** Azure Storage filters are predefined criteria that you can use to query your data in the Analysis Grid.
 - **Azure Storage View Layouts:** Azure Storage view layouts are predefined column layouts and groupings in the Analysis Grid.
5. Restart Message Analyzer after you've installed the assets.



NOTE

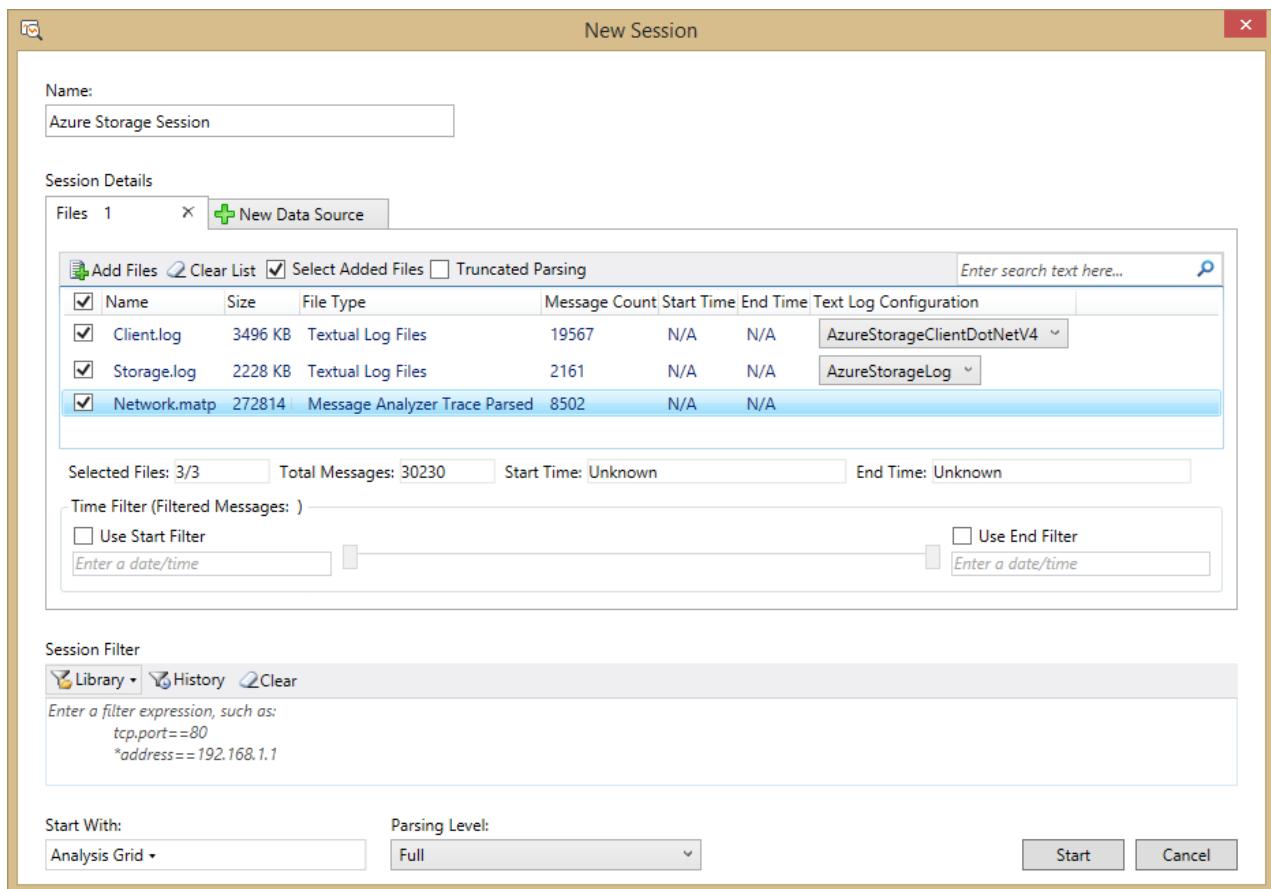
Install all of the Azure Storage assets shown for the purposes of this tutorial.

Import your log files into Message Analyzer

You can import all of your saved log files (server-side, client-side, and network) into a single session in Microsoft Message Analyzer for analysis.

1. On the **File** menu in Microsoft Message Analyzer, click **New Session**, and then click **Blank Session**. In the **New Session** dialog, enter a name for your analysis session. In the **Session Details** panel, click on the **Files** button.
2. To load the network trace data generated by Message Analyzer, click on **Add Files**, browse to the location where you saved your .matp file from your web tracing session, select the .matp file, and click **Open**.
3. To load the server-side log data, click on **Add Files**, browse to the location where you downloaded your server-side logs, select the log files for the time range you want to analyze, and click **Open**. Then, in the **Session Details** panel, set the **Text Log Configuration** drop-down for each server-side log file to **AzureStorageLog** to ensure that Microsoft Message Analyzer can parse the log file correctly.
4. To load the client-side log data, click on **Add Files**, browse to the location where you saved your client-side logs, select the log files you want to analyze, and click **Open**. Then, in the **Session Details** panel, set the **Text Log Configuration** drop-down for each client-side log file to **AzureStorageClientDotNetV4** to ensure that Microsoft Message Analyzer can parse the log file correctly.
5. Click **Start** in the **New Session** dialog to load and parse the log data. The log data displays in the Message Analyzer Analysis Grid.

The picture below shows an example session configured with server, client, and network trace log files.



Note that Message Analyzer loads log files into memory. If you have a large set of log data, you will want to filter it in order to get the best performance from Message Analyzer.

First, determine the time frame that you are interested in reviewing, and keep this time frame as small as possible. In many cases you will want to review a period of minutes or hours at most. Import the smallest set of logs that can meet your needs.

If you still have a large amount of log data, then you may want to specify a session filter to filter your log data before you load it. In the **Session Filter** box, select the **Library** button to choose a predefined filter; for example, choose **Global Time Filter I** from the Azure Storage filters to filter on a time interval. You can then edit the filter criteria to specify the starting and ending timestamp for the interval you want to see. You can also filter on a particular status code; for example, you can choose to load only log entries where the status code is 404.

For more information about importing log data into Microsoft Message Analyzer, see [Retrieving Message Data](#) on TechNet.

Use the client request ID to correlate log file data

The Azure Storage Client Library automatically generates a unique client request ID for every request. This value is written to the client log, the server log, and the network trace, so you can use it to correlate data across all three logs within Message Analyzer. See [Client request ID](#) for additional information about the client request ID.

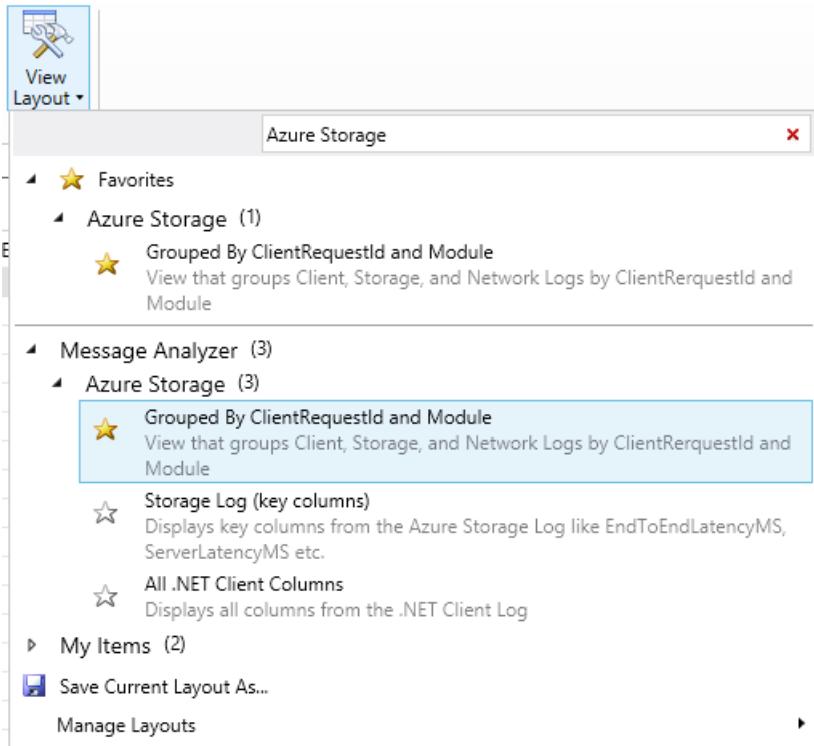
The sections below describe how to use pre-configured and custom layout views to correlate and group data based on the client request ID.

Select a view layout to display in the Analysis Grid

The Storage Assets for Message Analyzer include Azure Storage View Layouts, which are pre-configured views that you can use to display your data with useful groupings and columns for different scenarios. You can also create custom view layouts and save them for reuse.

The picture below shows the **View Layout** menu, available by selecting **View Layout** from the toolbar ribbon. The view layouts for Azure Storage are grouped under the **Azure Storage** node in the menu. You can search for **Azure Storage** in the search box to filter on Azure Storage view layouts only. You can also select the star next to a

view layout to make it a favorite and display it at the top of the menu.



To begin with, select **Grouped by ClientRequestId and Module**. This view layout groups log data from all three logs first by client request ID, then by source log file (or **Module** in Message Analyzer). With this view, you can drill down into a particular client request ID, and see data from all three log files for that client request ID.

The picture below shows this layout view applied to the sample log data, with a subset of columns displayed. You can see that for a particular client request ID, the Analysis Grid displays data from the client log, server log, and network trace.

ClientRequestId		Module			
MessageNumber	Timestamp	TimeElapse	Source	Destination	Module
ClientRequestId (3): 00d98e5a-cc8f-4a4b-8dc1-5a635d9d7265					
Module (9): AzureStorageClientDotNetV4					
17271					AzureStorageClientDotNetV4
17272					AzureStorageClientDotNetV4
17273					AzureStorageClientDotNetV4
17274					AzureStorageClientDotNetV4
17275					AzureStorageClientDotNetV4
17276					AzureStorageClientDotNetV4
17277					AzureStorageClientDotNetV4
17278					AzureStorageClientDotNetV4
17279					AzureStorageClientDotNetV4
Module (1): AzureStorageLog					
1732	2014-10-20T16:39:43.3627051				AzureStorageLog
Module (1): HTTP					
7791	2014-10-20T16:39:42.5689042	0.0319180	Local	photouploadercs.blob.cor...	HTTP
ClientRequestId (3): 0103dde1-8391-468f-ae87-f95d30718b2b					
ClientRequestId (3): 011652df-76b2-4ecd-8e92-819ac91087e6					

NOTE

Different log files have different columns, so when data from multiple log files is displayed in the Analysis Grid, some columns may not contain any data for a given row. For example, in the picture above, the client log rows do not show any data for the **Timestamp**, **TimeElapsed**, **Source**, and **Destination** columns, because these columns do not exist in the client log, but do exist in the network trace. Similarly, the **Timestamp** column displays timestamp data from the server log, but no data is displayed for the **TimeElapsed**, **Source**, and **Destination** columns, which are not part of the server log.

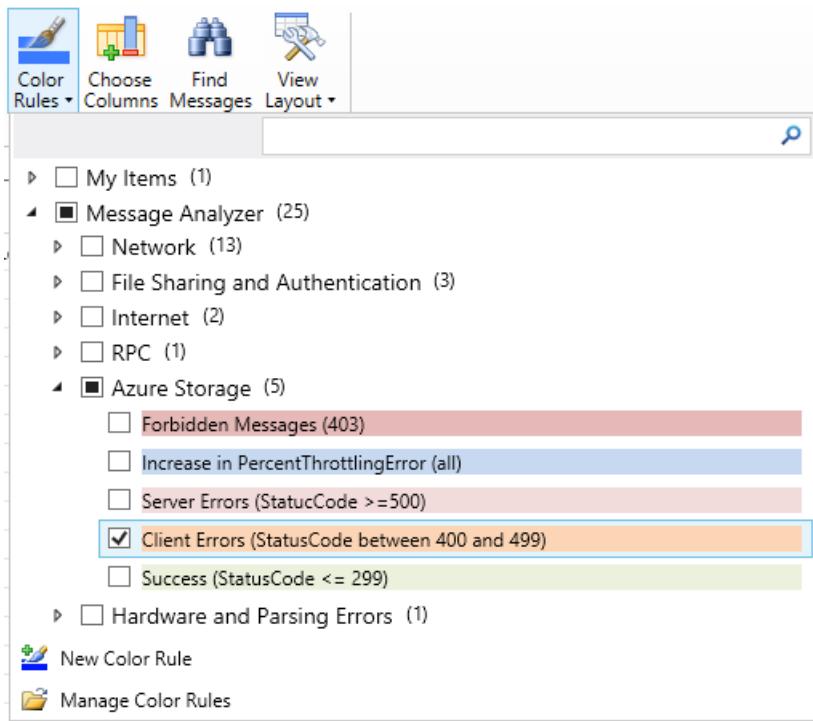
In addition to using the Azure Storage view layouts, you can also define and save your own view layouts. You can

select other desired fields for grouping data and save the grouping as part of your custom layout as well.

Apply color rules to the Analysis Grid

The Storage Assets also include color rules, which offer a visual means to identify different types of errors in the Analysis Grid. The predefined color rules apply to HTTP errors, so they appear only for the server log and network trace.

To apply color rules, select **Color Rules** from the toolbar ribbon. You'll see the Azure Storage color rules in the menu. For the tutorial, select **Client Errors (StatusCode between 400 and 499)**, as shown in the picture below.



In addition to using the Azure Storage color rules, you can also define and save your own color rules.

Group and filter log data to find 400-range errors

Next, we'll group and filter the log data to find all errors in the 400 range.

1. Locate the **StatusCode** column in the Analysis Grid, right-click the column heading, and select **Group**.
2. Next, group on the **ClientRequestId** column. You'll see that the data in the Analysis Grid is now organized by status code and by client request ID.
3. Display the View Filter tool window if it is not already displayed. On the toolbar ribbon, select **Tool Windows**, then **View Filter**.
4. To filter the log data to display only 400-range errors, add the following filter criteria to the **View Filter** window, and click **Apply**:

```
(AzureStorageLog.StatusCode >= 400 && AzureStorageLog.StatusCode <=499) || (HTTP.StatusCode >= 400 && HTTP.StatusCode <= 499)
```

The picture below shows the results of this grouping and filter. Expanding the **ClientRequestId** field beneath the grouping for status code 409, for example, shows an operation that resulted in that status code.

Start Page Azure Storage Se... X			
Status Code	Client Request ID	Message Number	Timestamp
StatusCode (77): 404			Summary
StatusCode (6): 409			
Client Request ID (2): 0c940cc5-a7b6-4f7c-87af-5e452c6cf90			
+ 4575	2014-10-20T16:38:53.9350134	Operation, Status: The specified blob already exists. (409), PUT http://photouploadercs.blob.c...	
1181	2014-10-20T16:38:54.0141050	PutBlob	
Client Request ID (2): 443aa117-6b6d-4c7c-801d-0919449378a1			
Client Request ID (2): acc91f3e-0ee2-431a-8d4d-618dca19020c			
Client Request ID (2): bfb9d10-f97a-4b97-9728-7a2f4df03656			
Client Request ID (2): c23d2ddf-38ca-4d71-9a07-786bc74bb8d06			
Client Request ID (2): f00f7507-b2b8-4771-9ecd-075513ba8fdb			
StatusCode (37): 412			
Client Request ID (2): 0588814e-38f2-4c58-9087-2717285004d5			
+ 4557	2014-10-20T16:38:53.6224286	Operation, Status: The condition specified using HTTP conditional header(s) is not met. (412),...	
1172	2014-10-20T16:38:53.7010738	PutBlob	
Client Request ID (2): 12869d87-cde1-464a-9324-2e5e0d1b68b7			

After applying this filter, you'll see that rows from the client log are excluded, as the client log does not include a **Status Code** column. To begin with, we'll review the server and network trace logs to locate 404 errors, and then we'll return to the client log to examine the client operations that led to them.

NOTE

You can filter on the **Status Code** column and still display data from all three logs, including the client log, if you add an expression to the filter that includes log entries where the status code is null. To construct this filter expression, use:

```
*StatusCode >= 400 or !*StatusCode
```

This filter returns all rows from the client log and only rows from the server log and HTTP log where the status code is greater than 400. If you apply it to the view layout grouped by client request ID and module, you can search or scroll through the log entries to find ones where all three logs are represented.

Filter log data to find 404 errors

The Storage Assets include predefined filters that you can use to narrow log data to find the errors or trends you are looking for. Next, we'll apply two predefined filters: one that filters the server and network trace logs for 404 errors, and one that filters the data on a specified time range.

1. Display the View Filter tool window if it is not already displayed. On the toolbar ribbon, select **Tool Windows**, then **View Filter**.
2. In the View Filter window, select **Library**, and search on **Azure Storage** to find the Azure Storage filters. Select the filter for **404 (Not Found) messages in all logs**.
3. Display the **Library** menu again, and locate and select the **Global Time Filter**.
4. Edit the timestamps shown in the filter to the range you wish to view. This will help to narrow the range of data to analyze.
5. Your filter should appear similar to the example below. Click **Apply** to apply the filter to the Analysis Grid.

```
((AzureStorageLog.StatusCode == 404 || HTTP.StatusCode == 404)) And  
(#Timestamp >= 2014-10-20T16:36:38 and #Timestamp <= 2014-10-20T16:36:39)
```

ClientRequestId			
MessageNumber	Timestamp	Summary	
ClientRequestId (2): 0b3d195b-aa64-42f3-ad58-c69632080f36		Operation, Status: The specified blob does not exist. (404), GET http://	
+ 3193	2014-10-20T16:36:38.8480664	GetBlob	
481	2014-10-20T16:36:38.9245975	GetBlob	
ClientRequestId (2): 58ca84bb-2953-4946-83af-e7ee014038cf		Operation, Status: The specified blob does not exist. (404), GET http://	
+ 3151	2014-10-20T16:36:38.0440455	GetBlob	
461	2014-10-20T16:36:38.1205171	GetBlob	
ClientRequestId (2): 0dbe2457-2a04-4676-914c-19a52f3cab62			
ClientRequestId (1): 2de8f2e5-5b7c-4092-9b5b-6bb12522bb13			
ClientRequestId (2): 6c9cf7ae-f85a-433f-968f-d32d31ce0538			
ClientRequestId (2): 73dc48c9-d209-48cd-9ed1-6b8ed6d58c2c			
ClientRequestId (2): dcba489ab-cdcf-407a-9fd8-22e02a506319			
ClientRequestId (2): f666f045-7f2f-4913-a770-e148840ff408			

Analyze your log data

Now that you have grouped and filtered your data, you can examine the details of individual requests that generated 404 errors. In the current view layout, the data is grouped by client request ID, then by log source. Since we are filtering on requests where the StatusCode field contains 404, we'll see only the server and network trace data, not the client log data.

The picture below shows a specific request where a Get Blob operation yielded a 404 because the blob did not exist. Note that some columns have been removed from the standard view in order to display the relevant data.

ClientRequestId				Module
MessageNumber	Timestamp	Module	Summary	StatusCode
ClientRequestId (2): 2db35a12-b9f8-42d8-853e-8b60ec18b22d				
ClientRequestId (2): 2de8f2e5-5b7c-4092-9b5b-6bb12522bb13				
ClientRequestId (2): 398bac41-7725-484b-8a69-2a9e48fc669a				
Module (1): AzureStorageLog				
+ 411	2014-10-20T16:36:36.1403190	AzureStorageLog	GetBlob	404
Module (1): HTTP				
+ 3045	2014-10-20T16:36:36.0630581	HTTP	Operation, Status: The specified blob does not exist. (404), GET http:... 404	
ClientRequestId (2): 3a89917e-24ff-4ed9-bd60-991670442cca				
ClientRequestId (2): 3e30de3b-abe1-4237-928d-1384ceb02e26				

Next, we'll correlate this client request ID with the client log data to see what actions the client was taking when the error happened. You can display a new Analysis Grid view for this session to view the client log data, which opens in a second tab:

- First, copy the value of the **ClientRequestId** field to the clipboard. You can do this by selecting either row, locating the **ClientRequestId** field, right-clicking on the data value, and choosing **Copy 'ClientRequestId'**.
- On the toolbar ribbon, select **New Viewer**, then select **Analysis Grid** to open a new tab. The new tab shows all data in your log files, without grouping, filtering, or color rules.
- On the toolbar ribbon, select **View Layout**, then select **All .NET Client Columns** under the **Azure Storage** section. This view layout shows data from the client log as well as the server and network trace logs. By default it is sorted on the **MessageNumber** column.
- Next, search the client log for the client request ID. On the toolbar ribbon, select **Find Messages**, then specify a custom filter on the client request ID in the **Find** field. Use this syntax for the filter, specifying your own client request ID:

```
*ClientRequestId == "398bac41-7725-484b-8a69-2a9e48fc669a"
```

Message Analyzer locates and selects the first log entry where the search criteria matches the client request ID. In the client log, there are several entries for each client request ID, so you may want to group them on the **ClientRequestId** field to make it easier to see them all together. The picture below shows all of the messages in the client log for the specified client request ID.

ClientRequestId						
MessageNumber	User	ClientRequestId	Level	LevelString	Description	
ClientRequestId (11): 39616321-08f2-4ce0-b1a8-45ac1ff182b4						
ClientRequestId (13): 398bac41-7725-484b-8a69-2a9e48fc669a						
ClientRequestId (12): 398bac41-7725-484b-8a69-2a9e48fc669a						
3803	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	Starting operation with location Primary per location mode PrimaryOnly.		
3804	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	Starting synchronous request to http://photouploadercs.blob.core.windows.net/testclde9e5dad9c54fc6b0...		
3805	398bac41-7725-484b-8a69-2a9e48fc669a	4	Verbose	StringToSign = GET.....x-ms-client-request-id:398bac41-7725-484b-8a69-2a9e48fc669a.x-ms-date:...		
3806	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	Waiting for response.		
3807	398bac41-7725-484b-8a69-2a9e48fc669a	2	Warning	Exception thrown while waiting for response: The remote server returned an error: (404) Not Found..		
3808	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	Response received. Status code = 404, Request ID = 793143c6-0001-0029-1c50-74ec56000000, Content-MD5=...		
3809	398bac41-7725-484b-8a69-2a9e48fc669a	2	Warning	Exception thrown during the operation: The remote server returned an error: (404) Not Found..		
3810	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	Checking if the operation should be retried. Retry count = 0, HTTP status code = 404, Retryable exce...		
3811	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	The next location has been set to Primary, based on the location mode.		
3812	398bac41-7725-484b-8a69-2a9e48fc669a	1	Error	Retry policy did not allow for a retry. Failing with The remote server returned an error: (404) Not...		
3045	398bac41-7725-484b-8a69-2a9e48fc669a					
411	398bac41-7725-484b-8a69-2a9e48fc669a					
ClientRequestId (13): 399fe04-555e-4b81-900d-f17804a8f352						
ClientRequestId (9): 39b83db1-8409-4b37-be31-bf38a3fcc86a						
ClientRequestId (11): 39cfc9f20-8a04-43b2-b67d-9785c1c7e031						

Using the data shown in the view layouts in these two tabs, you can analyze the request data to determine what may have caused the error. You can also look at requests that preceded this one to see if a previous event may have led to the 404 error. For example, you can review the client log entries preceding this client request ID to determine whether the blob may have been deleted, or if the error was due to the client application calling a `CreateIfNotExists` API on a container or blob. In the client log, you can find the blob's address in the **Description** field; in the server and network trace logs, this information appears in the **Summary** field.

Once you know the address of the blob that yielded the 404 error, you can investigate further. If you search the log entries for other messages associated with operations on the same blob, you can check whether the client previously deleted the entity.

Analyze other types of storage errors

Now that you are familiar with using Message Analyzer to analyze your log data, you can analyze other types of errors using view layouts, color rules, and searching/filtering. The tables below lists some issues you may encounter and the filter criteria you can use to locate them. For more information on constructing filters and the Message Analyzer filtering language, see [Filtering Message Data](#).

TO INVESTIGATE...	USE FILTER EXPRESSION...	EXPRESSION APPLIES TO LOG (CLIENT, SERVER, NETWORK, ALL)
Unexpected delays in message delivery on a queue	AzureStorageClientDotNetV4.Description contains "Retrying failed operation."	Client
HTTP Increase in PercentThrottlingError	HTTP.Response.StatusCode == 500 HTTP.Response.StatusCode == 503	Network
Increase in PercentTimeoutError	HTTP.Response.StatusCode == 500	Network
Increase in PercentTimeoutError (all)	*StatusCode == 500	All
Increase in PercentNetworkError	AzureStorageClientDotNetV4.EventLog Entry.Level < 2	Client
HTTP 403 (Forbidden) messages	HTTP.Response.StatusCode == 403	Network
HTTP 404 (Not found) messages	HTTP.Response.StatusCode == 404	Network
404 (all)	*StatusCode == 404	All

TO INVESTIGATE...	USE FILTER EXPRESSION...	EXPRESSION APPLIES TO LOG (CLIENT, SERVER, NETWORK, ALL)
Shared Access Signature (SAS) authorization issue	AzureStorageLog.RequestStatus == "SASAuthorizationError"	Network
HTTP 409 (Conflict) messages	HTTP.Response.StatusCode == 409	Network
409 (all)	*StatusCode == 409	All
Low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors	AzureStorageLog.RequestStatus == "ClientOtherError"	Server
Nagle Warning	((AzureStorageLog.EndToEndLatencyMS - AzureStorageLog.ServerLatencyMS) > (AzureStorageLog.ServerLatencyMS * 1.5)) and (AzureStorageLog.RequestPacketSize < 1460) and (AzureStorageLog.EndToEndLatencyMS - AzureStorageLog.ServerLatencyMS >= 200)	Server
Range of time in Server and Network logs	#Timestamp >= 2014-10-20T16:36:38 and #Timestamp <= 2014-10-20T16:36:39	Server, Network
Range of time in Server logs	AzureStorageLog.Timestamp >= 2014-10-20T16:36:38 and AzureStorageLog.Timestamp <= 2014-10-20T16:36:39	Server

Next steps

For more information about troubleshooting end-to-end scenarios in Azure Storage, see these resources:

- [Monitor, diagnose, and troubleshoot Microsoft Azure Storage](#)
- [Storage Analytics](#)
- [Monitor a storage account in the Azure Portal](#)
- [Transfer data with the AzCopy Command-Line Utility](#)
- [Microsoft Message Analyzer Operating Guide](#)

Troubleshoot errors when you delete Azure storage accounts, containers, or VHDs in a Resource Manager deployment

1/17/2017 • 4 min to read • [Edit on GitHub](#)

You might receive errors when you try to delete an Azure storage account, container, or virtual hard disk (VHD) in the [Azure portal](#). This article provides troubleshooting guidance to help resolve the problem in an Azure Resource Manager deployment.

If this article doesn't address your Azure problem, visit the Azure forums on [MSDN](#) and [Stack Overflow](#). You can post your problem on these forums or to @AzureSupport on Twitter. Also, you can file an Azure support request by selecting **Get support** on the [Azure support](#) site.

Symptoms

Scenario 1

When you try to delete a VHD in a storage account in a Resource Manager deployment, you receive the following error message:

Failed to delete blob 'vhds/BlobName.vhd'. Error: There is currently a lease on the blob and no lease ID was specified in the request.

This problem can occur because a virtual machine (VM) has a lease on the VHD that you are trying to delete.

Scenario 2

When you try to delete a container in a storage account in a Resource Manager deployment, you receive the following error message:

Failed to delete storage container 'vhds'. Error: There is currently a lease on the container and no lease ID was specified in the request.

This problem can occur because the container has a VHD that is locked in the lease state.

Scenario 3

When you try to delete a storage account in a Resource Manager deployment, you receive the following error message:

Failed to delete storage account 'StorageAccountName'. Error: The storage account cannot be deleted due to its artifacts being in use.

This problem can occur because the storage account contains a VHD that is in the lease state.

Solution

To resolve these problems, you must identify the VHD that is causing the error and the associated VM. Then, detach the VHD from the VM (for data disks) or delete the VM that is using the VHD (for OS disks). This removes the lease from the VHD and allows it to be deleted.

Step 1: Identify the problem VHD and the associated VM

1. Sign in to the [Azure portal](#).
2. On the **Hub** menu, select **All resources**. Go to the storage account that you want to delete, and then select

Blobs > vhds.

thomasdisks607
Storage account

+ Container Refresh Delete

Search (Ctrl+ /)

Overview (highlighted)

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

SETTINGS

Access keys

Configuration

Essentials

Resource group: thomas

Status: Primary: Available

Location: East Asia

Subscription name: Microsoft Azure Internal Consumption

Subscription ID: deb73ff9-8bc5-4ade-b896-6733c7a96d37

Search containers by prefix

NAME	URL	LAST MODIFIED
vhds	https://thomasdisks607.blob.core.windows.net/vhds	2016-10-11 5:41:48 PM

3. Check the properties of each VHD in the container. Locate the VHD that is in the **Leased** state. Then, determine which VM is using the VHD. Usually, you can determine which VM holds the VHD by checking name of the VHD:

- OS disks generally follow this naming convention: VMNameYYYYMMDDHHMMSS.vhd
- Data disks generally follow this naming convention: VMName-YYYYMMDD-HHMMSS.vhd

vhds

Container

Refresh Delete container Properties Access policy

Location:vhds

Search blobs by prefix (case-sensitive)

NAME	MODIFIED	BL
ThomasVM-20161012-172450.vhd	2016-10-12 5:25:12 PM	Pa
ThomasVM2016911174057.vhd	2016-10-13 11:19:15 AM	Pa
ThomasVM-20161012-172200.vhd	2016-10-12 5:23:12 PM	Pa

The name of the VM that holds this VHD

Blob properties

ThomasVM-20161012-172200.vhd

Download Delete

NAME
ThomasVM-20161012-172200.vhd

URL
https://thomasdisks607.blob.core.windows.net/vhds/ThomasVM-20161012-172200.vhd

LEASE STATUS
Locked

LEASE STATE
Leased

LEASE DURATION

Infinite

Step 2: Remove the lease from the VHD

To delete the VM that is using the VHD (for OS disks):

1. Sign in to the [Azure portal](#).
2. On the **Hub** menu, select **Virtual Machines**.
3. Select the VM that holds a lease on the VHD.
4. Make sure that nothing is actively using the virtual machine, and that you no longer need the virtual machine.
5. At the top of the **VM details** blade, select **Delete**, and then click **Yes** to confirm.
6. The VM should be deleted, but the VHD should be retained. However, the VHD should no longer have a lease on

it. It may take a few minutes for the lease to be released. To verify that the lease is released, go to **All resources** > **Storage Account Name** > **Blobs** > **vhds**. In the **Blob properties** pane, the **Lease Status** value should be **Unlocked**.

To detach the VHD from the VM that is using it (for data disks):

1. Sign in to the [Azure portal](#).
2. On the **Hub** menu, select **Virtual Machines**.
3. Select the VM that holds a lease on the VHD.
4. Select **Disk**s on the **VM details** blade.
5. Select the data disk that holds a lease on the VHD. You can determine which VHD is attached in the disk by checking the URL of the VHD.
6. Determine with certainty that nothing is actively using the data disk.
7. Click **Detach** on the **Disk details** blade.
8. The disk should now be detached from the VM, and the VHD should no longer have a lease on it. It may take a few minutes for the lease to be released. To verify that the lease has been released, go to **All resources** > **Storage Account Name** > **Blobs** > **vhds**. In the **Blob properties** pane, the **Lease Status** value should be **Unlocked**.

What is a lease?

A lease is a lock that can be used to control access to a blob (for example, a VHD). When a blob is leased, only the owners of the lease can access the blob. A lease is important for the following reasons:

- It prevents data corruption if multiple owners try to write to the same portion of the blob at the same time.
- It prevents the blob from being deleted if something is actively using it (for example, a VM).
- It prevents the storage account from being deleted if something is actively using it (for example, a VM).

Next steps

- [Delete a storage account](#)
- [How to break the locked lease of blob storage in Microsoft Azure \(PowerShell\)](#)

Troubleshoot deleting Azure storage accounts, containers, or VHDs in a classic deployment

1/17/2017 • 6 min to read • [Edit on GitHub](#)

You might receive errors when you try to delete the Azure storage account, container, or VHD in the [Azure portal](#) or the [Azure classic portal](#). The issues can be caused by the following circumstances:

- When you delete a VM, the disk and VHD are not automatically deleted. That might be the reason for failure on storage account deletion. We don't delete the disk so that you can use the disk to mount another VM.
- There is still a lease on a disk or the blob that's associated with the disk.
- There is still a VM image that is using a blob, container, or storage account.

If your Azure issue is not addressed in this article, visit the Azure forums on [MSDN and the Stack Overflow](#). You can post your issue on these forums or to @AzureSupport on Twitter. Also, you can file an Azure support request by selecting **Get support** on the [Azure support](#) site.

Symptoms

The following section lists common errors that you might receive when you try to delete the Azure storage accounts, containers, or VHDs.

Scenario 1: Unable to delete a storage account

When you navigate to the storage account in the [Azure portal](#) or [Azure classic portal](#) and select **Delete**, you might see the following error message:

Storage account StorageAccountName contains VM Images. Ensure these VM Images are removed before deleting this storage account.

You might also see this error:

On the Azure portal:

Failed to delete storage account . Unable to delete storage account : 'Storage account has some active image(s) and/or disk(s). Ensure these image(s) and/or disk(s) are removed before deleting this storage account.'

On the Azure classic portal:

Storage account has some active image(s) and/or disk(s), e.g. xxxxxxxx-xxxxxxx-O-209490240936090599. Ensure these image(s) and/or disk(s) are removed before deleting this storage account.

Or

On the Azure portal:

Storage account has 1 container(s) which have an active image and/or disk artifacts. Ensure those artifacts are removed from the image repository before deleting this storage account.

On the Azure classic portal:

Submit Failed Storage account has 1 container(s) which have an active image and/or disk artifacts. Ensure those artifacts are removed from the image repository before deleting this storage account. When you attempt to delete a storage account and there are still active disks associated with it, you will see a message telling you there are active disks that need to be deleted.

Scenario 2: Unable to delete a container

When you try to delete the storage container, you might see the following error:

Failed to delete storage container . Error: 'There is currently a lease on the container and no lease ID was specified in the request.'

Or

The following virtual machine disks use blobs in this container, so the container cannot be deleted:

VirtualMachineDiskName1, VirtualMachineDiskName2, ...

Scenario 3: Unable to delete a VHD

After you delete a VM and then try to delete the blobs for the associated VHDs, you might receive the following message:

Failed to delete blob 'path/XXXXXX-XXXXXX-os-1447379084699.vhd'. Error: 'There is currently a lease on the blob and no lease ID was specified in the request.'

Or

Blob 'BlobName.vhd' is in use as virtual machine disk 'VirtualMachineDiskName', so the blob cannot be deleted.

Solution

To resolve the most common issues, try the following method:

Step 1: Delete any disks that are preventing deletion of the storage account, container, or VHD

1. Switch to the [Azure classic portal](#).
2. Select **VIRTUAL MACHINE > DISKS**.

The screenshot shows the Azure classic portal interface. At the top, there's a navigation bar with 'Microsoft Azure' and a dropdown arrow, followed by a blue button 'Check out the new portal'. Below the navigation bar, there's a sidebar with icons for 'ALL ITEMS', 'WEB APPS' (with 3 items), 'VIRTUAL MACHINES' (with 17 items, highlighted with a red box), and 'MOBILE SERVICES' (with 0 items). The main area is titled 'virtual machines' and has tabs for 'INSTANCES', 'IMAGES', and 'DISKS' (which is currently selected). Below the tabs is a table with columns 'NAME', 'ATTACHED TO', and 'CONT.'.

3. Locate the disks that are associated with the storage account, container, or VHD that you want to delete.

When you check the location of the disk, you will find the associated storage account, container, or VHD.

This screenshot shows the 'virtual machines' section with a specific disk selected. The table has columns 'NAME', '...', and 'LOCATION'. The first row shows a VM name and a URL: 'DC01-DC01-0-201501050926360720 ... https://portalvhds9qh3ln560z74.blob.core.windows.net/vhds/DC01-2015-01-05.vhd'. Three red arrows point to the URL: one to 'Storage account' (blob.core.windows.net), one to 'container' (vhds/DC01-2015-01-05), and one to 'VHD' (DC01-2015-01-05.vhd).

4. Delete the disks use one of the following methods:

- If there is no VM listed on the **Attached To** field of the disk, you can delete the disk directly.

- If the disk is a data disk, follow these steps:
 - a. Check the name of the VM that the disk is attached to.
 - b. Go to **Virtual Machines > Instances**, and then locate the VM.
 - c. Make sure that nothing is actively using the disk.
 - d. Select **Detach Disk** at the bottom of the portal to detach the disk.
 - e. Go to **Virtual Machines > Disks**, and wait for the **Attached To** field to turn blank. This indicates the disk has successfully detached from the VM.
 - f. Select **Delete** at the bottom of **Virtual Machines > Disks** to delete the disk.
- If the disk is an OS disk (the **Contains OS** field has a value like Windows) and attached to a VM, follow these steps to delete the VM. The OS disk cannot be detached, so we have to delete the VM to release the lease.
 - a. Check the name of the Virtual Machine the Data Disk is attached to.
 - b. Go to **Virtual Machines > Instances**, and then select the VM that the disk is attached to.
 - c. Make sure that nothing is actively using the virtual machine, and that you no longer need the virtual machine.
 - d. Select the VM the disk is attached to, then select **Delete > Delete the attached disks**.
 - e. Go to **Virtual Machines > Disks**, and wait for the disk to disappear. It may take a few minutes for this to occur, and you may need to refresh the page.
 - f. If the disk does not disappear, wait for the **Attached To** field to turn blank. This indicates the disk has fully detached from the VM. Then, select the disk, and select **Delete** at the bottom of the page to delete the disk.

NOTE

If a disk is attached to a VM, you will not be able to delete it. Disks are detached from a deleted VM asynchronously. It might take a few minutes after the VM is deleted for this field to clear up.

Step 2: Delete any VM Images that are preventing deletion of the storage account or container

1. Switch to the [Azure classic portal](#).
2. Select **VIRTUAL MACHINE > IMAGES**, and then delete the images that are associated with the storage account, container, or VHD.

After that, try to delete the storage account, container, or VHD again.

WARNING

Be sure to back up anything you want to save before you delete the account. It is not possible to restore a deleted storage account or retrieve any of the content that it contained before deletion. This also holds true for any resources in the account: once you delete a VHD, blob, table, queue, or file, it is permanently deleted. Ensure that the resource is not in use.

About the Stopped (deallocated) status

VMs that were created in the classic deployment model and that have been retained will have the **Stopped (deallocated)** status on either the [Azure portal](#) or [Azure classic portal](#).

Azure classic portal:

virtual machines

INSTANCES IMAGES DISKS

NAME	↑	STATUS
MyMigratedVM	→	Stopped (Deallocated)

Azure portal:

The screenshot shows the Azure portal interface for a virtual machine named "MyMigratedVM". The top navigation bar includes links for "INSTANCES", "IMAGES", and "DISKS". Below the navigation, there is a table with columns "NAME" and "STATUS". A single row is present, showing "MyMigratedVM" and "Stopped (Deallocated)". The main content area displays the VM details, including its name, type ("Virtual machine (classic)"), and a summary bar indicating it is "Stopped (deallocated)". Below the summary bar are several management actions: Settings, Connect, Start, Restart, Stop, Capture, Reset, Remote..., and Delete.

A “Stopped (deallocated)” status releases the computer resources, such as the CPU, memory, and network. The disks, however, are still retained so that you can quickly re-create the VM if necessary. These disks are created on top of VHDs, which are backed by Azure storage. The storage account has these VHDs, and the disks have leases on those VHDs.

Next steps

- [Delete a storage account](#)
- [How to break the locked lease of blob storage in Microsoft Azure \(PowerShell\)](#)

Troubleshooting Azure File storage problems

1/17/2017 • 9 min to read • [Edit on GitHub](#)

This article lists common problems that are related to Microsoft Azure File storage when you connect from Windows and Linux clients. It also provides the possible causes of and resolutions for these problems.

General problems (occur in both Windows and Linux clients)

- [Quota error when trying to open a file](#)
- [Slow performance when you access Azure File storage from Windows or from Linux](#)

Windows client problems

- [Slow performance when you access Azure File storage from Windows 8.1 or Windows Server 2012 R2](#)
- [Error 53 attempting to mount an Azure File Share](#)
- [Net use was successful but I don't see the Azure file share mounted in Windows Explorer](#)
- [My storage account contains "/" and the net use command fails](#)
- [My application/service cannot access mounted Azure Files drive.](#)
- [Additional recommendations to optimize performance](#)

Linux client problems

- [Error "You are copying a file to a destination that does not support encryption" when uploading/copying files to Azure Files](#)
- ["Host is down" error on existing file shares, or the shell hangs when doing list commands on the mount point](#)
- [Mount error 115 when attempting to mount Azure Files on the Linux VM](#)
- [Linux VM experiencing random delays in commands like "ls"](#)

Quota error when trying to open a file

In Windows, you receive error messages that resemble the following:

1816 ERROR_NOT_ENOUGH_QUOTA <--> 0xc0000044

STATUS_QUOTA_EXCEEDED

Not enough quota is available to process this command

Invalid handle value GetLastError: 53

On Linux, you receive error messages that resemble the following:

[permission denied]

Disk quota exceeded

Cause

The problem occurs because you have reached the upper limit of concurrent open handles that are allowed for a file.

Solution

Reduce the number of concurrent open handles by closing some handles, and then retry. For more information, see [Microsoft Azure Storage Performance and Scalability Checklist](#).

Slow performance when accessing File storage from Windows or Linux

- If you don't have a specific minimum I/O size requirement, we recommend that you use 1 MB as the I/O size for optimal performance.
- If you know the final size of a file that you are extending with writes, and your software doesn't have compatibility issues when the not yet written tail on the file containing zeros, then set the file size in advance instead of every write being an extending write.

Slow performance when accessing the File storage from Windows 8.1 or Windows Server 2012 R2

For clients who are running Windows 8.1 or Windows Server 2012 R2, make sure that the hotfix [KB3114025](#) is installed. This hotfix improves the create and close handle performance.

You can run the following script to check whether the hotfix has been installed on:

```
reg query HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\Parameters\Policies
```

If hotfix is installed, the following output is displayed:

**HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\Parameters\Policies
{96c345ef-3cac-477b-8fcf-bea1a564241c} REG_DWORD 0x1**

NOTE

Windows Server 2012 R2 images in Azure Marketplace have the hotfix KB3114025 installed by default starting in December 2015.

Additional recommendations to optimize performance

Never create or open a file for cached I/O that is requesting write access but not read access. That is, when you call **CreateFile()**, never specify only **GENERIC_WRITE**, but always specify **GENERIC_READ | GENERIC_WRITE**. A write-only handle cannot cache small writes locally, even when it is the only open handle for the file. This imposes a severe performance penalty on small writes. Note that the "a" mode to CRT **fopen()** opens a write-only handle.

"Error 53" or "Error 67" when you try to mount or unmount an Azure File Share

This problem can be caused by following conditions:

Cause 1

"System error 53 has occurred. Access is denied." For security reasons, connections to Azure Files shares are blocked if the communication channel isn't encrypted and the connection attempt is not made from the same data center on which Azure File shares reside. Communication channel encryption is not provided if the user's client OS doesn't support SMB encryption. This is indicated by a "System error 53 has occurred. Access is denied" Error message when a user tries to mount a file share from on-premises or from a different data center. Windows 8, Windows Server 2012, and later versions of each negotiate request that includes SMB 3.0, which supports encryption.

Solution for Cause 1

Connect from a client that meets the requirements of Windows 8, Windows Server 2012 or later versions, or that connect from a virtual machine that is on the same data center as the Azure Storage account that is used for the Azure File share.

Cause 2

"System Error 53" or "System Error 67" when you mount an Azure file share can occur if Port 445 outbound communication to Azure Files data center is blocked. Click [here](#) to see the summary of ISPs that allow or disallow access from port 445.

Comcast and some IT organizations block this port. To understand whether this is the reason behind the "System Error 53" message, you can use Portqry to query the TCP:445 endpoint. If the TCP:445 endpoint is displayed as filtered, the TCP port is blocked. Here is an example query:

```
g:\DataDump\Tools\Portqry>PortQry.exe -n [storage account name].file.core.windows.net -p TCP -e 445
```

If the TCP 445 being blocked by a rule along the network path, you will see the following output:

TCP port 445 (microsoft-ds service): FILTERED

For more information on using Portqry, see [Description of the Portqry.exe command-line utility](#).

Solution for Cause 2

Work with your IT organization to open Port 445 outbound to [Azure IP ranges](#).

Cause 3

"System Error 53" can also be received if NTLMv1 communication is enabled on the client. Having NTLMv1 enabled creates a less-secure client. Therefore, communication will be blocked for Azure Files. To verify whether this is the cause of the error, verify that the following registry subkey is set to a value of 3:

HKLM\SYSTEM\CurrentControlSet\Control\Lsa > LmCompatibilityLevel.

For more information, see the [LmCompatibilityLevel](#) topic on TechNet.

Solution for Cause 3

To resolve this issue, revert the LmCompatibilityLevel value in the HKLM\SYSTEM\CurrentControlSet\Control\Lsa registry key to the default value of 3.

Azure Files supports only NTLMv2 authentication. Make sure that Group Policy is applied to the clients. This will prevent this error from occurring. This is also considered to be a security best practice. For more information, see [how to configure clients to use NTLMv2 using Group Policy](#)

The recommended policy setting is **Send NTLMv2 response only**. This corresponds to a registry value of 3. Clients use only NTLMv2 authentication, and they use NTLMv2 session security if the server supports it. Domain controllers accept LM, NTLM, and NTLMv2 authentication.

Net use was successful but don't see the Azure file share mounted in Windows Explorer

Cause

By default, Windows Explorer does not run as Administrator. If you run **net use** from an Administrator command prompt, you map the network drive "As Administrator." Because mapped drives are user-centric, the user account that is logged in does not display the drives if they are mounted under a different user account.

Solution

Mount the share from a non-administrator command line. Alternatively, you can follow [this TechNet topic](#) to configure the **EnableLinkedConnections** registry value.

My storage account contains "/" and the net use command fails

Cause

When the **net use** command is run under Command Prompt (cmd.exe), it's parsed by adding "/" as a command-line option. This causes the drive mapping to fail.

Solution

You can use either of the following steps to work around the issue:

- Use the following PowerShell command:

```
New-SmbMapping -LocalPath y: -RemotePath \\server\share -UserName accountName -Password "password can contain / and \ etc"
```

From a batch file this can be done as

```
Echo new-smbMapping ... | powershell -command -
```

- Put double quotation marks around the key to work around this issue — unless "/" is the first character. If it is, either use the interactive mode and enter your password separately or regenerate your keys to get a key that doesn't start with the forward slash (/) character.

My application/service cannot access mounted Azure Files drive

Cause

Drives are mounted per user. If your application or service is running under a different user account, users won't see the drive.

Solution

Mount drive from the same user account under which the application is. This can be done using tools such as psexec.

Alternatively, you can create a new user that has the same privileges as the network service or system account, and then run **cmdkey** and **net use** under that account. The user name should be the storage account name, and password should be the storage account key. Another option for **net use** is to pass in the storage account name and key in the user name and password parameters of the **net use** command.

After you follow these instructions, you may receive the following error message: "System error 1312 has occurred. A specified logon session does not exist. It may already have been terminated" when you run **net use** for the system/network service account. If this occurs, make sure that the username that is passed to **net use** includes domain information (for example: "[storage account name].file.core.windows.net").

Error "You are copying a file to a destination that does not support encryption"

Cause

Bitlocker-encrypted files can be copied to Azure Files. However, the File storage does not support NTFS EFS. Therefore, you are likely using EFS in this case. If you have files that are encrypted through EFS, a copy operation to the File storage can fail unless the copy command is decrypting a copied file.

Workaround

To copy a file to the File storage, you must first decrypt it. You can do this by using one of the following methods:

- Use **copy /d**.
- Set the following registry key:
 - Path=HKLM\Software\Policies\Microsoft\Windows\System
 - Value type=DWORD
 - Name = CopyFileAllowDecryptedRemoteDestination

- Value = 1

However, note that setting the registry key affects all copy operations to network shares.

"Host is down" error on existing file shares, or the shell hangs when you run list commands on the mount point

Cause

This error occurs on the Linux client when the client has been idle for an extended period of time. When this error occurs, the client disconnects, and the client connection times out.

Solution

This issue is now fixed in the Linux kernel as part of [change set](#), pending backport into Linux distribution.

To work around this issue, sustain the connection and avoid getting into an idle state, keep a file in the Azure File share that you write to periodically. This has to be a write operation, such as rewriting the created/modified date on the file. Otherwise, you might get cached results, and your operation might not trigger the connection.

"Mount error 115" when you try to mount Azure Files on the Linux VM

Cause

Linux distributions do not yet support encryption feature in SMB 3.0. In some distributions, user may receive a "115" error message if they try to mount Azure Files by using SMB 3.0 because of a missing feature.

Solution

If the Linux SMB client that is used does not support encryption, mount Azure Files by using SMB 2.1 from a Linux VM on the same data center as the File storage account.

Linux VM experiencing random delays in commands like "ls"

Cause

This can occur when the mount command does not include the **serverino** option. Without **serverino**, the ls command runs a **stat** on every file.

Solution

Check the **serverino** in your "/etc/fstab" entry:

```
//azureuser.file.core.windows.net/wms/comer on /home/sampledir type cifs  
(rw,nodev,relatime,vers=2.1,sec=ntlmssp,cache=strict,username=xxx,username=X,  
file_mode=0755,dir_mode=0755,serverino,rsize=65536,wszie=65536,actimeo=1)
```

If the **serverino** option is not present, unmount and mount Azure Files again by having the **serverino** option selected.

Learn more

- [Get started with Azure File storage on Windows](#)
- [Get started with Azure File storage on Linux](#)

What to do if an Azure Storage outage occurs

1/17/2017 • 3 min to read • [Edit on GitHub](#)

At Microsoft, we work hard to make sure our services are always available. Sometimes, forces beyond our control impact us in ways that cause unplanned service outages in one or more regions. To help you handle these rare occurrences, we provide the following high-level guidance for Azure Storage services.

How to prepare

It is critical for every customer to prepare their own disaster recovery plan. The effort to recover from a storage outage typically involves both operations personnel and automated procedures in order to reactivate your applications in a functioning state. Please refer to the Azure documentation below to build your own disaster recovery plan:

- [Disaster recovery and high availability for Azure applications](#)
- [Azure resiliency technical guidance](#)
- [Azure Site Recovery service](#)
- [Azure Storage replication](#)
- [Azure Backup service](#)

How to detect

The recommended way to determine the Azure service status is to subscribe to the [Azure Service Health Dashboard](#).

What to do if a Storage outage occurs

If one or more Storage services are temporarily unavailable at one or more regions, there are two options for you to consider. If you desire immediate access to your data, please consider Option 2.

Option 1: Wait for recovery

In this case, no action on your part is required. We are working diligently to restore the Azure service availability. You can monitor the service status on the [Azure Service Health Dashboard](#).

Option 2: Copy data from secondary

If you chose [Read-access geo-redundant storage \(RA-GRS\)](#) (recommended) for your storage accounts, you will have read access to your data from the secondary region. You can use tools such as [AzCopy](#), [Azure PowerShell](#), and the [Azure Data Movement library](#) to copy data from the secondary region into another storage account in an unimpacted region, and then point your applications to that storage account for both read and write availability.

What to expect if a Storage failover occurs

If you chose [Geo-redundant storage \(GRS\)](#) or [Read-access geo-redundant storage \(RA-GRS\)](#) (recommended), Azure Storage will keep your data durable in two regions (primary and secondary). In both regions, Azure Storage constantly maintains multiple replicas of your data.

When a regional disaster affects your primary region, we will first try to restore the service in that region. Dependent upon the nature of the disaster and its impacts, in some rare occasions we may not be able to restore the primary region. At that point, we will perform a geo-failover. The cross-region data replication is an asynchronous process which can involve a delay, so it is possible that changes that have not yet been replicated to

the secondary region may be lost. You can query the “[Last Sync Time](#)” of your storage account to get details on the replication status.

A couple of points regarding the storage geo-failover experience:

- Storage geo-failover will only be triggered by the Azure Storage team – there is no customer action required.
- Your existing storage service endpoints for blobs, tables, queues, and files will remain the same after the failover; the DNS entry will need to be updated to switch from the primary region to the secondary region.
- Before and during the geo-failover, you won’t have write access to your storage account due to the impact of the disaster but you can still read from the secondary if your storage account has been configured as RA-GRS.
- When the geo-failover has been completed and the DNS changes propagated, your read and write access to your storage account will be resumed. You can query “[Last Geo Failover Time](#)” of your storage account to get more details.
- After the failover, your storage account will be fully functioning, but in a “degraded” status, as it is actually hosted in a standalone region with no geo-replication possible. To mitigate this risk, we will restore the original primary region and then do a geo-failback to restore the original state. If the original primary region is unrecoverable, we will allocate another secondary region. For more details on the infrastructure of Azure Storage geo replication, please refer to the article on the Storage team blog about [Redundancy Options and RA-GRS](#).

Best Practices for protecting your data

There are some recommended approaches to back up your storage data on a regular basis.

- VM Disks – Use the [Azure Backup service](#) to back up the VM disks used by your Azure virtual machines.
- Block blobs –Create a [snapshot](#) of each block blob, or copy the blobs to another storage account in another region using [AzCopy](#), [Azure PowerShell](#), or the [Azure Data Movement library](#).
- Tables – use [AzCopy](#) to export the table data into another storage account in another region.
- Files – use [AzCopy](#) or [Azure PowerShell](#) to copy your files to another storage account in another region.

Moving data to and from Azure Storage

1/17/2017 • 2 min to read • [Edit on GitHub](#)

If you want to move on-premises data to Azure Storage (or vice versa), there are a variety of ways to do this. The approach that works best for you will depend on your scenario. This article will provide a quick overview of different scenarios and appropriate offerings for each one.

Building Applications

If you're building an application, developing against the REST API or one of our many client libraries is a great way to move data to and from Azure Storage.

Azure Storage provides rich client libraries for .NET, iOS, Java, Android, Universal Windows Platform (UWP), Xamarin, C++, NodeJS, PHP, Ruby, and Python. The client libraries offer advanced capabilities such as retry logic, logging, and parallel uploads. You can also develop directly against the REST API, which can be called by any language that makes HTTP/HTTPS requests.

See [Get Started with Azure Blob Storage](#) to learn more.

In addition, we also offer the [Azure Storage Data Movement Library](#) which is a library designed for high-performance copying of data to and from Azure. Please refer to our Data Movement Library [documentation](#) to learn more.

Quickly viewing/interacting with your data

If you want an easy way to view your Azure Storage data while also having the ability to upload and download your data, then consider using an Azure Storage Explorer.

Check out our list of [Azure Storage Explorers](#) to learn more.

System Administration

If you require or are more comfortable with a command-line utility (e.g. System Administrators), here are a few options for you to consider:

AzCopy

AzCopy is a Windows command-line utility designed for high-performance copying of data to and from Azure Storage. You can also copy data within a storage account, or between different storage accounts.

See [Transfer data with the AzCopy Command-Line Utility](#) to learn more.

Azure PowerShell

Azure PowerShell is a module that provides cmdlets for managing services on Azure. It's a task-based command-line shell and scripting language designed especially for system administration.

See [Using Azure PowerShell with Azure Storage](#) to learn more.

Azure CLI

Azure CLI provides a set of open source, cross-platform commands for working with Azure services. Azure CLI is available on Windows, OSX, and Linux.

See [Using the Azure CLI with Azure Storage](#) to learn more.

Moving large amounts of data with a slow network

One of the biggest challenges associated with moving large amounts of data is the transfer time. If you want to get data to/from Azure Storage without worrying about networks costs or writing code, then Azure Import/Export is an appropriate solution.

See [Azure Import/Export](#) to learn more.

Backing up your data

If you simply need to backup your data to Azure Storage, Azure Backup is the way to go. This is a powerful solution for backing up on-premises data and Azure VMs.

See [Azure Backup](#) to learn more.

Accessing your data on-premises and from the cloud

If you need a solution for accessing your data on-premises and from the cloud, then you should consider using Azure's hybrid cloud storage solution, StorSimple. This solution consists of a physical StorSimple device that intelligently stores frequently used data on SSDs, occasionally used data on HDDs, and inactive/backup/archival data on Azure Storage.

See [StorSimple](#) to learn more.

Recovering your data

When you have on-premises workloads and applications, you'll need a solution that allows your business to continue running in the event of a disaster. Azure Site Recovery handles replication, failover, and recovery of virtual machines and physical servers. Replicated data is stored in Azure Storage, allowing you to eliminate the need for a secondary on-site datacenter.

See [Azure Site Recovery](#) to learn more.

Transfer data with the AzCopy Command-Line Utility

1/17/2017 • 27 min to read • [Edit on GitHub](#)

Overview

AzCopy is a Windows command-line utility designed for copying data to and from Microsoft Azure Blob, File, and Table storage using simple commands with optimal performance. You can copy data from one object to another within your storage account, or between storage accounts.

NOTE

This guide assumes that you are already familiar with [Azure Storage](#). If not, reading the [Introduction to Azure Storage](#) documentation will be helpful. Most importantly, you will need to [create a Storage account](#) in order to start using AzCopy.

Download and install AzCopy

Windows

Download the [latest version of AzCopy](#).

Mac/Linux

AzCopy is not available for Mac/Linux OSs. However, Azure CLI is a suitable alternative for copying data to and from Azure Storage. Read [Using the Azure CLI with Azure Storage](#) to learn more.

Writing your first AzCopy command

The basic syntax for AzCopy commands is:

```
AzCopy /Source:<source> /Dest:<destination> [Options]
```

Open a command window and navigate to the AzCopy installation directory on your computer - where the `AzCopy.exe` executable is located. If desired, you can add the AzCopy installation location to your system path. By default, AzCopy is installed to `%ProgramFiles(x86)%\Microsoft SDKs\Azure\AzCopy` or `%ProgramFiles%\Microsoft SDKs\Azure\AzCopy`.

The following examples demonstrate a variety of scenarios for copying data to and from Microsoft Azure Blobs, Files, and Tables. Refer to the [AzCopy Parameters](#) section for a detailed explanation of the parameters used in each sample.

Blob: Download

Download single blob

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer /Dest:C:\myfolder /SourceKey:key  
/Pattern:"abc.txt"
```

Note that if the folder `C:\myfolder` does not exist, AzCopy will create it and download `abc.txt` into the new folder.

Download single blob from secondary region

```
AzCopy /Source:https://myaccount-secondary.blob.core.windows.net/mynewcontainer /Dest:C:\myfolder  
/SourceKey:key /Pattern:abc.txt
```

Note that you must have read-access geo-redundant storage enabled.

Download all blobs

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer /Dest:C:\myfolder /SourceKey:key /S
```

Assume the following blobs reside in the specified container:

```
abc.txt  
abc1.txt  
abc2.txt  
vd1\abc.txt  
vd1\abcd.txt
```

After the download operation, the directory `C:\myfolder` will include the following files:

```
C:\myfolder\abc.txt  
C:\myfolder\abc1.txt  
C:\myfolder\abc2.txt  
C:\myfolder\vd1\abc.txt  
C:\myfolder\vd1\abcd.txt
```

If you do not specify option `/S`, no blobs will be downloaded.

Download blobs with specified prefix

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer /Dest:C:\myfolder /SourceKey:key  
/Pattern:a /S
```

Assume the following blobs reside in the specified container. All blobs beginning with the prefix `a` will be downloaded:

```
abc.txt  
abc1.txt  
abc2.txt  
xyz.txt  
vd1\abc.txt  
vd1\abcd.txt
```

After the download operation, the folder `C:\myfolder` will include the following files:

```
C:\myfolder\abc.txt  
C:\myfolder\abc1.txt  
C:\myfolder\abc2.txt
```

The prefix applies to the virtual directory, which forms the first part of the blob name. In the example shown above, the virtual directory does not match the specified prefix, so it is not downloaded. In addition, if the option `/S` is not specified, AzCopy will not download any blobs.

Set the last-modified time of exported files to be same as the source blobs

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer /Dest:C:\myfolder /SourceKey:key /MT
```

You can also exclude blobs from the download operation based on their last-modified time. For example, if you want to exclude blobs whose last modified time is the same or newer than the destination file, add the `/XN` option:

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer /Dest:C:\myfolder /SourceKey:key /MT  
/XN
```

Or if you want to exclude blobs whose last modified time is the same or older than the destination file, add the `/XO` option:

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer /Dest:C:\myfolder /SourceKey:key /MT  
/XO
```

Blob: Upload

Upload single file

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.blob.core.windows.net/mycontainer /DestKey:key  
/Pattern:"abc.txt"
```

If the specified destination container does not exist, AzCopy will create it and upload the file into it.

Upload single file to virtual directory

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.blob.core.windows.net/mycontainer/vd /DestKey:key  
/Pattern:abc.txt
```

If the specified virtual directory does not exist, AzCopy will upload the file to include the virtual directory in its name (*e.g.*, `vd/abc.txt` in the example above).

Upload all files

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.blob.core.windows.net/mycontainer /DestKey:key /S
```

Specifying option `/S` uploads the contents of the specified directory to Blob storage recursively, meaning that all subfolders and their files will be uploaded as well. For instance, assume the following files reside in folder `C:\myfolder`:

```
C:\myfolder\abc.txt  
C:\myfolder\abc1.txt  
C:\myfolder\abc2.txt  
C:\myfolder\subfolder\a.txt  
C:\myfolder\subfolder\abcd.txt
```

After the upload operation, the container will include the following files:

```
abc.txt  
abc1.txt  
abc2.txt  
subfolder\a.txt  
subfolder\abcd.txt
```

If you do not specify option `/S`, AzCopy will not upload recursively. After the upload operation, the container will include the following files:

```
abc.txt  
abc1.txt  
abc2.txt
```

Upload files matching specified pattern

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.blob.core.windows.net/mycontainer /DestKey:key  
/Pattern:a* /S
```

Assume the following files reside in folder `C:\myfolder`:

```
C:\myfolder\abc.txt  
C:\myfolder\abc1.txt  
C:\myfolder\abc2.txt  
C:\myfolder\xyz.txt  
C:\myfolder\subfolder\a.txt  
C:\myfolder\subfolder\abcd.txt
```

After the upload operation, the container will include the following files:

```
abc.txt  
abc1.txt  
abc2.txt  
subfolder\a.txt  
subfolder\abcd.txt
```

If you do not specify option `/S`, AzCopy will only upload blobs that don't reside in a virtual directory:

```
C:\myfolder\abc.txt  
C:\myfolder\abc1.txt  
C:\myfolder\abc2.txt
```

Specify the MIME content type of a destination blob

By default, AzCopy sets the content type of a destination blob to `application/octet-stream`. Beginning with version 3.1.0, you can explicitly specify the content type via the option `/SetContentType:[content-type]`. This syntax sets the content type for all blobs in an upload operation.

```
AzCopy /Source:C:\myfolder\ /Dest:https://myaccount.blob.core.windows.net/myContainer/ /DestKey:key  
/Pattern:ab /SetContentType:video/mp4
```

If you specify `/SetContentType` without a value, then AzCopy will set each blob or file's content type according to its file extension.

```
AzCopy /Source:C:\myfolder\ /Dest:https://myaccount.blob.core.windows.net/myContainer/ /DestKey:key  
/Pattern:ab /SetContentType
```

Blob: Copy

Copy single blob within Storage account

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer1  
/Dest:https://myaccount.blob.core.windows.net/mycontainer2 /SourceKey:key /DestKey:key /Pattern:abc.txt
```

When you copy a blob within a Storage account, a [server-side copy](#) operation is performed.

Copy single blob across Storage accounts

```
AzCopy /Source:https://sourceaccount.blob.core.windows.net/mycontainer1  
/Dest:https://destaccount.blob.core.windows.net/mycontainer2 /SourceKey:key1 /DestKey:key2  
/Pattern:abc.txt
```

When you copy a blob across Storage accounts, a [server-side copy](#) operation is performed.

Copy single blob from secondary region to primary region

```
AzCopy /Source:https://myaccount1-secondary.blob.core.windows.net/mynewcontainer1  
/Dest:https://myaccount2.blob.core.windows.net/mynewcontainer2 /SourceKey:key1 /DestKey:key2  
/Pattern:abc.txt
```

Note that you must have read-access geo-redundant storage enabled.

Copy single blob and its snapshots across Storage accounts

```
AzCopy /Source:https://sourceaccount.blob.core.windows.net/mycontainer1  
/Dest:https://destaccount.blob.core.windows.net/mycontainer2 /SourceKey:key1 /DestKey:key2  
/Pattern:abc.txt /Snapshot
```

After the copy operation, the target container will include the blob and its snapshots. Assuming the blob in the example above has two snapshots, the container will include the following blob and snapshots:

```
abc.txt  
abc (2013-02-25 080757).txt  
abc (2014-02-21 150331).txt
```

Synchronously copy blobs across Storage accounts

AzCopy by default copies data between two storage endpoints asynchronously. Therefore, the copy operation will run in the background using spare bandwidth capacity that has no SLA in terms of how fast a blob will be copied, and AzCopy will periodically check the copy status until the copying is completed or failed.

The `/SyncCopy` option ensures that the copy operation will get consistent speed. AzCopy performs the synchronous copy by downloading the blobs to copy from the specified source to local memory, and then uploading them to the Blob storage destination.

```
AzCopy /Source:https://myaccount1.blob.core.windows.net/myContainer/  
/Dest:https://myaccount2.blob.core.windows.net/myContainer/ /SourceKey:key1 /DestKey:key2 /Pattern:ab  
/SyncCopy
```

`/SyncCopy` might generate additional egress cost compared to asynchronous copy, the recommended approach is to use this option in an Azure VM that is in the same region as your source storage account to avoid egress cost.

File: Download

Download single file

```
AzCopy /Source:https://myaccount.file.core.windows.net/myfileshare/myfolder1/ /Dest:C:\myfolder  
/SourceKey:key /Pattern:abc.txt
```

If the specified source is an Azure file share, then you must either specify the exact file name, (e.g. `abc.txt`) to download a single file, or specify option `/s` to download all files in the share recursively. Attempting

to specify both a file pattern and option `/S` together will result in an error.

Download all files

```
AzCopy /Source:https://myaccount.file.core.windows.net/myfileshare/ /Dest:C:\myfolder /SourceKey:key /S
```

Note that any empty folders will not be downloaded.

File: Upload

Upload single file

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.file.core.windows.net/myfileshare/ /DestKey:key  
/Pattern:abc.txt
```

Upload all files

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.file.core.windows.net/myfileshare/ /DestKey:key /S
```

Note that any empty folders will not be uploaded.

Upload files matching specified pattern

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.file.core.windows.net/myfileshare/ /DestKey:key  
/Pattern:ab* /S
```

File: Copy

Copy across file shares

```
AzCopy /Source:https://myaccount1.file.core.windows.net/myfileshare1/  
/Dest:https://myaccount2.file.core.windows.net/myfileshare2/ /SourceKey:key1 /DestKey:key2 /S
```

Copy from file share to blob

```
AzCopy /Source:https://myaccount1.file.core.windows.net/myfileshare/  
/Dest:https://myaccount2.blob.core.windows.net/mycontainer/ /SourceKey:key1 /DestKey:key2 /S
```

Note that asynchronous copying from File Storage to Page Blob is not supported.

Copy from blob to file share

```
AzCopy /Source:https://myaccount1.blob.core.windows.net/mycontainer/  
/Dest:https://myaccount2.file.core.windows.net/myfileshare/ /SourceKey:key1 /DestKey:key2 /S
```

Synchronously copy files

You can specify the `/SyncCopy` option to copy data from File Storage to File Storage, from File Storage to Blob Storage and from Blob Storage to File Storage synchronously, AzCopy does this by downloading the source data to local memory and upload it again to destination.

```
AzCopy /Source:https://myaccount1.file.core.windows.net/myfileshare1/  
/Dest:https://myaccount2.file.core.windows.net/myfileshare2/ /SourceKey:key1 /DestKey:key2 /S /SyncCopy
```

When copying from File Storage to Blob Storage, the default blob type is block blob, user can specify option `/BlobType:page` to change the destination blob type.

Note that `/SyncCopy` might generate additional egress cost comparing to asynchronous copy, the recommended approach is to use this option in the Azure VM which is in the same region as your source storage account to avoid egress cost.

Table: Export

Export table

```
AzCopy /Source:https://myaccount.table.core.windows.net/myTable/ /Dest:C:\myfolder\ /SourceKey:key
```

AzCopy writes a manifest file to the specified destination folder. The manifest file is used in the import process to locate the necessary data files and perform data validation. The manifest file uses the following naming convention by default:

```
<account name>_<table name>_<timestamp>.manifest
```

User can also specify the option `/Manifest:<manifest file name>` to set the manifest file name.

```
AzCopy /Source:https://myaccount.table.core.windows.net/myTable/ /Dest:C:\myfolder\ /SourceKey:key  
/Manifest:abc.manifest
```

Split export into multiple files

```
AzCopy /Source:https://myaccount.table.core.windows.net/mytable/ /Dest:C:\myfolder /SourceKey:key /S  
/SplitOptions:100
```

AzCopy uses a *volume index* in the split data file names to distinguish multiple files. The volume index consists of two parts, a *partition key range index* and a *split file index*. Both indexes are zero-based.

The partition key range index will be 0 if user does not specify option `/PKRS`.

For instance, suppose AzCopy generates two data files after the user specifies option `/SplitOptions`. The resulting data file names might be:

```
myaccount_mytable_20140903T051850.8128447Z_0_0_C3040FE8.json  
myaccount_mytable_20140903T051850.8128447Z_0_1_0AB9AC20.json
```

Note that the minimum possible value for option `/SplitOptions` is 32MB. If the specified destination is Blob storage, AzCopy will split the data file once its sizes reaches the blob size limitation (200GB), regardless of whether option `/SplitOptions` has been specified by the user.

Export table to JSON or CSV data file format

AzCopy by default exports tables to JSON data files. You can specify the option `/PayloadFormat:JSON|CSV` to export the tables as JSON or CSV.

```
AzCopy /Source:https://myaccount.table.core.windows.net/myTable/ /Dest:C:\myfolder\ /SourceKey:key  
/PayloadFormat:CSV
```

When specifying the CSV payload format, AzCopy will also generate a schema file with file extension `.schema.csv` for each data file.

Export table entities concurrently

```
AzCopy /Source:https://myaccount.table.core.windows.net/myTable/ /Dest:C:\myfolder\ /SourceKey:key  
/PKRS:"aa#bb"
```

AzCopy will start concurrent operations to export entities when the user specifies option `/PKRS`. Each operation exports one partition key range.

Note that the number of concurrent operations is also controlled by option `/NC`. AzCopy uses the number of core processors as the default value of `/NC` when copying table entities, even if `/NC` was not specified. When the user specifies option `/PKRS`, AzCopy uses the smaller of the two values - partition key ranges versus implicitly or explicitly specified concurrent operations - to determine the number of concurrent operations to start. For more details, type `AzCopy /?:NC` at the command line.

Export table to blob

```
AzCopy /Source:https://myaccount.table.core.windows.net/myTable/  
/Dest:https://myaccount.blob.core.windows.net/mycontainer/ /SourceKey:key1 /Destkey:key2
```

AzCopy will generate a JSON data file into the blob container with following naming convention:

```
<account name>_<table name>_<timestamp>_<volume index>_<CRC>.json
```

The generated JSON data file follows the payload format for minimal metadata. For details on this payload format, see [Payload Format for Table Service Operations](#).

Note that when exporting tables to blobs, AzCopy will download the Table entities to local temporary data files and then upload those entities to the blob. These temporary data files are put into the journal file folder with the default path "`%LocalAppData%\Microsoft\Azure\AzCopy`", you can specify option `/Z:[journal-file-folder]` to change the journal file folder location and thus change the temporary data files location. The temporary data files' size is decided by your table entities' size and the size you specified with the option `/SplitSize`, although the temporary data file in local disk will be deleted instantly once it has been uploaded to the blob, please make sure you have enough local disk space to store these temporary data files before they are deleted.

Table: Import

Import table

```
AzCopy /Source:C:\myfolder\ /Dest:https://myaccount.table.core.windows.net/mytable1/ /DestKey:key  
/Manifest:"myaccount_mytable_20140103T112020.manifest" /EntityOperation:InsertOrReplace
```

The option `/EntityOperation` indicates how to insert entities into the table. Possible values are:

- `InsertOrSkip`: Skips an existing entity or inserts a new entity if it does not exist in the table.
- `InsertOrMerge`: Merges an existing entity or inserts a new entity if it does not exist in the table.
- `InsertOrReplace`: Replaces an existing entity or inserts a new entity if it does not exist in the table.

Note that you cannot specify option `/PKRS` in the import scenario. Unlike the export scenario, in which you must specify option `/PKRS` to start concurrent operations, AzCopy will by default start concurrent operations when you import a table. The default number of concurrent operations started is equal to the number of core processors; however, you can specify a different number of concurrent with option `/NC`. For more details, type `AzCopy /?:NC` at the command line.

Note that AzCopy only supports importing for JSON, not CSV. AzCopy does not support table imports from user-created JSON and manifest files. Both of these files must come from an AzCopy table export. To

avoid errors, please do not modify the exported JSON or manifest file.

Import entities to table using blobs

Assume a Blob container contains the following: A JSON file representing an Azure Table and its accompanying manifest file.

```
myaccount_mytable_20140103T112020.manifest  
myaccount_mytable_20140103T112020_0_0_AF395F1DC42E952.json
```

You can run the following command to import entities into a table using the manifest file in that blob container:

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer  
/Dest:https://myaccount.table.core.windows.net/mytable /SourceKey:key1 /DestKey:key2  
/Manifest:"myaccount_mytable_20140103T112020.manifest" /EntityOperation:"InsertOrReplace"
```

Other AzCopy features

Only copy data that doesn't exist in the destination

The `/XO` and `/XN` parameters allow you to exclude older or newer source resources from being copied, respectively. If you only want to copy source resources that don't exist in the destination, you can specify both parameters in the AzCopy command:

```
/Source:http://myaccount.blob.core.windows.net/mycontainer /Dest:C:\myfolder /SourceKey:<sourcekey> /S  
/XO /XN  
  
/Source:C:\myfolder /Dest:http://myaccount.file.core.windows.net/myfileshare /DestKey:<destkey> /S /XO  
/XN  
  
/Source:http://myaccount.blob.core.windows.net/mycontainer  
/Dest:http://myaccount.blob.core.windows.net/mycontainer1 /SourceKey:<sourcekey> /DestKey:<destkey> /S  
/XO /XN
```

Note that this is not supported when either the source or destination is a table.

Use a response file to specify command-line parameters

```
AzCopy @:"C:\responsefiles\copyoperation.txt"
```

You can include any AzCopy command-line parameters in a response file. AzCopy processes the parameters in the file as if they had been specified on the command line, performing a direct substitution with the contents of the file.

Assume a response file named `copyoperation.txt`, that contains the following lines. Each AzCopy parameter can be specified on a single line

```
/Source:http://myaccount.blob.core.windows.net/mycontainer /Dest:C:\myfolder /SourceKey:<sourcekey> /S  
/Y
```

or on separate lines:

```
/Source:http://myaccount.blob.core.windows.net/mycontainer  
/Dest:C:\myfolder  
/SourceKey:<sourcekey>  
/S  
/Y
```

AzCopy will fail if you split the parameter across two lines, as shown here for the `/sourcekey` parameter:

```
http://myaccount.blob.core.windows.net/mycontainer  
C:\myfolder  
/sourcekey:  
<sourcekey>  
/S  
/Y
```

Use multiple response files to specify command-line parameters

Assume a response file named `source.txt` that specifies a source container:

```
/Source:http://myaccount.blob.core.windows.net/mycontainer
```

And a response file named `dest.txt` that specifies a destination folder in the file system:

```
/Dest:C:\myfolder
```

And a response file named `options.txt` that specifies options for AzCopy:

```
/S /Y
```

To call AzCopy with these response files, all of which reside in a directory `C:\responsefiles`, use this command:

```
AzCopy /@:"C:\responsefiles\source.txt" /@:"C:\responsefiles\dest.txt" /SourceKey:<sourcekey>  
/@:"C:\responsefiles\options.txt"
```

AzCopy processes this command just as it would if you included all of the individual parameters on the command line:

```
AzCopy /Source:http://myaccount.blob.core.windows.net/mycontainer /Dest:C:\myfolder /SourceKey:  
<sourcekey> /S /Y
```

Specify a shared access signature (SAS)

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer1  
/Dest:https://myaccount.blob.core.windows.net/mycontainer2 /SourceSAS:SAS1 /DestSAS:SAS2  
/Pattern:abc.txt
```

You can also specify a SAS on the container URI:

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer1/?SourceSASToken /Dest:C:\myfolder /S
```

Journal file folder

Each time you issue a command to AzCopy, it checks whether a journal file exists in the default folder, or whether it exists in a folder that you specified via this option. If the journal file does not exist in either place, AzCopy treats the operation as new and generates a new journal file.

If the journal file does exist, AzCopy will check whether the command line that you input matches the command line in the journal file. If the two command lines match, AzCopy resumes the incomplete operation. If they do not match, you will be prompted to either overwrite the journal file to start a new operation, or to cancel the current operation.

If you want to use the default location for the journal file:

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.blob.core.windows.net/mycontainer /DestKey:key /Z
```

If you omit option `/Z`, or specify option `/Z` without the folder path, as shown above, AzCopy creates the journal file in the default location, which is

`%SystemDrive%\Users\%username%\AppData\Local\Microsoft\Azure\AzCopy`. If the journal file already exists, then AzCopy resumes the operation based on the journal file.

If you want to specify a custom location for the journal file:

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.blob.core.windows.net/mycontainer /DestKey:key /Z:C:\journalfolder\
```

This example creates the journal file if it does not already exist. If it does exist, then AzCopy resumes the operation based on the journal file.

If you want to resume an AzCopy operation:

```
AzCopy /Z:C:\journalfolder\
```

This example resumes the last operation, which may have failed to complete.

Generate a log file

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.blob.core.windows.net/mycontainer /DestKey:key /V
```

If you specify option `/V` without providing a file path to the verbose log, then AzCopy creates the log file in the default location, which is `%SystemDrive%\Users\%username%\AppData\Local\Microsoft\Azure\AzCopy`.

Otherwise, you can create an log file in a custom location:

```
AzCopy /Source:C:\myfolder /Dest:https://myaccount.blob.core.windows.net/mycontainer /DestKey:key /V:C:\myfolder\azcopy1.log
```

Note that if you specify a relative path following option `/V`, such as `/V:test/azcopy1.log`, then the verbose log is created in the current working directory within a subfolder named `test`.

Specify the number of concurrent operations to start

Option `/NC` specifies the number of concurrent copy operations. By default, AzCopy starts a certain number of concurrent operations to increase the data transfer throughput. For Table operations, the number of concurrent operations is equal to the number of processors you have. For Blob and File operations, the number of concurrent operations is equal 8 times the number of processors you have. If you are running AzCopy across a low-bandwidth network, you can specify a lower number for `/NC` to avoid failure caused by resource competition.

Run AzCopy against Azure Storage Emulator

You can run AzCopy against the [Azure Storage Emulator](#) for Blobs:

```
AzCopy /Source:https://127.0.0.1:10000/myaccount/mycontainer/ /Dest:C:\myfolder /SourceKey:key /SourceType:Blob /S
```

and Tables:

```
AzCopy /Source:https://127.0.0.1:10002/myaccount/mytable/ /Dest:C:\myfolder /SourceKey:key  
/SourceType:Table
```

AzCopy Parameters

Parameters for AzCopy are described below. You can also type one of the following commands from the command line for help in using AzCopy:

- For detailed command-line help for AzCopy: `AzCopy /?`
- For detailed help with any AzCopy parameter: `AzCopy /?:SourceKey`
- For command-line examples: `AzCopy /?:Samples`

/Source:"source"

Specifies the source data from which to copy. The source can be a file system directory, a blob container, a blob virtual directory, a storage file share, a storage file directory, or an Azure table.

Applicable to: Blobs, Files, Tables

/Dest:"destination"

Specifies the destination to copy to. The destination can be a file system directory, a blob container, a blob virtual directory, a storage file share, a storage file directory, or an Azure table.

Applicable to: Blobs, Files, Tables

/Pattern:"file-pattern"

Specifies a file pattern that indicates which files to copy. The behavior of the /Pattern parameter is determined by the location of the source data, and the presence of the recursive mode option. Recursive mode is specified via option /S.

If the specified source is a directory in the file system, then standard wildcards are in effect, and the file pattern provided is matched against files within the directory. If option /S is specified, then AzCopy also matches the specified pattern against all files in any subfolders beneath the directory.

If the specified source is a blob container or virtual directory, then wildcards are not applied. If option /S is specified, then AzCopy interprets the specified file pattern as a blob prefix. If option /S is not specified, then AzCopy matches the file pattern against exact blob names.

If the specified source is an Azure file share, then you must either specify the exact file name, (e.g. abc.txt) to copy a single file, or specify option /S to copy all files in the share recursively. Attempting to specify both a file pattern and option /S together will result in an error.

AzCopy uses case-sensitive matching when the /Source is a blob container or blob virtual directory, and uses case-insensitive matching in all the other cases.

The default file pattern used when no file pattern is specified is . for a file system location or an empty prefix for an Azure Storage location. Specifying multiple file patterns is not supported.

Applicable to: Blobs, Files

/DestKey:"storage-key"

Specifies the storage account key for the destination resource.

Applicable to: Blobs, Files, Tables

/DestSAS:"sas-token"

Specifies a Shared Access Signature (SAS) with READ and WRITE permissions for the destination (if applicable). Surround the SAS with double quotes, as it may contain special command-line characters.

If the destination resource is a blob container, file share or table, you can either specify this option followed by the SAS token, or you can specify the SAS as part of the destination blob container, file share or table's URI, without this option.

If the source and destination are both blobs, then the destination blob must reside within the same storage account as the source blob.

Applicable to: Blobs, Files, Tables

/SourceKey:"storage-key"

Specifies the storage account key for the source resource.

Applicable to: Blobs, Files, Tables

/SourceSAS:"sas-token"

Specifies a Shared Access Signature with READ and LIST permissions for the source (if applicable).

Surround the SAS with double quotes, as it may contain special command-line characters.

If the source resource is a blob container, and neither a key nor a SAS is provided, then the blob container will be read via anonymous access.

If the source is a file share or table, a key or a SAS must be provided.

Applicable to: Blobs, Files, Tables

/S

Specifies recursive mode for copy operations. In recursive mode, AzCopy will copy all blobs or files that match the specified file pattern, including those in subfolders.

Applicable to: Blobs, Files

/BlobType:"block" | "page" | "append"

Specifies whether the destination blob is a block blob, a page blob, or an append blob. This option is applicable only when you are uploading a blob. Otherwise, an error is generated. If the destination is a blob and this option is not specified, by default, AzCopy creates a block blob.

Applicable to: Blobs

/CheckMD5

Calculates an MD5 hash for downloaded data and verifies that the MD5 hash stored in the blob or file's Content-MD5 property matches the calculated hash. The MD5 check is turned off by default, so you must specify this option to perform the MD5 check when downloading data.

Note that Azure Storage doesn't guarantee that the MD5 hash stored for the blob or file is up-to-date. It is client's responsibility to update the MD5 whenever the blob or file is modified.

AzCopy always sets the Content-MD5 property for an Azure blob or file after uploading it to the service.

Applicable to: Blobs, Files

/Snapshot

Indicates whether to transfer snapshots. This option is only valid when the source is a blob.

The transferred blob snapshots are renamed in this format: blob-name (snapshot-time).extension

By default, snapshots are not copied.

Applicable to: Blobs

/V:[verbose-log-file]

Outputs verbose status messages into a log file.

By default, the verbose log file is named AzCopyVerbose.log in `%LocalAppData%\Microsoft\Azure\AzCopy`. If you specify an existing file location for this option, the verbose log will be appended to that file.

Applicable to: Blobs, Files, Tables

/Z:[journal-file-folder]

Specifies a journal file folder for resuming an operation.

AzCopy always supports resuming if an operation has been interrupted.

If this option is not specified, or it is specified without a folder path, then AzCopy will create the journal file in the default location, which is `%LocalAppData%\Microsoft\Azure\AzCopy`.

Each time you issue a command to AzCopy, it checks whether a journal file exists in the default folder, or whether it exists in a folder that you specified via this option. If the journal file does not exist in either place, AzCopy treats the operation as new and generates a new journal file.

If the journal file does exist, AzCopy will check whether the command line that you input matches the command line in the journal file. If the two command lines match, AzCopy resumes the incomplete operation. If they do not match, you will be prompted to either overwrite the journal file to start a new operation, or to cancel the current operation.

The journal file is deleted upon successful completion of the operation.

Note that resuming an operation from a journal file created by a previous version of AzCopy is not supported.

Applicable to: Blobs, Files, Tables

/@:"parameter-file"

Specifies a file that contains parameters. AzCopy processes the parameters in the file just as if they had been specified on the command line.

In a response file, you can either specify multiple parameters on a single line, or specify each parameter on its own line. Note that an individual parameter cannot span multiple lines.

Response files can include comments lines that begin with the # symbol.

You can specify multiple response files. However, note that AzCopy does not support nested response files.

Applicable to: Blobs, Files, Tables

/Y

Suppresses all AzCopy confirmation prompts.

Applicable to: Blobs, Files, Tables

/L

Specifies a listing operation only; no data is copied.

AzCopy will interpret the using of this option as a simulation for running the command line without this option /L and count how many objects will be copied, you can specify option /V at the same time to check which objects will be copied in the verbose log.

The behavior of this option is also determined by the location of the source data and the presence of the recursive mode option /S and file pattern option /Pattern.

AzCopy requires LIST and READ permission of this source location when using this option.

Applicable to: Blobs, Files

/MT

Sets the downloaded file's last-modified time to be the same as the source blob or file's.

Applicable to: Blobs, Files

/XN

Excludes a newer source resource. The resource will not be copied if the last modified time of the source is the same or newer than destination.

Applicable to: Blobs, Files

/XO

Excludes an older source resource. The resource will not be copied if the last modified time of the source is the same or older than destination.

Applicable to: Blobs, Files

/A

Uploads only files that have the Archive attribute set.

Applicable to: Blobs, Files

/IA:[RASHCNETOI]

Uploads only files that have any of the specified attributes set.

Available attributes include:

- R = Read-only files
- A = Files ready for archiving
- S = System files
- H = Hidden files
- C = Compressed files
- N = Normal files
- E = Encrypted files
- T = Temporary files
- O = Offline files
- I = Non-indexed files

Applicable to: Blobs, Files

/XA:[RASHCNETOI]

Excludes files that have any of the specified attributes set.

Available attributes include:

- R = Read-only files
- A = Files ready for archiving
- S = System files
- H = Hidden files
- C = Compressed files
- N = Normal files
- E = Encrypted files

- T = Temporary files
- O = Offline files
- I = Non-indexed files

Applicable to: Blobs, Files

/Delimiter:"delimiter"

Indicates the delimiter character used to delimit virtual directories in a blob name.

By default, AzCopy uses / as the delimiter character. However, AzCopy supports using any common character (such as @, #, or %) as a delimiter. If you need to include one of these special characters on the command line, enclose the file name with double quotes.

This option is only applicable for downloading blobs.

Applicable to: Blobs

/NC:"number-of-concurrent-operations"

Specifies the number of concurrent operations.

AzCopy by default starts a certain number of concurrent operations to increase the data transfer throughput. Note that large number of concurrent operations in a low-bandwidth environment may overwhelm the network connection and prevent the operations from fully completing. Throttle concurrent operations based on actual available network bandwidth.

The upper limit for concurrent operations is 512.

Applicable to: Blobs, Files, Tables

/SourceType:"Blob" | "Table"

Specifies that the `source` resource is a blob available in the local development environment, running in the storage emulator.

Applicable to: Blobs, Tables

/DestType:"Blob" | "Table"

Specifies that the `destination` resource is a blob available in the local development environment, running in the storage emulator.

Applicable to: Blobs, Tables

/PKRS:"key1#key2#key3#..."

Splits the partition key range to enable exporting table data in parallel, which increases the speed of the export operation.

If this option is not specified, then AzCopy uses a single thread to export table entities. For example, if the user specifies /PKRS:"aa#bb", then AzCopy starts three concurrent operations.

Each operation exports one of three partition key ranges, as shown below:

[first-partition-key, aa)

[aa, bb)

[bb, last-partition-key]

Applicable to: Tables

/SplitSize:"file-size"

Specifies the exported file split size in MB, the minimal value allowed is 32.

If this option is not specified, AzCopy will export table data to single file.

If the table data is exported to a blob, and the exported file size reaches the 200 GB limit for blob size, then AzCopy will split the exported file, even if this option is not specified.

Applicable to: Tables

/EntityOperation:"InsertOrSkip" | "InsertOrMerge" | "InsertOrReplace"

Specifies the table data import behavior.

- InsertOrSkip - Skips an existing entity or inserts a new entity if it does not exist in the table.
- InsertOrMerge - Merges an existing entity or inserts a new entity if it does not exist in the table.
- InsertOrReplace - Replaces an existing entity or inserts a new entity if it does not exist in the table.

Applicable to: Tables

/Manifest:"manifest-file"

Specifies the manifest file for the table export and import operation.

This option is optional during the export operation, AzCopy will generate a manifest file with predefined name if this option is not specified.

This option is required during the import operation for locating the data files.

Applicable to: Tables

/SyncCopy

Indicates whether to synchronously copy blobs or files between two Azure Storage endpoints.

AzCopy by default uses server-side asynchronous copy. Specify this option to perform a synchronous copy, which downloads blobs or files to local memory and then uploads them to Azure Storage.

You can use this option when copying files within Blob storage, within File storage, or from Blob storage to File storage or vice versa.

Applicable to: Blobs, Files

/SetContentType:"content-type"

Specifies the MIME content type for destination blobs or files.

AzCopy sets the content type for a blob or file to application/octet-stream by default. You can set the content type for all blobs or files by explicitly specifying a value for this option.

If you specify this option without a value, then AzCopy will set each blob or file's content type according to its file extension.

Applicable to: Blobs, Files

/PayloadFormat:"JSON" | "CSV"

Specifies the format of the table exported data file.

If this option is not specified, by default AzCopy exports table data file in JSON format.

Applicable to: Tables

Known Issues and Best Practices

Limit concurrent writes while copying data

When you copy blobs or files with AzCopy, keep in mind that another application may be modifying the

data while you are copying it. If possible, ensure that the data you are copying is not being modified during the copy operation. For example, when copying a VHD associated with an Azure virtual machine, make sure that no other applications are currently writing to the VHD. A good way to do this is by leasing the resource to be copied. Alternately, you can create a snapshot of the VHD first and then copy the snapshot.

If you cannot prevent other applications from writing to blobs or files while they are being copied, then keep in mind that by the time the job finishes, the copied resources may no longer have full parity with the source resources.

Run one AzCopy instance on one machine.

AzCopy is designed to maximize the utilization of your machine resource to accelerate the data transfer, we recommend you run only one AzCopy instance on one machine, and specify the option `/NC` if you need more concurrent operations. For more details, type `AzCopy /?:NC` at the command line.

Enable FIPS compliant MD5 algorithms for AzCopy when you "Use FIPS compliant algorithms for encryption, hashing and signing".

AzCopy by default uses .NET MD5 implementation to calculate the MD5 when copying objects, but there are some security requirements that need AzCopy to enable FIPS compliant MD5 setting.

You can create an app.config file `AzCopy.exe.config` with property `AzureStorageUseV1MD5` and put it aside with AzCopy.exe.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="AzureStorageUseV1MD5" value="false"/>
  </appSettings>
</configuration>
```

For property "AzureStorageUseV1MD5" • True - The default value, AzCopy will use .NET MD5 implementation. • False – AzCopy will use FIPS compliant MD5 algorithm.

Note that FIPS compliant algorithms is disabled by default on your Windows machine, you can type secpol.msc in your Run window and check this switch at Security Setting->Local Policy->Security Options->System cryptography: Use FIPS compliant algorithms for encryption, hashing and signing.

Next steps

For more information about Azure Storage and AzCopy, refer to the following resources.

Azure Storage documentation:

- [Introduction to Azure Storage](#)
- [How to use Blob storage from .NET](#)
- [How to use File storage from .NET](#)
- [How to use Table storage from .NET](#)
- [How to create, manage, or delete a storage account](#)

Azure Storage blog posts:

- [Introducing Azure Storage Data Movement Library Preview](#)
- [AzCopy: Introducing synchronous copy and customized content type](#)
- [AzCopy: Announcing General Availability of AzCopy 3.0 plus preview release of AzCopy 4.0 with Table and File support](#)
- [AzCopy: Optimized for Large-Scale Copy Scenarios](#)
- [AzCopy: Support for read-access geo-redundant storage](#)

- [AzCopy: Transfer data with re-startable mode and SAS token](#)
- [AzCopy: Using cross-account Copy Blob](#)
- [AzCopy: Uploading/downloading files for Azure Blobs](#)

Use the Microsoft Azure Import/Export Service to transfer data to Blob storage

1/17/2017 • 30 min to read • [Edit on GitHub](#)

Overview

Azure Import/Export Service allows you to securely transfer large amounts of data to Azure blob storage by shipping hard disk drives to an Azure data center. You can also use this service to transfer data from Azure blob storage to hard disk drives and ship to your on-premises site. This service is suitable in situations where you want to transfer several TBs of data to or from Azure, but uploading or downloading over the network is not feasible due to limited bandwidth or high network costs.

The service requires that hard disk drives should be BitLocker encrypted for the security of your data. The service supports both the Classic and Azure Resource Manager storage accounts (standard and cool tier) present in all the regions of Public Azure. You must ship hard disk drives to one of the supported locations specified later in this article.

In this article, you will learn more about the Azure Import/Export service and how to ship drives for copying your data to and from Azure Blob storage.

When should I use the Azure Import/Export service?

You can consider using Azure Import/Export service when uploading or downloading data over the network is too slow or getting additional network bandwidth is cost prohibitive.

You can use this service in scenarios such as:

- Migrating data to the cloud: Move large amounts of data to Azure quickly and cost effectively.
- Content distribution: Quickly send data to your customer sites.
- Backup: Take backups of your on-premises data to store in Azure blob storage.
- Data recovery: Recover large amount of data stored in blob storage and have it delivered to your on-premises location.

Prerequisites

In this section, we have listed the prerequisites required to use this service. Please review them carefully before shipping your drives.

Storage account

You must have an existing Azure subscription and one or more storage accounts to use the Import/Export service. Each job may be used to transfer data to or from only one storage account. In other words, a single import/export job cannot span across multiple storage accounts. For information on creating a new storage account, see [How to Create a Storage Account](#).

Blob types

You can use Azure Import/Export service to copy data to **Block** blobs or **Page** blobs. Conversely, you can only export **Block** blobs, **Page** blobs or **Append** blobs from Azure storage using this service.

Job

To begin the process of importing to or exporting from Blob storage, you first create a job. A job can be an import

job or an export job:

- Create an import job when you want to transfer data you have on-premises to blobs in your Azure storage account.
- Create an export job when you want to transfer data currently stored as blobs in your storage account to hard drives that are shipped to you.

When you create a job, you notify the Import/Export service that you will be shipping one or more hard drives to an Azure data center.

- For an import job, you will be shipping hard drives containing your data.
- For an export job, you will be shipping empty hard drives.
- You can ship up to 10 hard disk drives per job.

You can create an import or export job using the Azure Portal or the [Azure Storage Import/Export REST API](#).

WAImportExport tool

The first step in creating an **import** job is to prepare your drives that will be shipped for import. To prepare your drives, you must connect it to a local server and run the WAImportExport Tool on the local server. This WAImportExport tool facilitates copying your data to the drive, encrypting the data on the drive with BitLocker, and generating the drive journal files.

The journal files store basic information about your job and drive such as drive serial number and storage account name. This journal file is not stored on the drive. It is used during import job creation. Step by step details about job creation is provided later in this article.

The WAImportExport tool is only compatible with 64-bit Windows operating system. See the [Operating System](#) section for specific OS versions supported.

Download the latest version of the [WAImportExport tool](#). For more details about using the WAImportExport Tool, see the [Using the WAImportExport Tool](#).

NOTE

Previous Version: You can [download WAImportExpot V1](#) version of the tool and refer to [WAImportExpot V1 usage guide](#). WAImportExpot V1 version of the tool does provide support for **preparing disks when data is already pre-written to the disk**. Also you will need to use WAImportExpot V1 tool if the only key available is SAS-Key.

Hard disk drives

Only 2.5 inch SSD or 2.5" or 3.5" SATA II or III internal hard drives are supported for use with the Import/Export service. You can use hard drives up to 10TB. For import jobs, only the first data volume on the drive will be processed. The data volume must be formatted with NTFS. When copying data to your hard drive, you can attach it directly using a 2.5 inch SSD or 2.5" or 3.5" SATA II or III connector or you can attach it externally using an external 2.5 inch SSD or 2.5" or 3.5" SATA II or III USB adaptor.

IMPORTANT

External hard disk drives that come with a built-in USB adaptor are not supported by this service. Also, the disk inside the casing of an external HDD cannot be used; please do not send external HDDs.

Encryption

The data on the drive must be encrypted using BitLocker Drive Encryption. This protects your data while it is in transit.

For import jobs, there are two ways to perform the encryption. The first way is to specify the option when using

dataset CSV file while running the WAImportExport tool during drive preparation. The second way is to enable BitLocker encryption manually on the drive and specify the encryption key in the driveset CSV when running WAImportExport tool command line during drive preparation.

For export jobs, after your data is copied to the drives, the service will encrypt the drive using BitLocker before shipping it back to you. The encryption key will be provided to you via the Azure portal.

Operating System

You can use one of the following 64-bit Operating Systems to prepare the hard drive using the WAImportExport Tool before shipping the drive to Azure:

Windows 7 Enterprise, Windows 7 Ultimate, Windows 8 Pro, Windows 8 Enterprise, Windows 8.1 Pro, Windows 8.1 Enterprise, Windows 10¹, Windows Server 2008 R2, Windows Server 2012, Windows Server 2012 R2. All of these operating systems support BitLocker Drive Encryption.

Locations

The Azure Import/Export service supports copying data to and from all Public Azure storage accounts. You can ship hard disk drives to one of the following locations. If your storage account is in a public Azure location which is not specified here, an alternate shipping location will be provided when you are creating the job using the Azure portal or the Import/Export REST API.

Supported shipping locations:

- East US
- West US
- East US 2
- Central US
- North Central US
- South Central US
- North Europe
- West Europe
- East Asia
- Southeast Asia
- Australia East
- Australia Southeast
- Japan West
- Japan East
- Central India
- Canada
- US Gov
- China

Shipping

Shipping drives to the data center:

When creating an import or export job, you will be provided a shipping address of one of the supported locations to ship your drives. The shipping address provided will depend on the location of your storage account, but it may not be the same as your storage account location.

You can use carriers like FedEx, DHL, UPS, or the US Postal Service to ship your drives to the shipping address.

Shipping drives from the data center:

When creating an import or export job, you must provide a return address for Microsoft to use when shipping the

drives back after your job is complete. Please make sure you provide a valid return address to avoid delays in processing.

You can use a carrier of your choice in order to forward ship the hard disk. The carrier should have appropriate tracking in order to maintain chain of custody. You must also provide a valid FedEx or DHL carrier account number to be used by Microsoft for shipping the drives back. A FedEx account number is required for shipping drives back from the US and Europe locations. A DHL account number is required for shipping drives back from Asia and Australia locations. You can create a [FedEx](#) (for US and Europe) or [DHL](#) (Asia and Australia) carrier account if you do not have one. If you already have a carrier account number, please verify that it is valid.

In shipping your packages, you must follow the terms at [Microsoft Azure Service Terms](#).

IMPORTANT

Please note that the physical media that you are shipping may need to cross international borders. You are responsible for ensuring that your physical media and data are imported and/or exported in accordance with the applicable laws. Before shipping the physical media, check with your advisers to verify that your media and data can legally be shipped to the identified data center. This will help to ensure that it reaches Microsoft in a timely manner. For instance, any package that will cross international borders needs a commercial invoice to be accompanied with the package (except if crossing borders within European Union). You could print out a filled copy of the commercial invoice from carrier website. Example of commercial invoice are [DHL Commercial Invoice](#) and [FedEx Commercial Invoice](#). Make sure that Microsoft has not been indicated as the exporter.

How does the Azure Import/Export service work?

You can transfer data between your on-premises site and Azure blob storage using the Azure Import/Export service by creating jobs and shipping hard disk drives to an Azure data center. Each hard disk drive you ship is associated with a single job. Each job is associated with a single storage account. Review the [pre-requisites section](#) carefully to learn about the specifics of this service such as supported blob types, disk types, locations, and shipping.

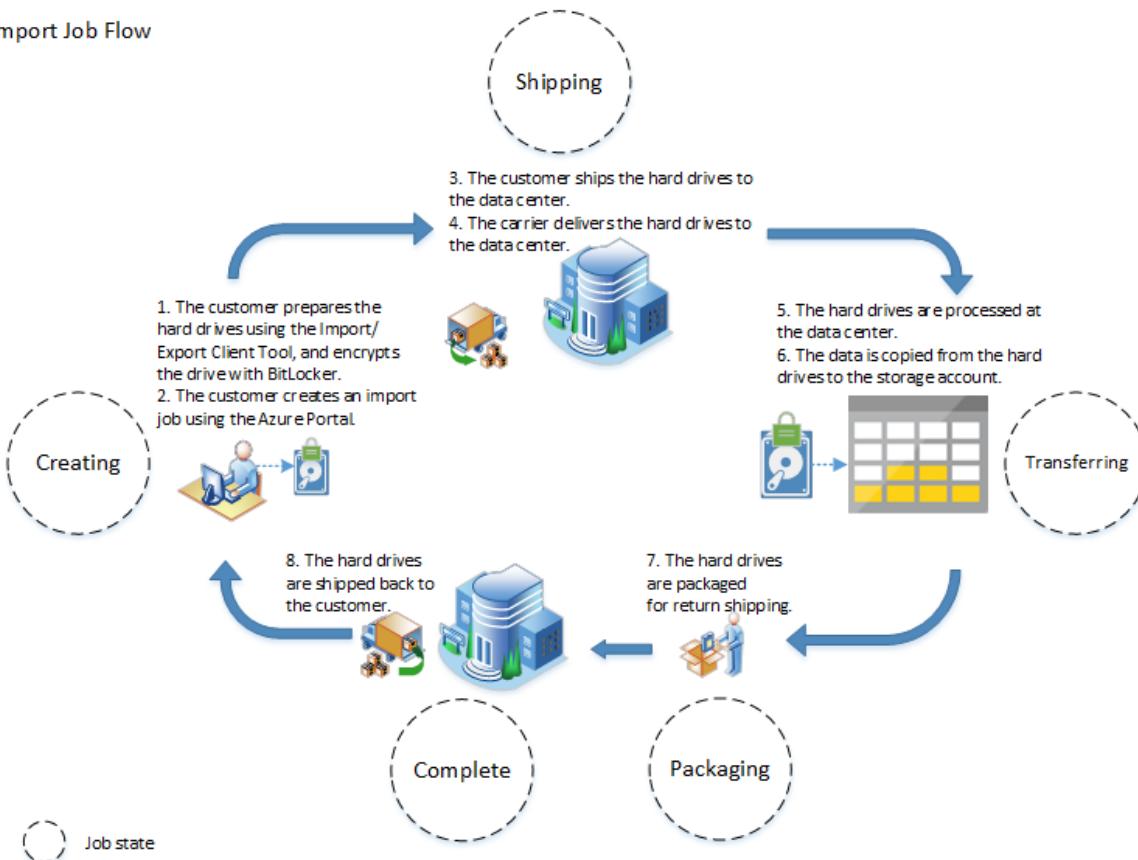
In this section, we will describe at a high level the steps involved in import and export jobs. Later in the [Quick Start section](#), we will provide step-by-step instructions to create an import and export job.

Inside an Import job

At a high level, an import job involves the following steps:

- Determine the data to be imported, and the number of drives you will need.
- Identify the destination blob location for your data in Blob storage.
- Use the WAImpoerExport Tool to copy your data to one or more hard disk drives and encrypt them with BitLocker.
- Create an import job in your target storage account using the Azure portal or the Import/Export REST API. If using the Azure portal, upload the drive journal files.
- Provide the return address and carrier account number to be used for shipping the drives back to you.
- Ship the hard disk drives to the shipping address provided during job creation.
- Update the delivery tracking number in the import job details and submit the import job.
- Drives are received and processed at the Azure data center.
- Drives are shipped using your carrier account to the return address provided in the import job.

Import Job Flow

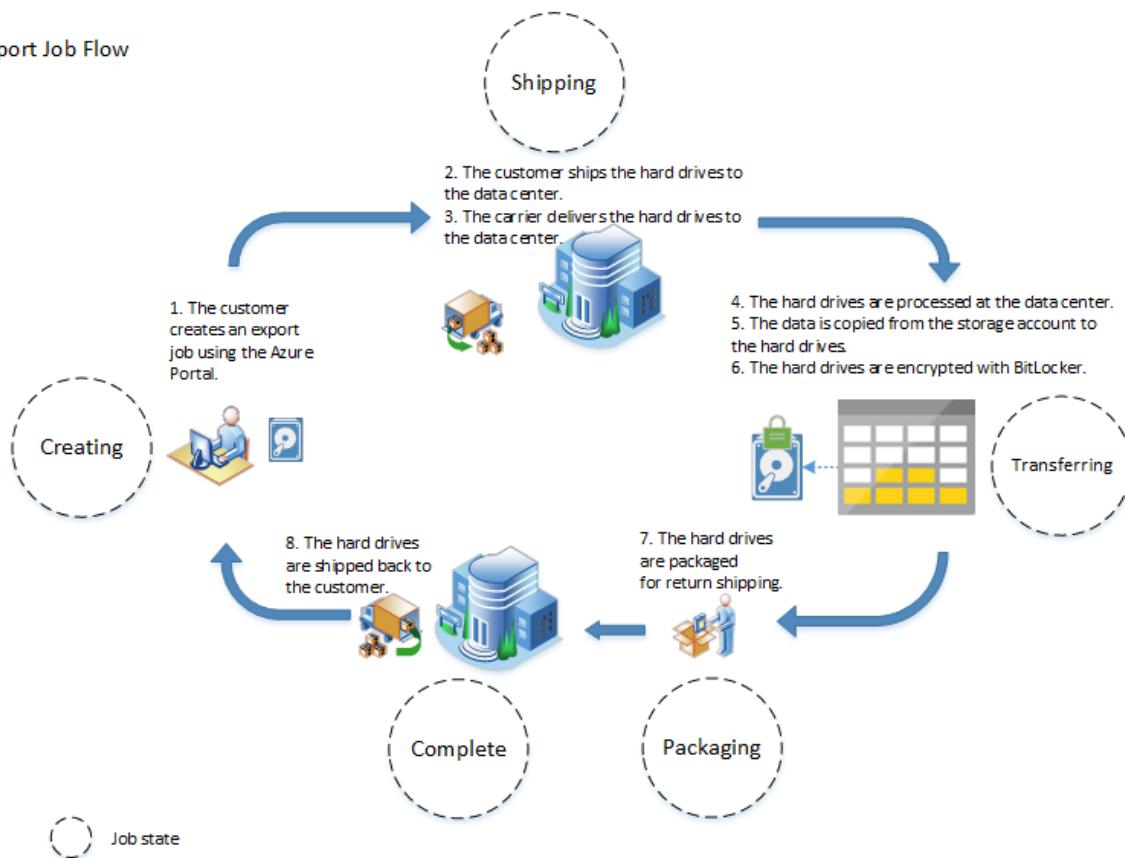


Inside an Export job

At a high level, an export job involves the following steps:

- Determine the data to be exported and the number of drives you will need.
- Identify the source blobs or container paths of your data in Blob storage.
- Create an export job in your source storage account using the Azure portal or the Import/Export REST API.
- Specify the source blobs or container paths of your data in the export job.
- Provide the return address and carrier account number for to be used for shipping the drives back to you.
- Ship the hard disk drives to the shipping address provided during job creation.
- Update the delivery tracking number in the export job details and submit the export job.
- The drives are received and processed at the Azure data center.
- The drives are encrypted with BitLocker; the keys are available via the Azure portal.
- The drives are shipped using your carrier account to the return address provided in the import job.

Export Job Flow



Viewing your job and drive status

You can track the status of your import or export jobs from the Azure portal. Click the **Import/Export** tab. A list of your jobs will appear on the page.

DRIVE ID	14310CD5BC48
COPY STATUS	-
Manifest information	
MANIFEST FILE	\DriveManifest.xml
HASH	428AEB94976CEBCEC02BBFFC183FC35E
URI	-
Log URI	
VERBOSE LOGS	-
ERROR LOGS	-

You will see one of the following job statuses depending on where your drive is in the process.

JOB STATUS	DESCRIPTION
Creating	After a job is created, its state is set to Creating. While the job is in the Creating state, the Import/Export service assumes the drives have not been shipped to the data center. A job may remain in the Creating state for up to two weeks, after which it is automatically deleted by the service.
Shipping	After you ship your package, you should update the tracking information in the Azure portal. This will turn the job into "Shipping". The job will remain in the Shipping state for up to two weeks.
Received	After all drives have been received at the data center, the job state will be set to the Received.
Transferring	Once at least one drive has begun processing, the job state will be set to the Transferring. See the Drive States section below for detailed information.
Packaging	After all drives have completed processing, the job will be placed in the Packaging state until the drives are shipped back to you.
Completed	After all drives have been shipped back to the customer, if the job has completed without errors, then the job will be set to the Completed state. The job will be automatically deleted after 90 days in the Completed state.
Closed	After all drives have been shipped back to the customer, if there have been any errors during the processing of the job, then the job will be set to the Closed state. The job will be automatically deleted after 90 days in the Closed state.

The table above describes the life cycle of an individual drive as it transitions through an import or export job. The current state of each drive in a job is now visible from the Azure Portal. The following table describes each state that each drive in a job may pass through.

120901

X Cancel job Delete

Essentials ^

Resource group (change) testrg	Overall percent completed 96%
Status Transferring	Storage account xtprodtestos2
Location Japan West	Type Import
Subscription name (change) Subscription-1	Delivery tracking number 3992016120901
Subscription ID afb1a4eb-bc88-4ce1-b1af-dfec80067464	Return tracking number -

2 drives

DRIVE ID	DRIVE STATE	PERCENT COMPLETE
WD-WX51DA476H28	● Completed with info	-
WD-WX21D5533HP2	● Transferring	96%

DRIVE STATE	DESCRIPTION
Specified	For an import job, when the job is created from the Azure Portal, the initial state for a drive is the Specified state. For an export job, since no drive is specified when the job is created, the initial drive state is the Received state.
Received	The drive transitions to the Received state when the Import/Export service operator has processed the drives that were received from the shipping company for an import job. For an export job, the initial drive state is the Received state.
NeverReceived	The drive will move to the NeverReceived state when the package for a job arrives but the package doesn't contain the drive. A drive can also move into this state if it has been two weeks since the service received the shipping information, but the package has not yet arrived at the data center.

Drive State	Description
Transferring	A drive will move to the Transferring state when the service begins to transfer data from the drive to Windows Azure Storage.
Completed	A drive will move to the Completed state when the service has successfully transferred all the data with no errors.
CompletedMoreInfo	A drive will move to the CompletedMoreInfo state when the service has encountered some issues while copying data either from or to the drive. The information can include errors, warnings, or informational messages about overwriting blobs.
ShippedBack	The drive will move to the ShippedBack state when it has been shipped from the data center back to the return address.

The following table describes the drive failure states and the actions taken for each state.

Drive State	Event	Resolution / Next Step
NeverReceived	A drive that is marked as NeverReceived (because it was not received as part of the job's shipment) arrives in another shipment.	The operations team will move the drive to the Received state.
N/A	A drive that is not part of any job arrives at the data center as part of another job.	The drive will be marked as an extra drive and will be returned to the customer when the job associated with the original package is completed.

Time to process job

The amount of time it takes to process an import/export job varies depending on different factors such as shipping time, job type, type and size of the data being copied, and the size of the disks provided. The Import/Export Service does not have an SLA. You can use the REST API to track the job progress more closely. There is a percent complete parameter in the List Jobs operation which gives an indication of copy progress. Reach out to us if you need an estimate to complete a time critical import/export job.

Pricing

Drive handling fee

There is a drive handling fee for each drive processed as part of your import or export job. See the details on the [Azure Import/Export Pricing](#).

Shipping costs

When you ship drives to Azure, you pay the shipping cost to the shipping carrier. When Microsoft returns the drives to you, the shipping cost is charged to the carrier account which you provided at the time of job creation.

Transaction costs

There are no transaction costs when importing data into blob storage. The standard egress charges are applicable when data is exported from blob storage. For more details on transaction costs, see [Data transfer pricing](#).

Quick Start

In this section, we will provide step-by-step instructions for creating an import and an export job. Please make sure you meet all of the [pre-requisites](#) before moving forward.

How to create an Import Job?

Create an import job to copy data to your Azure storage account from hard drives by shipping one or more drives containing data to the specified data center. The import job conveys details about hard disk drives, data to be copied, target storage account, and shipping information to the Azure Import/Export service. Creating an import job is a three-step process. First, prepare your drives using the WAImportExport tool. Second, submit an import job using the Azure portal. Third, ship the drives to the shipping address provided during job creation and update the shipping info in your job details.

IMPORTANT

You can submit only one job per storage account. Each drive you ship can be imported to one storage account. For example, let's say you wish to import data into two storage accounts. You must use separate hard disk drives for each storage account and create separate jobs per storage account.

Prepare Your Drives

The first step when importing data using the Azure Import/Export service is to prepare your drives using the WAImportExport tool. Follow the steps below to prepare your drives.

1. Identify the data to be imported. This could be directories and standalone files on the local server or a network share.
2. Determine the number of drives you will need depending on total size of the data. Procure the required number of 2.5 inch SSD or 2.5" or 3.5" SATA II or III hard disk drives.
3. Identify the target storage account, container, virtual directories, and blobs.
4. Determine the directories and/or standalone files that will be copied to each hard disk drive.
5. Create the CSV files for dataset and driveset.

Dataset CSV File

Below is a sample dataset CSV file example:

```
BasePath,DstBlobPathOrPrefix,BlobType,Disposition,MetadataFile,PropertiesFile
"F:\50M_original\100M_1.csv.txt","containername/100M_1.csv.txt",BlockBlob,rename,"None",None
"F:\50M_original\" , "containername/",BlockBlob,rename,"None",None
```

In the above example, 100M_1.csv.txt will be copied to the root of the container named "containername". If the container name "containername" does not exist, one will be created. All files and folders under 50M_original will be recursively copied to containername. Folder structure will be maintained.

Learn more about [preparing the dataset CSV file](#).

Remember: By default, the data will be imported as Block Blobs. You can use the BlobType field-value to import data as a Page Blobs. For example, if you are importing VHD files which will be mounted as disks on an Azure VM, you must import them as Page Blobs.

Driveset CSV File

The value of the driveset flag is a CSV file which contains the list of disks to which the drive letters are mapped in order to the tool to correctly pick the list of disks to be prepared.

Below is the example of driveset CSV file:

```
DriveLetter,FormatOption,SilentOrPromptOnFormat,Encryption,ExistingBitLockerKey  
G,AlreadyFormatted,SilentMode,AlreadyEncrypted,060456-014509-132033-080300-252615-584177-672089-411631 |  
H,Format,SilentMode,Encrypt,
```

In the above example, it is assumed that two disks are attached and basic NTFS volumes with volume-letter G:\ and H:\ have been created. The tool will format and encrypt the disk which hosts H:\ and will not format or encrypt the disk hosting volume G:.

Learn more about [preparing the driveset CSV file](#).

6. Use the [WAImportExport Tool](#) to copy your data to one or more hard drives.
7. You can specify "Encrypt" on Encryption field in drivset CSV to enable BitLocker encryption on the hard disk drive. Alternatively, you could also enable BitLocker encryption manually on the hard disk drive and specify "AlreadyEncrypted" and supply the key in the driveset CSV while running the tool.
8. Do not modify the data on the hard disk drives or the journal file after completing disk preparation.

IMPORTANT

Each hard disk drive you prepare will result in a journal file. When you are creating the import job using the Azure portal, you must upload all the journal files of the drives which are part of that import job. Drives without journal files will not be processed.

Below are the commands and examples for preparing the hard disk drive using WAImportExport tool.

WAImportExport tool PreImport command for the first copy session to copy directories and/or files with a new copy session:

```
WAImportExport.exe PrepImport /j:<JournalFile> /id:<SessionId> [/logdir:<LogDirectory>] [/sk:<StorageAccountKey>] [/silentmode] [/InitialDriveSet:<driveset.csv>] DataSet:<dataset.csv>
```

Example:

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#1 /sk:***** /InitialDriveSet:driveset-1.csv /DataSet:dataset-1.csv /logdir:F:\logs
```

In order to **add more drives**, one can create a new driveset file and run the command as below. For subsequent copy sessions to the different disk drives than specified in InitialDriveset .csv file, specify a new driveset CSV file and provide it as a value to the parameter "AdditionalDriveSet". Use the **same journal file** name and provide a **new session ID**. The format of AdditionalDriveset CSV file is same as InitialDriveSet format.

```
WAImportExport.exe PrepImport /j:<JournalFile> /id:<SessionId> /AdditionalDriveSet:<driveset.csv>
```

Example

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#3 /AdditionalDriveSet:driveset-2.csv
```

In order to add additional data to the same driveset, WAImportExport tool PreImport command can be called for subsequent copy sessions to copy additional files/directory: For subsequent copy sessions to the same hard disk drives specified in InitialDriveset .csv file, specify the **same journal file** name and provide a **new session ID**; there is no need to provide the storage account key.

```
WAImportExport PrepImport /j:<JournalFile> /id:<SessionId> /j:<JournalFile> /id:<SessionId> [/logdir:<LogDirectory>] DataSet:<dataset.csv>
```

Example:

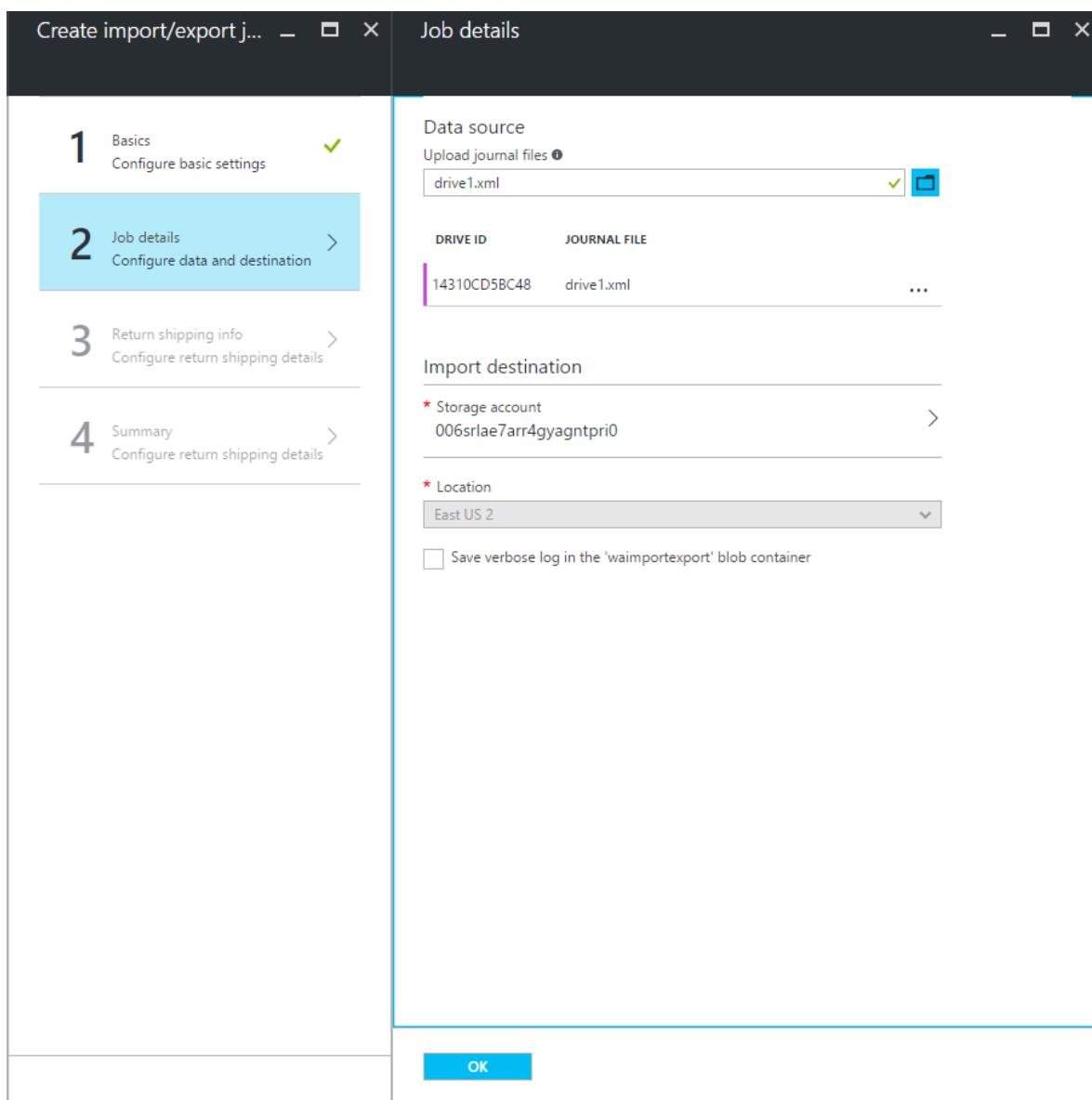
```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#2 /DataSet:dataset-2.csv
```

See more details about using the WAImportExport tool in [Preparing Hard Drives for Import](#).

Also, refer to the [Sample Workflow to Prepare Hard Drives for an Import Job](#) for more detailed step-by-step instructions.

Create the Import Job

1. Once you have prepared your drive, navigate to your storage account in the Azure portal and view the Dashboard. Under **Quick Glance**, click **Create an Import Job**. Review the steps and select the checkbox to indicate that you have prepared your drive and that you have the drive journal file available.
2. In Step 1, provide contact information for the person responsible for this import job and a valid return address. If you wish to save verbose log data for the import job, check the option to **Save the verbose log in my 'waimportexport' blob container**.
3. In Step 2, upload the drive journal files that you obtained during the drive preparation step. You will need to upload one file for each drive that you have prepared.



4. In Step 3, enter a descriptive name for the import job. Note that the name you enter may contain only lowercase letters, numbers, hyphens, and underscores, must start with a letter, and may not contain spaces. You will use the name you choose to track your jobs while they are in progress and once they are completed.

Next, select your data center region from the list. The data center region will indicate the data center and address to which you must ship your package. See the FAQ below for more information.

5. In Step 4, select your return carrier from the list and enter your carrier account number. Microsoft will use this account to ship the drives back to you once your import job is complete.

If you have your tracking number, select your delivery carrier from the list and enter your tracking number.

If you do not have a tracking number yet, choose **I will provide my shipping information for this import job once I have shipped my package**, then complete the import process.

6. To enter your tracking number after you have shipped your package, return to the **Import/Export** page for your storage account in the Azure portal, select your job from the list, and choose **Shipping Info**. Navigate through the wizard and enter your tracking number in Step 2.

If the tracking number is not updated within 2 weeks of creating the job, the job will expire.

If the job is in the Creating, Shipping or Transferring state, you can also update your carrier account number in Step 2 of the wizard. Once the job is in the Packaging state, you cannot update your carrier account number for that job.

7. You can track your job progress on the portal dashboard. See what each job state in the previous section means by [Viewing your job status](#).

How to create an Export Job?

Create an export job to notify the Import/Export service that you'll be shipping one or more empty drives to the data center so that data can be exported from your storage account to the drives and the drives then shipped to you.

Prepare your drives

Following pre-checks are recommended for preparing your drives for an export job:

1. Check the number of disks required using the WAImportExport tool's PreviewExport command. For more information, see [Previewing Drive Usage for an Export Job](#). It helps you preview drive usage for the blobs you selected, based on the size of the drives you are going to use.
2. Check that you can read/write to the hard drive that will be shipped for the export job.

Create the Export job

1. To create an export job, navigate to your storage account in the Azure portal, and view the Dashboard. Under **Quick Glance**, click **Create an Export Job** and proceed through the wizard.
2. In Step 2, provide contact information for the person responsible for this export job. If you wish to save verbose log data for the export job, check the option to **Save the verbose log in my 'waimportexport' blob container**.
3. In Step 3, specify which blob data you wish to export from your storage account to your blank drive or drives. You can choose to export all blob data in the storage account, or you can specify which blobs or sets of blobs to export.

To specify a blob to export, use the **Equal To** selector, and specify the relative path to the blob, beginning with the container name. Use `$root` to specify the root container.

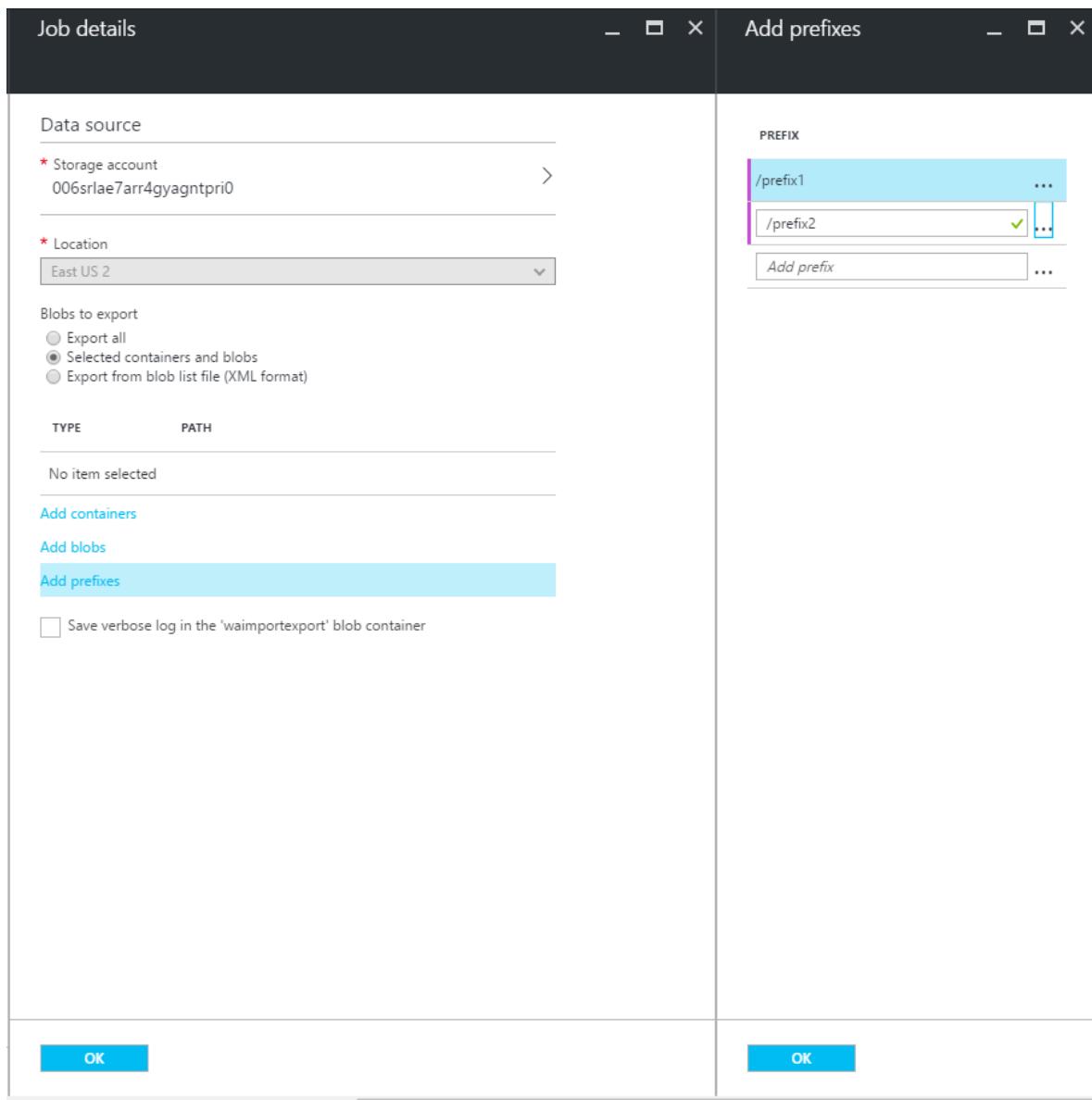
To specify all blobs starting with a prefix, use the **Starts With** selector, and specify the prefix, beginning

with a forward slash '/'. The prefix may be the prefix of the container name, the complete container name, or the complete container name followed by the prefix of the blob name.

The following table shows examples of valid blob paths:

SELECTOR	BLOB PATH	DESCRIPTION
Starts With	/	Exports all blobs in the storage account
Starts With	/\$root/	Exports all blobs in the root container
Starts With	/book	Exports all blobs in any container that begins with prefix book
Starts With	/music/	Exports all blobs in container music
Starts With	/music/love	Exports all blobs in container music that begin with prefix love
Equal To	\$root/logo.bmp	Exports blob logo.bmp in the root container
Equal To	videos/story.mp4	Exports blob story.mp4 in container videos

You must provide the blob paths in valid formats to avoid errors during processing, as shown in this screenshot.



4. In Step 4, enter a descriptive name for the export job. The name you enter may contain only lowercase letters, numbers, hyphens, and underscores, must start with a letter, and may not contain spaces.

The data center region will indicate the data center to which you must ship your package. See the FAQ below for more information.

5. In Step 5, select your return carrier from the list, and enter your carrier account number. Microsoft will use this account to ship your drives back to you once your export job is complete.

If you have your tracking number, select your delivery carrier from the list and enter your tracking number.

If you do not have a tracking number yet, choose **I will provide my shipping information for this export job once I have shipped my package**, then complete the export process.

6. To enter your tracking number after you have shipped your package, return to the **Import/Export** page for your storage account in the Azure portal, select your job from the list, and choose **Shipping Info**. Navigate through the wizard and enter your tracking number in Step 2.

If the tracking number is not updated within 2 weeks of creating the job, the job will expire.

If the job is in the Creating, Shipping or Transferring state, you can also update your carrier account number in Step 2 of the wizard. Once the job is in the Packaging state, you cannot update your carrier account number for that job.

NOTE

If the blob to be exported is in use at the time of copying to hard drive, Azure Import/Export service will take a snapshot of the blob and copy the snapshot.

7. You can track your job progress on the dashboard in the Azure portal. See what each job state means in the previous section on "Viewing your job status".
8. After you receive the drives with your exported data, you can view and copy the BitLocker keys generated by the service for your drive. Navigate to your storage account in the Azure portal and click the Import/Export tab. Select your export job from the list, and click the View Keys button. The BitLocker keys appear as shown below:

The screenshot shows the Azure Storage Explorer interface for an import/export job named 'import1 - BitLocker keys'. The left sidebar has a 'BitLocker keys' item selected, highlighted with a blue background. The main pane displays a table of BitLocker keys:

DRIVE ID	BITLOCKER KEY
14310CD5BC48	272888-439538-223245-281875-605429-085250-340747-264242
28310CD5BC50	542888-439538-223245-281875-605429-085250-340747-264268

Please go through the FAQ section below as it covers the most common questions customers encounter when using this service.

Frequently Asked Questions

Can I copy Azure Files using the Azure Import/Export service?

No, the Azure Import/Export service only supports Block Blobs and Page Blobs. All other storage types including

Azure Files, Tables, and Queues are not supported.

Is the Azure Import/Export service available for CSP subscriptions?

No, the Azure Import/Export service does not support CSP subscriptions. The support will be added in the future.

Can I skip the drive preparation step for an import job or can I prepare a drive without copying?

Any drive that you want to ship for importing data must be prepared using the Azure WAImportExport tool. You must use the WAImportExport tool to copy data to the drive.

Do I need to perform any disk preparation when creating an export job?

No, but some pre-checks are recommended. Check the number of disks required using the WAImportExport tool's PreviewExport command. For more information, see [Previewing Drive Usage for an Export Job](#). It helps you preview drive usage for the blobs you selected, based on the size of the drives you are going to use. Also check that you can read from and write to the hard drive that will be shipped for the export job.

What happens if I accidentally send an HDD which does not conform to the supported requirements?

The Azure data center will return the drive that does not conform to the supported requirements to you. If only some of the drives in the package meet the support requirements, those drives will be processed, and the drives that do not meet the requirements will be returned to you.

Can I cancel my job?

You can cancel a job when its status is Creating or Shipping.

How long can I view the status of completed jobs in the Azure portal?

You can view the status for completed jobs for up to 90 days. Completed jobs will be deleted after 90 days.

If I want to import or export more than 10 drives, what should I do?

One import or export job can reference only 10 drives in a single job for the Import/Export service. If you want to ship more than 10 drives, you can create multiple jobs. Drives that are associated with the same job must be shipped together in the same package.

Does the service format the drives before returning them?

No. All drives are encrypted with BitLocker.

Can I purchase drives for import/export jobs from Microsoft?

No. You will need to ship your own drives for both import and export jobs.

After the import job completes, what will my data look like in the storage account? Will my directory hierarchy be preserved?

When preparing a hard drive for an import job, the destination is specified by the DstBlobPathOrPrefix field in dataset CSV. This is the destination container in the storage account to which data from the hard drive is copied. Within this destination container, virtual directories are created for folders from the hard drive and blobs are created for files.

If the drive has files that already exist in my storage account, will the service overwrite existing blobs in my storage account?

When preparing the drive, you can specify whether the destination files should be overwritten or ignored using the field in dataset CSV file called Disposition. By default, the service will rename the new files rather than overwrite existing blobs.

Is the WAImportExport tool compatible with 32-bit operating systems? No. The WAImportExport tool is

only compatible with 64-bit Windows operating systems. Please refer to the Operating Systems section in the [pre-requisites](#) for a complete list of supported OS versions.

Should I include anything other than the hard disk drive in my package?

Please ship only your hard drives. Do not include items like power supply cables or USB cables.

Do I have to ship my drives using FedEx or DHL?

You can ship drives to the data center using any known carrier like FedEx, DHL, UPS, or US Postal Service. However, for shipping the drives back to you from the data center, you must provide a FedEx account number in the US and EU, or a DHL account number in the Asia and Australia regions.

Are there any restrictions with shipping my drive internationally?

Please note that the physical media that you are shipping may need to cross international borders. You are responsible for ensuring that your physical media and data are imported and/or exported in accordance with the applicable laws. Before shipping the physical media, check with your advisors to verify that your media and data can legally be shipped to the identified data center. This will help to ensure that it reaches Microsoft in a timely manner.

When creating a job, the shipping address is a location that is different from my storage account location. What should I do?

Some storage account locations are mapped to alternate shipping locations. Previously available shipping locations can also be temporarily mapped to alternate locations. Always check the shipping address provided during job creation before shipping your drives.

When shipping my drive, the carrier asks for the data center contact name and phone number. What should I provide?

The phone number is provided to you during job creation. If you need a contact name, please contact us at waimportexport@microsoft.com and we will provide you with that information.

Can I use the Azure Import/Export service to copy PST mailboxes and SharePoint data to O365?

Please refer to [Import PST files or SharePoint data to Office 365](#).

Can I use the Azure Import/Export service to copy my backups offline to the Azure Backup Service?

Please refer to [Offline Backup workflow in Azure Backup](#).

See also:

- [Setting up the WAImportExport tool](#)
- [Transfer data with the AzCopy command-line utility](#)
- [Azure Import Export REST API Sample](#)

Using the Azure Import/Export Tool

1/17/2017 • 1 min to read • [Edit on GitHub](#)

The Azure Import/Export Tool (WAIImportExport.exe) is used to create and manage jobs for the Azure Import/Export Service, enabling you to transfer large amounts of data into or out of Azure Blob Storage.

This documentation is for the most recent version of the Azure Import/Export Tool. For information about using the v1 tool, please see [Using the Azure Import/Export Tool v1](#).

In these articles, you will see how to use the tool to do the following:

- Install and set up the Import/Export Tool.
- Prepare your hard drives for a job where you import data from your drives to Azure Blob Storage.
- Review the status of a job with Copy Log Files.
- Repair an import job.
- Repair an export job.
- Troubleshoot the Azure Import/Export Tool, in case you had a problem during process.

Setting up the Azure Import/Export tool

1/17/2017 • 5 min to read • [Edit on GitHub](#)

The Microsoft Azure Import/Export tool is the drive preparation and repair tool that you can use with the Microsoft Azure Import/Export service. You can use the tool for the following functions:

- Before creating an import job, you can use this tool to copy data to the hard drives you are going to ship to an Azure data center.
- After an import job has completed, you can use this tool to repair any blobs that were corrupted, were missing, or conflicted with other blobs.
- After you receive the drives from a completed export job, you can use this tool to repair any files that were corrupted or missing on the drives.

Prerequisites

If you are **preparing drives** for an import job, you will need to meet the following prerequisites:

- You must have an active Azure subscription.
- Your subscription must include a storage account with enough available space to store the files you are going to import.
- You need at least one of the account keys for the storage account.
- You need a computer (the "copy machine") with Windows 7, Windows Server 2008 R2, or a newer Windows operating system installed.
- The .NET Framework 4 must be installed on the copy machine.
- BitLocker must be enabled on the copy machine.
- You will need one or more empty 3.5-inch SATA hard drives connected to the copy machine.
- The files you plan to import must be accessible from the copy machine, whether they are on a network share or a local hard drive.

If you are attempting to **repair an import** that has partially failed, you will need:

- The copy log files
- The storage account key

If you are attempting to **repair an export** that has partially failed, you will need:

- The copy log files
- The manifest files (optional)
- The storage account key

Installing the Azure Import/Export tool

First, [download the Azure Import/Export tool](#) and extract it to a directory on your computer, for example

`c:\WATImportExport`

The Azure Import/Export tool consists of the following files:

- dataset.csv
- driveset.csv
- hddid.dll

- Microsoft.Data.Services.Client.dll
- Microsoft.WindowsAzure.Storage.dll
- WAImportExport.exe
- WAImportExport.exe.config
- WAImportExportCore.dll
- WAImportExportRepair.dll

Next, open a Command Prompt window in **Administrator mode**, and change into the directory containing the extracted files.

To output help for the command, run the tool without parameters:

```
WAImportExport, a client tool for Windows Azure Import/Export Service. Microsoft (c) 2013

Copy directories and/or files with a new copy session:
WAImportExport.exe PrepImport
/j:<JournalFile>
/id:<SessionId> [/logdir:<LogDirectory>]
[sk:<StorageAccountKey>]
[/silentmode]
[/InitialDriveSet:<driveset.csv>]
DataSet:<dataset.csv>

Add more drives:
WAImportExport.exe PrepImport
/j:<JournalFile>
/id:<SessionId>
/AdditionalDriveSet:<driveset.csv>

Abort an interrupted copy session:
WAImportExport.exe PrepImport
/j:<JournalFile>
/id:<SessionId>
/AbortSession

Resume an interrupted copy session:
WAImportExport.exe PrepImport
/j:<JournalFile>
/id:<SessionId>
/ResumeSession

List drives:
WAImportExport.exe PrepImport /j:<JournalFile> /ListDrives

List copy sessions:
WAImportExport.exe PrepImport /j:<JournalFile> /ListCopySessions

Repair a Drive:
WAImportExport.exe RepairImport | RepairExport
/r:<RepairFile> [/logdir:<LogDirectory>]
[d:<TargetDirectories>] [/bk:<BitLockerKey>]
/sn:<StorageAccountName> /sk:<StorageAccountKey>
[/CopyLogFile:<DriveCopyLogFile>] [/ManifestFile:<DriveManifestFile>]
[/PathMapFile:<DrivePathMapFile>]

Preview an Export Job:
WAImportExport.exe PreviewExport
[/logdir:<LogDirectory>]
/sn:<StorageAccountName> /sk:<StorageAccountKey>
/ExportBlobListFile:<ExportBlobListFile> /DriveSize:<DriveSize>

Parameters:
/j:<JournalFile>
```

- Required. Path to the journal file. A journal file tracks a set of drives and records the progress in preparing these drives. The journal file must always be specified.

/logdir:<LogDirectory>

- Optional. The log directory. Verbose log files as well as some temporary files will be written to this directory. If not specified, current directory will be used as the log directory. The log directory can be specified only once for the same journal file.

/id:<SessionId>

- Optional. The session Id is used to identify a copy session. It is used to ensure accurate recovery of an interrupted copy session.

/ResumeSession

- Optional. If the last copy session was terminated abnormally, this parameter can be specified to resume the session.

/AbortSession

- Optional. If the last copy session was terminated abnormally, this parameter can be specified to abort the session.

/sn:<StorageAccountName>

- Required. Only applicable for RepairImport and RepairExport. The name of the storage account.

/sk:<StorageAccountKey>

- Required. The key of the storage account.

/InitialDriveSet:<driveset.csv>

- Required. A .csv file that contains a list of drives to prepare.

/AdditionalDriveSet:<driveset.csv>

- Required. A .csv file that contains a list of additional drives to be added.

/r:<RepairFile>

- Required. Only applicable for RepairImport and RepairExport.
Path to the file for tracking repair progress. Each drive must have one and only one repair file.

/d:<TargetDirectories>

- Required. Only applicable for RepairImport and RepairExport.
For RepairImport, one or more semicolon-separated directories to repair;
For RepairExport, one directory to repair, e.g. root directory of the drive.

/CopyLogFile:<DriveCopyLogFile>

- Required. Only applicable for RepairImport and RepairExport. Path to the drive copy log file (verbose or error).

/ManifestFile:<DriveManifestFile>

- Required. Only applicable for RepairExport. Path to the drive manifest file.

/PathMapFile:<DrivePathMapFile>

- Optional. Only applicable for RepairImport. Path to the file containing mappings of file paths relative to the drive root to locations of actual files (tab-delimited). When first specified, it will be populated with file paths with empty targets, which means either they are not found in TargetDirectories, access denied, with invalid name, or they exist in multiple directories. The path map file can be manually edited to include the correct target paths and specified again for the tool to resolve the file paths correctly.

/ExportBlobListFile:<ExportBlobListFile>

- Required. Path to the XML file containing list of blob paths or blob path prefixes for the blobs to be exported. The file format is the same as the blob list blob format in the Put Job operation of the Import/Export Service REST API.

/DriveSize:<DriveSize>

- Required. Size of drives to be used for export. For example, 500GB, 1.5TB.
Note: 1 GB = 1,000,000,000 bytes
1 TB = 1,000,000,000,000 bytes

/DataSet:<dataset.csv>

- Required. A .csv file that contains a list of directories and/or a list files to be copied to target drives.

/silentmode

- Optional. If not specified, it will remind you the requirement of drives and need your confirmation to continue.

Examples:

Copy a data set to a drive:
 WAImportExport.exe PrepImport
 /j:9WM35C2V.jrn /id:session#1 /sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN0lp+kp0C9MEdx3GEL

```
xmBw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ== /InitialDriveSet:driveset1.csv  
/DataSet:dataset1.csv  
  
Copy another dataset to the same drive following the above command:  
WAImportExport.exe PrepImport /j:9WM35C2V.jrn /id:session#2 /DataSet:dataset2.csv  
  
Preview how many 1.5 TB drives are needed for an export job:  
WAImportExport.exe PreviewExport  
/sn:mytestaccount /sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN0lp+kp0C9MEdx3GELxmBw4hK94f7K  
ysbbeKLDksg7VoN1W/a5UuM2zNgQ== /ExportBlobListFile:C:\temp\myexportbloblist.xml  
/DriveSize:1.5TB  
  
Repair an finished import job:  
WAImportExport.exe RepairImport  
/r:9WM35C2V.rep /d:X:\ /bk:442926-020713-108086-436744-137335-435358-242242-2795  
98 /sn:mytestaccount /sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN0lp+kp0C9MEdx3GELxmBw4hK94  
f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ== /CopyLogFile:C:\temp\9WM35C2V_error.log
```

Next steps

- [Preparing hard drives for an import job](#)
- [Previewing drive usage for an export job](#)
- [Reviewing job status with copy log files](#)
- [Repairing an import job](#)
- [Repairing an export job](#)
- [Troubleshooting the Azure Import/Export tool](#)

Preparing hard drives for an Import Job

1/17/2017 • 19 min to read • [Edit on GitHub](#)

Overview

The WAImportExport tool is the drive preparation and repair tool that you can use with the [Microsoft Azure Import/Export service](#). You can use this tool to copy data to the hard drives you are going to ship to an Azure datacenter. After an import job has completed, you can use this tool to repair any blobs that were corrupted, were missing, or conflicted with other blobs. After you receive the drives from a completed export job, you can use this tool to repair any files that were corrupted or missing on the drives. In this article we will go over the working of this tool.

Prerequisites

Prerequisites for running WAImportExport.exe

- **Machine configuration**

- Windows 7, Windows Server 2008 R2, or a newer Windows operating system
- .NET Framework 4 must be installed. See [FAQ](#) on how to check if .Net Framework is installed on the machine.

- **Storage account key** - You need at least one of the account keys for the storage account.

Preparing disk for Import Job

- **BitLocker** - BitLocker must be enabled on the machine which is running WAImportExport Tool. See [FAQ](#) for how to enable BitLocker
- **Disk**s accessible from machine on which WAImportExport Tool is run. See [FAQ](#) for disk specification.
- **Source files** - The files you plan to import must be accessible from the copy machine, whether they are on a network share or a local hard drive.

Repairing a partially failed import job

- **Copy log files** that is generated when Azure Import/Export Service copies data between Storage Account and Disk. It is located in your target storage account.

Repairing a partially failed export job

- **Copy log file** that is generated when Azure Import/Export Service copies data between Storage Account and Disk. It is located in your source storage account.
- **Manifest file** - [Optional] Located on exported drive that was returned by Microsoft.

Download and install WAImportExport

Download the [latest version of WAImportExport.exe](#). Extract the zipped content to a directory on your computer.

Your next task is to create CSV files.

Prepare the dataset CSV file

What is dataset CSV

Dataset CSV file is the value of /dataset flag is a CSV file that contains a list of directories and/or a list files to be copied to target drives. The first step to creating an import job is to determine which directories and files you are going to import. This can be a list of directories, a list of unique files, or a combination of those two. When a

directory is included, all files in the directory and its subdirectories will be part of the import job.

For each directory or file that will be imported, you need to identify a destination virtual directory or blob in the Azure Blob service. You will use these targets as inputs to the WAImportExport tool. Note that directories should be delimited with the forward slash character "/".

The following table shows some examples of blob targets:

SOURCE FILE OR DIRECTORY	DESTINATION BLOB OR VIRTUAL DIRECTORY
H:\Video	https://mystorageaccount.blob.core.windows.net/video
H:\Photo	https://mystorageaccount.blob.core.windows.net/photo
K:\Temp\FavoriteVideo.ISO	https://mystorageaccount.blob.core.windows.net/favorite/FavoriteVideo.ISO
\myshare\john\music	https://mystorageaccount.blob.core.windows.net/music

Sample dataset.csv

```
BasePath,DstBlobPathOrPrefix,BlobType,Disposition,MetadataFile,PropertiesFile
"F:\50M_original\100M_1.csv.txt","containername/100M_1.csv.txt",BlockBlob,rename,"None",None
"F:\50M_original\","containername/",BlockBlob,rename,"None",None
```

Dataset CSV file fields

FIELD	DESCRIPTION
BasePath	[Required] The value of this parameter represents the source where the data to be imported is located. The tool will recursively copy all data located under this path. Allowed Values: This has to be a valid path on local computer or a valid share path and should be accessible by the user. The directory path must be an absolute path (not a relative path). If the path ends with "\", it represents a directory else a path ending without "\" represents a file. No regex are allowed in this field. If the path contains spaces, put it in "". Example: "c\Directory\c\Directory\File.txt" "\FBaseFilesharePath.domain.net\sharename\directory 1"

FIELD	DESCRIPTION
DstBlobPathOrPrefix	<p>[Required]</p> <p>The path to the destination virtual directory in your Windows Azure storage account. The virtual directory may or may not already exist. If it does not exist, Import/Export Service will create one.</p> <p>Be sure to use valid container names when specifying destination virtual directories or blobs. Keep in mind that container names must be lowercase. For container naming rules, see Naming and Referencing Containers, Blobs, and Metadata. If only root is specified, the directory structure of the source is replicated in the destination blob container. If a different directory structure is desired than the one in source, multiple rows of mapping in CSV</p> <p>You can specify a container, or a blob prefix like music/70s/. The destination directory must begin with the container name, followed by a forward slash "/", and optionally may include a virtual blob directory that ends with "/".</p> <p>When the destination container is the root container, you must explicitly specify the root container, including the forward slash, as \$root/. Since blobs under the root container cannot include "/" in their names, any subdirectories in the source directory will not be copied when the destination directory is the root container.</p> <p>Example If the destination blob path is https://mystorageaccount.blob.core.windows.net/video, the value of this field can be video/</p>
BlobType	<p>[Optional] block page</p> <p>Currently Import/Export Service supports 2 kinds of Blobs. Page blobs and Block Blobs. By default all files will be imported as Block Blobs. And *.vhd and *.vhdx will be imported as Page Blobs. There is a limit on the block-blob and page-blob allowed size. See Storage scalability targets for more information.</p>
Disposition	<p>[Optional] rename no-overwrite overwrite</p> <p>This field specifies the copy-behavior during import i.e when data is being uploaded to the storage account from the disk. Available options are: rename overwrite no-overwrite. Defaults to "rename" if nothing specified.</p> <p>Rename: If the object with same name present, creates a copy in destination.</p> <p>Overwrite: overwrites the file with newer file. The file with last-modified wins.</p> <p>No-overwrite: Skips writing the file if already present.</p>
MetadataFile	<p>[Optional]</p> <p>The value to this field is the metadata file which can be provided if the one needs to preserve the metadata of the objects or provide custom metadata. Path to the metadata file for the destination blobs. See Import-Export Service Metadata and Properties File Format for more information</p>

FIELD	DESCRIPTION
PropertiesFile	<p>[Optional] Path to the property file for the destination blobs. See Import-Export Service Metadata and Properties File Format for more information.</p>

Prepare InitialDriveSet or AdditionalDriveSet CSV file

What is driveset CSV

The value of the /InitialDriveSet or /AdditionalDriveSet flag is a CSV file which contains the list of disks to which the drive letters are mapped so that the tool can correctly pick the list of disks to be prepared. If the data size is greater than a single disk size, the WAImportExport tool will distribute the data across multiple disks enlisted in this CSV file in an optimized way.

There is no limit on the number of disks the data can be written to in a single session. The tool will distribute data based on disk size and folder size. It will select the disk which is most optimized for the object-size. The data when uploaded to the storage account will be converged back to the directory structure which was specified in dataset file. In order to create a driveset CSV, follow the steps below.

Create basic volume and assign drive letter

In order to create a basic volume and assign a drive letter, by following the instructions for "Simpler partition creation" given at [Overview of Disk Management](#).

Sample InitialDriveSet and AdditionalDriveSet CSV file

```
DriveLetter,FormatOption,SilentOrPromptOnFormat,Encryption,ExistingBitLockerKey
G,AlreadyFormatted,SilentMode,AlreadyEncrypted,060456-014509-132033-080300-252615-584177-672089-411631
H,Format,SilentMode,Encrypt,
```

Driveset CSV file fields

FIELDS	VALUE
DriveLetter	<p>[Required] Each drive that is being provided to the tool as the destination needs have a simple NTFS volume on it and a drive letter assigned to it.</p> <p>Example: R or r</p>
FormatOption	<p>[Required] Format AlreadyFormatted</p> <p>Format: Specifying this will format all the data on the disk. AlreadyFormatted: The tool will skip formatting when this value is specified.</p>
SilentOrPromptOnFormat	<p>[Required] SilentMode PromptOnFormat</p> <p>SilentMode: Providing this value will enable user to run the tool in Silent Mode. PromptOnFormat: The tool will prompt the user to confirm whether the action is really intended at every format.</p> <p>If not set, command will abort and prompt error message: "Incorrect value for SilentOrPromptOnFormat: none"</p>

FIELDS	VALUE
Encryption	<p>[Required] Encrypt AlreadyEncrypted The value of this field decides which disk to encrypt and which not to.</p> <p>Encrypt: Tool will format the drive. If value of "FormatOption" field is "Format" then this value is required to be "Encrypt". If "AlreadyEncrypted" is specified in this case, it will result into an error "When Format is specified, Encrypt must also be specified".</p> <p>AlreadyEncrypted: Tool will decrypt the drive using the BitLockerKey provided in "ExistingBitLockerKey" Field. If value of "FormatOption" field is "AlreadyFormatted", then this value can be either "Encrypt" or "AlreadyEncrypted"</p>
ExistingBitLockerKey	<p>[Required] If value of "Encryption" field is "AlreadyEncrypted" The value of this field is the BitLocker key which is associated with the particular disk.</p> <p>This field should be left blank if the value of "Encryption" field is "Encrypt". If BitLocker Key is specified in this case, it will result into an error "Bitlocker Key should not be specified".</p> <p>Example: 060456-014509-132033-080300-252615-584177-672089-411631</p>

Preparing disk for import job

To prepare drives for an import job, call the WAImportExport tool with the **PreImport** command. Which parameters you include depends on whether this is the first copy session, or a subsequent copy session.

First session

First Copy Session to Copy a Single/Multiple Directory to a single/multiple Disk (depending on what is specified in CSV file) WAImportExport tool PreImport command for the first copy session to copy directories and/or files with a new copy session:

```
WAImportExport.exe PrepImport /j:<JournalFile> /id:<SessionId> [/logdir:<LogDirectory>] [/sk:<StorageAccountKey>] [/silentmode] [/InitialDriveSet:<driveset.csv>] DataSet:<dataset.csv>
```

Example:

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#1 /sk:***** /InitialDriveSet:driveset-1.csv /DataSet:dataset-1.csv /logdir:F:\logs
```

Add data in subsequent session

In subsequent copy sessions, you do not need to specify the initial parameters. You need to use the same journal file in order for the tool to remember where it left in the previous session. The state of the copy session is written to the journal file. Here is the syntax for a subsequent copy session to copy additional directories and or files:

```
WAImportExport.exe PrepImport /j:<SameJournalFile> /id:<DifferentSessionId> [DataSet:<differentdataset.csv>]
```

Example:

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#2 /DataSet:dataset-2.csv
```

Add drives to latest session

If the data did not fit in specified drives in InitialDriveset, one can use the tool to add additional drives to same copy session.

NOTE

The session id should match the previous session id. Journal file should match the one specified in previous session.

```
WAImportExport.exe PrepImport /j:<SameJournalFile> /id:<SameSessionId> /AdditionalDriveSet:<newdriveset.csv>
```

Example:

```
WAImportExport.exe PrepImport /j:SameJournalTest.jrn /id:session#2 /AdditionalDriveSet:driveset-2.csv
```

Abort the latest session:

If a copy session is interrupted and it is not possible to resume (for example, if a source directory proved inaccessible), you must abort the current session so that it can be rolled back and new copy sessions can be started:

```
WAImportExport.exe PrepImport /j:<SameJournalFile> /id:<SameSessionId> /AbortSession
```

Example:

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#2 /AbortSession
```

Only the last copy session, if terminated abnormally, can be aborted. Note that you cannot abort the first copy session for a drive. Instead you must restart the copy session with a new journal file.

Resume a latest interrupted session:

If a copy session is interrupted for any reason, you can resume it by running the tool with only the journal file specified:

```
WAImportExport.exe PrepImport /j:<SameJournalFile> /id:<SameSessionId> /ResumeSession
```

Example:

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#2 /ResumeSession
```

IMPORTANT

When you resume a copy session, do not modify the source data files and directories by adding or removing files.

WAImportExport Parameters

PARAMETERS	DESCRIPTION
/j:<JournalFile>	Required Path to the journal file. A journal file tracks a set of drives and records the progress in preparing these drives. The journal file must always be specified.

PARAMETERS	DESCRIPTION
/logdir:<LogDirectory>	<p>Optional. The log directory. Verbose log files as well as some temporary files will be written to this directory. If not specified, current directory will be used as the log directory. The log directory can be specified only once for the same journal file.</p>
/id:<SessionId>	<p>Required The session Id is used to identify a copy session. It is used to ensure accurate recovery of an interrupted copy session.</p>
/ResumeSession	<p>Optional. If the last copy session was terminated abnormally, this parameter can be specified to resume the session.</p>
/AbortSession	<p>Optional. If the last copy session was terminated abnormally, this parameter can be specified to abort the session.</p>
/sn:<StorageAccountName>	<p>Required Only applicable for RepairImport and RepairExport. The name of the storage account.</p>
/sk:<StorageAccountKey>	<p>Required The key of the storage account.</p>
/InitialDriveSet:<driveset.csv>	<p>Required When running the first copy session A CSV file that contains a list of drives to prepare.</p>
/AdditionalDriveSet:<driveset.csv>	<p>Required. When adding drives to current copy session. A CSV file that contains a list of additional drives to be added.</p>
/r:<RepairFile>	<p>Required Only applicable for RepairImport and RepairExport. Path to the file for tracking repair progress. Each drive must have one and only one repair file.</p>
/d:<TargetDirectories>	<p>Required. Only applicable for RepairImport and RepairExport. For RepairImport, one or more semicolon-separated directories to repair; For RepairExport, one directory to repair, e.g. root directory of the drive.</p>
/CopyLogFile:<DriveCopyLogFile>	<p>Required Only applicable for RepairImport and RepairExport. Path to the drive copy log file (verbose or error).</p>
/ManifestFile:<DriveManifestFile>	<p>Required Only applicable for RepairExport. Path to the drive manifest file.</p>
/PathMapFile:<DrivePathMapFile>	<p>Optional. Only applicable for RepairImport. Path to the file containing mappings of file paths relative to the drive root to locations of actual files (tab-delimited). When first specified, it will be populated with file paths with empty targets, which means either they are not found in TargetDirectories, access denied, with invalid name, or they exist in multiple directories. The path map file can be manually edited to include the correct target paths and specified again for the tool to resolve the file paths correctly.</p>

PARAMETERS	DESCRIPTION
/ExportBlobListFile:<ExportBlobListFile>	Required. Only applicable for PreviewExport. Path to the XML file containing list of blob paths or blob path prefixes for the blobs to be exported. The file format is the same as the blob list blob format in the Put Job operation of the Import/Export Service REST API.
/DriveSize:<DriveSize>	Required. Only applicable for PreviewExport. Size of drives to be used for export. For example, 500GB, 1.5TB. Note: 1 GB = 1,000,000,000 bytes 1 TB = 1,000,000,000,000 bytes
/DataSet:<dataset.csv>	Required A CSV file that contains a list of directories and/or a list files to be copied to target drives.
/silentmode	Optional. If not specified, it will remind you the requirement of drives and need your confirmation to continue.

Tool Output

Sample Drive Manifest file

```

<?xml version="1.0" encoding="UTF-8"?>
<DriveManifest Version="2011-MM-DD">
    <Drive>
        <DriveId>drive-id</DriveId>
        <StorageAccountKey>storage-account-key</StorageAccountKey>
        <ClientCreator>client-creator</ClientCreator>
        <!-- First Blob List -->
        <BlobList Id="session#1-0">
            <!-- Global properties and metadata that applies to all blobs -->
            <MetadataPath Hash="md5-hash">global-metadata-file-path</MetadataPath>
            <PropertiesPath Hash="md5-hash">global-properties-file-path</PropertiesPath>
            <!-- First Blob -->
            <Blob>
                <BlobPath>blob-path-relative-to-account</BlobPath>
                <FilePath>file-path-relative-to-transfer-disk</FilePath>
                <ClientData>client-data</ClientData>
                <Length>content-length</Length>
                <ImportDisposition>import-disposition</ImportDisposition>
                <!-- page-range-list-or-block-list -->
                <!-- page-range-list -->
                <PageRangeList>
                    <PageRange Offset="1073741824" Length="512" Hash="md5-hash" />
                    <PageRange Offset="1073741824" Length="512" Hash="md5-hash" />
                </PageRangeList>
                <!-- block-list -->
                <BlockList>
                    <Block Offset="1073741824" Length="4194304" Id="block-id" Hash="md5-hash" />
                    <Block Offset="1073741824" Length="4194304" Id="block-id" Hash="md5-hash" />
                </BlockList>
                <MetadataPath Hash="md5-hash">metadata-file-path</MetadataPath>
                <PropertiesPath Hash="md5-hash">properties-file-path</PropertiesPath>
            </Blob>
        </BlobList>
    </Drive>
</DriveManifest>

```

Sample journal file for each drive: ending with .xml

```
[BeginUpdateRecord][2016/11/01 21:22:25.379][Type:ActivityRecord]
ActivityId: DriveInfo
DriveState: [BeginValue]
<?xml version="1.0" encoding="UTF-8"?>
<Drive>
<DriveId>drive-id</DriveId>
<BitLockerKey>*****</BitLockerKey>
<ManifestFile>\DriveManifest.xml</ManifestFile>
<ManifestHash>D863FE44F861AE0DA4DCEAEEFFCCCE68</ManifestHash> </Drive>
[EndValue]
SaveCommandOutput: Completed
[EndUpdateRecord]
```

Sample journal file for session: ended with .jrn which records the trail of sessions

```
[BeginUpdateRecord][2016/11/02 18:24:14.735][Type>NewJournalFile]
VocabularyVersion: 2013-02-01
[EndUpdateRecord]
[BeginUpdateRecord][2016/11/02 18:24:14.749][Type:ActivityRecord]
ActivityId: PrepImportDriveCommandContext
LogDirectory: F:\logs
[EndUpdateRecord]
[BeginUpdateRecord][2016/11/02 18:24:14.754][Type:ActivityRecord]
ActivityId: PrepImportDriveCommandContext
StorageAccountKey: *****
[EndUpdateRecord]
```

FAQ

General

What is WAImportExport tool?

The WAImportExport tool is the drive preparation and repair tool that you can use with the Microsoft Azure Import/Export Service. You can use this tool to copy data to the hard drives you are going to ship to an Azure data center. After an import job has completed, you can use this tool to repair any blobs that were corrupted, were missing, or conflicted with other blobs. After you receive the drives from a completed export job, you can use this tool to repair any files that were corrupted or missing on the drives.

How does the WAImportExport tool work on multiple source dir and disks?

If the data size is greater than the disk size, the WAImportExport tool will distribute the data across the disks in an optimized way. The data copy to multiple disks can be done in parallel or sequentially. There is no limit on the number of disks the data can be written to simultaneously. The tool will distribute data based on disk size and folder size. It will select the disk which is most optimized for the object-size. The data when uploaded to the storage account will be converged back to the specified directory structure.

Where can I find previous version of WAImportExport tool?

WAImportExport tool has all functionalities that WAImportExport V1 tool had. WAImportExport tool allows users to specify multiple source and write to multiple drives. Additionally, one can easily manage multiple source locations from which the data needs to be copied in a single CSV file. However, in case you need SAS support or want to copy single source to single disk, you can [download WAImportExport V1 Tool](#) and refer to [WAImportExport V1 Reference](#) for help with WAImportExport V1 usage.

What is a Session ID?

The tool expects the copy session (/id) parameter to be the same if the intent is to spread the data across multiple disks. Maintaining the same name of the copy session will enable user to copy data from one or multiple source locations into one or multiple destination disks/directories. Maintaining same session id enables the tool to pick up the copy of files from where it was left the last time.

However, same copy session cannot be used to import data to different storage accounts.

When copy-session name is same across multiple runs of the tool, the logfile (/logdir) and storage account key (/sk) is also expected to be the same.

SessionId can consist of letters, 0~9, underscore (_), dash (-) or hash (#), and its length must be 3~30.

e.g. session-1 or session#1 or session_1

What is a journal file?

Each time you run the WAImportExport tool to copy files to the hard drive, the tool creates a copy session. The state of the copy session is written to the journal file. If a copy session is interrupted (for example, due to a system power loss), it can be resumed by running the tool again and specifying the journal file on the command line.

For each hard drive that you prepare with the Azure Import/Export tool, the tool will create a single journal file with name "<DriveID>.xml" where drive Id is the serial number associated to the drive that the tool reads from the disk. You will need the journal files from all of your drives to create the import job. The journal file can also be used to resume drive preparation if the tool is interrupted.

What is a log directory?

The log directory specifies a directory to be used to store verbose logs as well as temporary manifest files. If not specified, the current directory will be used as the log directory. The log are verbose logs.

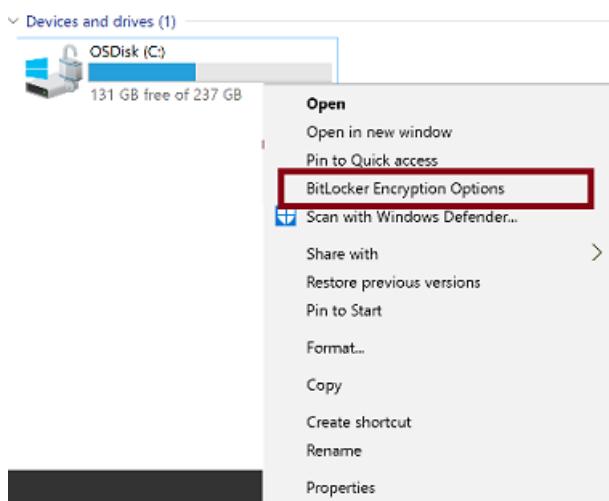
Prerequisites

What are the specifications of my disk?

One or more empty 2.5 or 3.5-inch SATAII or III or SSD hard drives connected to the copy machine.

How can I enable BitLocker on my machine?

Simple way to check is by right-clicking on System drive. It will show you options for Bitlocker if the capability is turned on. If it is off, you wont see it.



Here is an article on [how to enable BitLocker](#)

It is possible that your machine does not have tpm chip. If you do not get an output using tpm.msc, look at the next FAQ.

How to disable Trusted Platform Module (TPM) in BitLocker?

NOTE

Only if there is no TPM in their servers, you need to disable TPM policy. It is not necessary to disable TPM if there is a trusted TPM in user's server.

In order to disable TPM in BitLocker, go through the following steps:

1. Launch **Group Policy Editor** by typing gpedit.msc on a command prompt. If **Group Policy Editor** appears to

be unavailable, for enabling BitLocker first. See previous FAQ.

2. Open **Local Computer Policy > Computer Configuration > Administrative Templates > Windows Components > BitLocker Drive Encryption > Operating System Drives**.
3. Edit **Require additional authentication at startup** policy.
4. Set the policy to **Enabled** and make sure **Allow BitLocker without a compatible TPM** is checked.

How to check if .Net 4 or higher version is installed on my machine?

All Microsoft .NET Framework versions are installed in following directory: %windir%\Microsoft.NET\Framework\

Navigate to the above mentioned part on your target machine where the tool needs to run. Look for folder name starting with "v4". Absence of such a directory means .Net v4 is not installed on your machine. You can download .Net 4 on your machine using [Microsoft .NET Framework 4 \(Web Installer\)](#).

Limits

How many drives can I Prepare/send at the same time?

There is no limit on the number of disks that the tool can prepare. However, the tool expects drive letters as inputs. That limits it to 25 simultaneous disk preparation. A single job can handle maximum of 10 disks at a time. If you prepare more than 10 disks targeting the same storage account, the disks can be distributed across multiple jobs.

Can I target more than one storage account?

Only one storage account can be submitted per job and in single copy session.

Capabilities

Does WAImportExport.exe encrypt my data?

Yes. BitLocker encryption is enabled and required for this process.

What will be the hierarchy of my data when it appears in the storage account?

Although data is distributed across disks, the data when uploaded to the storage account will be converged back to the directory structure specified in the dataset CSV file.

How many of the input disks will have active IO in parallel, when copy is in progress?

The tool distributes data across the input disks based on the size of the input files. That said, the number of active disks in parallel completely depends on the nature of the input data. Depending on the size of individual files in the input dataset, one or more disks may show active IO in parallel. See next FAQ for more details.

How does the tool distribute the files across the disks?

WAImportExport Tool reads and writes files batch by batch, one batch contains max of 100000 files. This means that max 100000 files can be written parallel. Multiple disks are written to simultaneously if these 100000 files are distributed to multi drives. However whether the tool writes to multiple disk simultaneously or a single disk depends on the cumulative size of the batch. For instance, in case of smaller files, if all of 10,0000 files are able to fit in a single drive, tool will write to only one disk during the processing of this batch.

WAImportExport Output

There are two journal files. Which one should I upload to Azure Portal?

.xml - For each hard drive that you prepare with the WAImportExport tool, the tool will create a single journal file with name "<DriveID>.xml" where drive Id is the serial number associated to the drive that the tool reads from the disk. You will need the journal files from all of your drives to create the import job in the Azure portal. This journal file can also be used to resume drive preparation if the tool is interrupted.

.jrn - The journal file with suffix jrn contains the status for all copy sessions for a hard drives. It also contains the information needed to create the import job. You must always specify a journal file when running the WAImportExport tool, as well as a copy session ID.

Setting properties and metadata during the import process

1/17/2017 • 1 min to read • [Edit on GitHub](#)

When you run the Microsoft Azure Import/Export tool to prepare your drives, you can specify properties and metadata to be set on the destination blobs. Follow these steps:

1. To set blob properties, create a text file on your local computer that specifies property names and values.
2. To set blob metadata, create a text file on your local computer that specifies metadata names and values.
3. Pass the full path to one or both of these files to the Azure Import/Export tool as part of the `PrepImport` operation.

NOTE

When you specify a properties or metadata file as part of a copy session, those properties or metadata are set for every blob that is imported as part of that copy session. If you want to specify a different set of properties or metadata for some of the blobs being imported, you'll need to create a separate copy session with different properties or metadata files.

Specify blob properties in a text file

To specify blob properties, create a local text file, and include XML that specifies property names as elements, and property values as values. Here's an example that specifies some property values:

```
<?xml version="1.0" encoding="UTF-8"?>
<Properties>
  <Content-Type>application/octet-stream</Content-Type>
  <Content-MD5>Q2h1Y2sgSW50ZWdyXR5IQ==</Content-MD5>
  <Cache-Control>no-cache</Cache-Control>
</Properties>
```

Save the file to a local location like `C:\WAImportExport\ImportProperties.txt`.

Specify blob metadata in a text file

Similarly, to specify blob metadata, create a local text file that specifies metadata names as elements, and metadata values as values. Here's an example that specifies some metadata values:

```
<?xml version="1.0" encoding="UTF-8"?>
<Metadata>
  <UploadMethod>Windows Azure Import/Export Service</UploadMethod>
  <DataSetName>SampleData</DataSetName>
  <CreationDate>10/1/2013</CreationDate>
</Metadata>
```

Save the file to a local location like `C:\WAImportExport\ImportMetadata.txt`.

Add the path to properties and metadata files in dataset.csv

```
BasePath,DstBlobPathOrPrefix,BlobType,Disposition,MetadataFile,PropertiesFile  
H:\Video\,https://mystorageaccount.blob.core.windows.net/video/,BlockBlob,rename,None,H:\mydirectory\properties.xml  
H:\Photo\,https://mystorageaccount.blob.core.windows.net/photo/,BlockBlob,rename,None,H:\mydirectory\properties.xml  
K:\Temp\FavoriteVideo.ISO,https://mystorageaccount.blob.core.windows.net/favorite/FavoriteVideo.ISO,BlockBlob,rename,None,H:\mydirectory\properties.xml  
\myshare\john\music\,https://mystorageaccount.blob.core.windows.net/music/,BlockBlob,rename,None,H:\mydirectory\properties.xml
```

Next steps

[Import-Export Service Metadata and Properties File Format](#)

Sample workflow to prepare hard drives for an import job

1/17/2017 • 2 min to read • [Edit on GitHub](#)

This article walks you through the complete process of preparing drives for an import job.

Sample data

This example imports the following data into an Azure storage account named `mystorageaccount`:

LOCATION	DESCRIPTION	DATA SIZE
H:\Video	A collection of videos	12 TB
H:\Photo	A collection of photos	30 GB
K:\Temp\FavoriteMovie.ISO	A Blu-Ray™ disk image	25 GB
\bigshare\john\music	A collection of music files on a network share	10 GB

Storage account destinations

The import job will import the data into the following destinations in the storage account:

SOURCE	DESTINATION VIRTUAL DIRECTORY OR BLOB
H:\Video	https://mystorageaccount.blob.core.windows.net/video
H:\Photo	https://mystorageaccount.blob.core.windows.net/photo
K:\Temp\FavoriteMovie.ISO	https://mystorageaccount.blob.core.windows.net/favorite/FavoriteMovies.ISO
\bigshare\john\music	https://mystorageaccount.blob.core.windows.net/music

With this mapping, the file `H:\Video\Drama\GreatMovie.mov` will be imported to the blob

`https://mystorageaccount.blob.core.windows.net/video/Drama/GreatMovie.mov`.

Determine hard drive requirements

Next, to determine how many hard drives are needed, compute the size of the data:

`12TB + 30GB + 25GB + 10GB = 12TB + 65GB`

For this example, two 8TB hard drives should be sufficient. However, since the source directory `H:\Video` has 12TB of data and your single hard drive's capacity is only 8TB, you will be able to specify this in the following way in the `dataset.csv` file:

```
BasePath,DstBlobPathOrPrefix,BlobType,Disposition,MetadataFile,PropertiesFile
H:\Video\,https://mystorageaccount.blob.core.windows.net/video/,BlockBlob, rename, None,H:\mydirectory\properties.xml
H:\Photo\,https://mystorageaccount.blob.core.windows.net/photo/,BlockBlob, rename, None,H:\mydirectory\properties.xml
K:\Temp\FavoriteVideo.ISO,https://mystorageaccount.blob.core.windows.net/favorite/FavoriteVideo.ISO,BlockBlob, rename, None,H:\mydirectory\properties.xml
\\myshare\john\music\,https://mystorageaccount.blob.core.windows.net/music/,BlockBlob, rename, None,H:\mydirectory\properties.xml
```

Attach drives and configure the job

You will attach both disks to the machine and create volumes. Then author **driveset.csv** file:

```
DriveLetter,FormatOption,SilentOrPromptOnFormat,Encryption,ExistingBitLockerKey
X,Format,SilentMode,Encrypt,
Y,Format,SilentMode,Encrypt,
```

The tool will distribute data across two hard drives in an optimized way.

In addition, you can set the following metadata for all files:

- **UploadMethod:** Windows Azure Import/Export Service
- **DataSetName:** SampleData
- **CreationDate:** 10/1/2013

To set metadata for the imported files, create a text file, `c:\WAIImportExport\SampleMetadata.txt`, with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<Metadata>
    <UploadMethod>Windows Azure Import/Export Service</UploadMethod>
    <DataSetName>SampleData</DataSetName>
    <CreationDate>10/1/2013</CreationDate>
</Metadata>
```

You can also set some properties for the `FavoriteMovie.ISO` blob:

- **Content-Type:** application/octet-stream
- **Content-MD5:** Q2h1Y2sgSW50ZWdyaXR5IQ==
- **Cache-Control:** no-cache

To set these properties, create a text file, `c:\WAIImportExport\SampleProperties.txt`:

```
<?xml version="1.0" encoding="UTF-8"?>
<Properties>
    <Content-Type>application/octet-stream</Content-Type>
    <Content-MD5>Q2h1Y2sgSW50ZWdyaXR5IQ==</Content-MD5>
    <Cache-Control>no-cache</Cache-Control>
</Properties>
```

Run the Azure Import/Export tool (WAIImportExport.exe)

Now you are ready to run the Azure Import/Export tool to prepare the two hard drives.

For the first session:

```
WAIImportExport.exe PrepImport /j:JournalTest.jrn /id:session#1 /sk:***** /InitialDriveSet:driveset-1.csv  
/DataSet:dataset-1.csv /logdir:F:\logs
```

If any more data needs to be added, create another dataset file (same format as Initialdataset).

For the second session:

```
WAIImportExport.exe PrepImport /j:JournalTest.jrn /id:session#2 /DataSet:dataset-2.csv
```

Once the copy sessions have completed, you can disconnect the two drives from the copy computer and ship them to the appropriate Azure data center. You'll upload the two journal files, <FirstDriveSerialNumber>.xml and <SecondDriveSerialNumber>.xml, when you create the import job in the Azure portal.

Next steps

- [Preparing hard drives for an import job](#)
- [Quick reference for frequently used commands](#)

Quick reference for frequently used commands for import jobs

1/17/2017 • 1 min to read • [Edit on GitHub](#)

This article provides a quick reference for some frequently used commands. For detailed usage, see [Preparing Hard Drives for an Import Job](#).

Import job quick reference

For the first session:

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#1 /sk:***** /InitialDriveSet:driveset-1.csv  
/DataSet:dataset-1.csv /logdir:F:\logs
```

Second session:

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#2 /DataSet:dataset-2.csv
```

Abort the latest session:

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#2 /AbortSession
```

Resume the latest interrupted session:

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#3 /ResumeSession
```

Add drives to the latest session:

```
WAImportExport.exe PrepImport /j:JournalTest.jrn /id:session#3 /AdditionalDriveSet:driveset-2.csv
```

Next steps

[Sample workflow to prepare hard drives for an import job](#)

Previewing Drive Usage for an Export Job

1/17/2017 • 2 min to read • [Edit on GitHub](#)

Before you create an export job, you need to choose a set of blobs that are to be exported. The Microsoft Azure Import/Export service allows you to use a list of blob paths or blob prefixes to represent the blobs you have selected.

Next you need to determine how many drives you need to send. The Microsoft Azure Import/Export tool provides the `PreviewExport` command to preview drive usage for the blobs you selected, based on the size of the drives you are going to use. You can specify the following parameters:

COMMAND-LINE OPTION	DESCRIPTION
<code>/logdir:</code>	Optional. The log directory. Verbose log files will be written to this directory. If no log directory is specified, the current directory will be used as the log directory.
<code>/sn:</code>	Required. The name of the storage account for the export job.
<code>/sk:</code>	Required if and only if a container SAS is not specified. The account key for the storage account for the export job.
<code>/csas:</code>	Required if and only if a storage account key is not specified. The container SAS for listing the blobs to be exported in the export job.
<code>/ExportBlobListFile:</code>	Required. Path to the XML file containing list of blob paths or blob path prefixes for the blobs to be exported. The file format used in the <code>BlobListBlobPath</code> element in the Put Job operation of the Import/Export Service REST API.
<code>/DriveSize:</code>	Required. The size of drives to use for an export job, e.g., 500GB, 1.5TB.

The following example demonstrates the `PreviewExport` command:

```
WAIImportExport.exe PreviewExport /sn:bobmediaaccount  
/sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN0lp+kP0C9MEdx3GELxmBw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ==  
/ExportBlobListFile:C:\WAIImportExport\mybloblist.xml /DriveSize:500GB
```

The export blob list file may contain blob names and blob prefixes, as shown here:

```
<?xml version="1.0" encoding="utf-8"?>  
<BlobList>  
<BlobPath>pictures/animals/koala.jpg</BlobPath>  
<BlobPathPrefix>/vhds/</BlobPathPrefix>  
<BlobPathPrefix>/movies/</BlobPathPrefix>  
</BlobList>
```

The Azure Import/Export tool lists all blobs to be exported and calculates how to pack them into drives of the specified size, taking into account any necessary overhead, then estimates the number of drives needed to hold the blobs and drive usage information.

Here is an example of the output, with informational logs omitted:

```
Number of unique blob paths/prefixes: 3
Number of duplicate blob paths/prefixes: 0
Number of nonexistent blob paths/prefixes: 1

Drive size: 500.00 GB
Number of blobs that can be exported: 6
Number of blobs that cannot be exported: 2
Number of drives needed: 3
    Drive #1: blobs = 1, occupied space = 454.74 GB
    Drive #2: blobs = 3, occupied space = 441.37 GB
    Drive #3: blobs = 2, occupied space = 131.28 GB
```

See Also

[Azure Import-Export Tool Reference](#)

Reviewing Job Status with Copy Log Files

1/17/2017 • 1 min to read • [Edit on GitHub](#)

When the Microsoft Azure Import/Export service processes drives associated with an import or export job, it writes copy log files to the storage account to or from which you are importing or exporting blobs. The log file contains detailed status about each file that was imported or exported. The URL to each copy log file is returned when you query the status of a completed job; see [Get Job](#) for more information.

The following are example URLs for copy log files for an import job with two drives:

```
http://myaccount.blob.core.windows.net/ImportExportStatesPath/waies/myjob_9WM35C2V_20130921-034307-902_error.xml
```

```
http://myaccount.blob.core.windows.net/ImportExportStatesPath /waies/myjob_9WM45A6Q_20130921-042122-021_error.xml
```

See [Import-Export Service Log File Format](#) for the format of copy logs and the full list of status codes.

See Also

[Setting Up the Azure Import-Export Tool](#)

[Preparing Hard Drives for an Import Job](#)

[Repairing an Import Job](#)

[Repairing an Export Job](#)

[Troubleshooting the Azure Import-Export Tool](#)

Repairing an Import Job

1/17/2017 • 4 min to read • [Edit on GitHub](#)

The Microsoft Azure Import/Export service may fail to copy some of your files or parts of a file to the Windows Azure Blob service. Some reasons for failures include:

- Corrupted files
- Damaged drives
- The storage account key changed while the file was being transferred.

You can run the Microsoft Azure Import/Export tool with the import job's copy log files, and the tool will upload the missing files (or parts of a file) to your Windows Azure storage account to complete import job.

The command for repairing an import job is **RepairImport**. The following parameters can be specified:

/r:	Required. Path to the repair file, which tracks the progress of the repair, and allows you to resume an interrupted repair. Each drive must have one and only one repair file. When you start a repair for a given drive, you will pass in the path to a repair file which does not yet exist. To resume an interrupted repair, you should pass in the name of an existing repair file. The repair file corresponding to the target drive must always be specified.
/logdir:	Optional. The log directory. Verbose log files will be written to this directory. If no log directory is specified, the current directory will be used as the log directory.
/d:	Required. One or more semicolon-separated directories that contain the original files that were imported. The import drive may also be used, but is not required if alternate locations of original files are available.
/blk:	Optional. You should specify the BitLocker key if you want the tool to unlock an encrypted drive where the original files are available.
/sn:	Required. The name of the storage account for the import job.
/sk:	Required if and only if a container SAS is not specified. The account key for the storage account for the import job.
/csas:	Required if and only if the storage account key is not specified. The container SAS for accessing the blobs associated with the import job.

/CopyLogFile:	<p>Required. Path to the drive copy log file (either verbose log or error log). The file is generated by the Windows Azure Import/Export service and can be downloaded from the blob storage associated with the job. The copy log file contains information about failed blobs or files which are to be repaired.</p>
/PathMapFile:	<p>Optional. Path to a text file that can be used to resolve ambiguities if you have multiple files with the same name that you were importing in the same job. The first time the tool is run, it can populate this file with all of the ambiguous names. Subsequent runs of the tool will use this file to resolve the ambiguities.</p>

Using the RepairImport Command

To repair import data by streaming the data over the network, you must specify the directories that contain the original files you were importing using the `/d` parameter. You must also specify the copy log file that you downloaded from your storage account. A typical command line to repair an import job with partial failures looks like:

```
WAImportExport.exe RepairImport /r:C:\WAImportExport\9WM35C2V.rep /d:C:\Users\bob\Pictures;X:\BobBackup\photos
/sn:bobmediaaccount /sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN01p+kp0C9MEdx3GELxmBw4hK94f7KysbbeKLdksg7VoN1W/a5UuM2zNgQ==
/CopyLogFile:C:\WAImportExport\9WM35C2V.log
```

The following is an example of a copy log file. In this case, one 64K piece of a file was corrupted on the drive that was shipped for the import job. Since this is the only failure indicated, the rest of the blobs in the job were successfully imported.

```
<?xml version="1.0" encoding="utf-8"?>
<DriveLog>
  <DriveId>9WM35C2V</DriveId>
  <Blob Status="CompletedWithErrors">
    <BlobPath>pictures/animals/koala.jpg</BlobPath>
    <FilePath>\animals\koala.jpg</FilePath>
    <Length>163840</Length>
    <ImportDisposition Status="Overwritten">overwrite</ImportDisposition>
    <PageRangeList>
      <PageRange Offset="65536" Length="65536" Hash="AA2585F6F6FD01C4AD4256E018240CD4" Status="Corrupted" />
    </PageRangeList>
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>
```

When this copy log is passed to the Azure Import/Export tool, the tool will try to finish the import for this file by copying the missing contents across the network. Following the example above, the tool will look for the original file `\animals\koala.jpg` within the two directories `C:\Users\bob\Pictures` and `X:\BobBackup\photos`. If the file `C:\Users\bob\Pictures\animals\koala.jpg` exists, the Azure Import/Export tool will copy the missing range of data to the corresponding blob `http://bobmediaaccount.blob.core.windows.net/pictures/animals/koala.jpg`.

Resolving Conflicts When Using RepairImport

In some situations, the tool may not be able to find or open the necessary file for one of the following reasons: the file could not be found, the file is not accessible, the file name is ambiguous, or the content of the file is no longer correct.

An ambiguous error could occur if the tool is trying to locate `\animals\koala.jpg` and there is a file with that name

under both `C:\Users\bob\pictures` and `X:\BobBackup\photos`. That is, both `C:\Users\bob\pictures\animals\koala.jpg` and `X:\BobBackup\photos\animals\koala.jpg` exist on the import job drives.

The `/PathMapFile` option will allow you to resolve these errors. You can specify the name of the file which will contains the list of files that the tool was not able to correctly identify. The following is an example command line that would populate `9WM35C2V_pathmap.txt`:

```
WAIImportExport.exe RepairImport /r:C:\WAIImportExport\9WM35C2V.rep /d:C:\Users\bob\Pictures;X:\BobBackup\photos /sn:bobmediaaccount /sk:VKGbrUqBWLYJ6zg1m29VOTrxpBgdN0lp+kp0C9MEdx3GELxmBw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ== /CopyLogFile:C:\WAIImportExport\9WM35C2V.log /PathMapFile:C:\WAIImportExport\9WM35C2V_pathmap.txt
```

The tool will then write the problematic file paths to `9WM35C2V_pathmap.txt`, one on each line. For instance, the file may contain the following entries after running the command:

```
\animals\koala.jpg  
\animals\kangaroo.jpg
```

For each file in the list, you should attempt to locate and open the file to ensure it is available to the tool. If you wish to tell the tool explicitly where to find a file, you can modify the path map file and add the path to each file on the same line, separated by a tab character:

```
\animals\koala.jpg      C:\Users\bob\Pictures\animals\koala.jpg  
\animals\kangaroo.jpg   X:\BobBackup\photos\animals\kangaroo.jpg
```

After making the necessary files available to the tool, or updating the path map file, you can rerun the tool to complete the import process.

See Also

- [Setting Up the Azure Import-Export Tool](#)
- [Preparing Hard Drives for an Import Job](#)
- [Reviewing Job Status with Copy Log Files](#)
- [Repairing an Export Job](#)
- [Troubleshooting the Azure Import-Export Tool](#)

Repairing an Export Job

1/17/2017 • 4 min to read • [Edit on GitHub](#)

After an export job has completed, you can run the Microsoft Azure Import/Export tool on premise to:

1. Download any files that the Azure Import/Export service was unable to export.
2. Validate that the files on the drive were correctly exported.

You must have connectivity to Azure Storage to use this functionality.

The command for repairing an import job is **RepairExport**. The following parameters can be specified:

PARAMETER	DESCRIPTION
/r:	Required. Path to the repair file, which tracks the progress of the repair, and allows you to resume an interrupted repair. Each drive must have one and only one repair file. When you start a repair for a given drive, you will pass in the path to a repair file which does not yet exist. To resume an interrupted repair, you should pass in the name of an existing repair file. The repair file corresponding to the target drive must always be specified.
/logdir:	Optional. The log directory. Verbose log files will be written to this directory. If no log directory is specified, the current directory will be used as the log directory.
/d:	Required. The directory to validate and repair. This is usually the root directory of the export drive, but could also be a network file share containing a copy of the exported files.
/blk:	Optional. You should specify the BitLocker key if you want the tool to unlock an encrypted where the exported files are stored.
/sn:	Required. The name of the storage account for the export job.
/sk:	Required if and only if a container SAS is not specified. The account key for the storage account for the export job.
/csas:	Required if and only if the storage account key is not specified. The container SAS for accessing the blobs associated with the export job.
/CopyLogFile:	Required. The path to the drive copy log file. The file is generated by the Windows Azure Import/Export service and can be downloaded from the blob storage associated with the job. The copy log file contains information about failed blobs or files which are to be repaired.

PARAMETER	DESCRIPTION
/ManifestFile:	<p>Optional. The path to the export drive's manifest file. This file is generated by the Windows Azure Import/Export service and stored on the export drive, and optionally in a blob in the storage account associated with the job.</p> <p>The content of the files on the export drive will be verified with the MD5 hashes contained in this file. Any files that are determined to be corrupted will be downloaded and rewritten to the target directories.</p>

Using RepairExport Mode to Correct Failed Exports

You can use the Azure Import/Export tool to download files that failed to export. The copy log file will contain a list of files that failed to export.

The causes of export failures include the following possibilities:

- Damaged drives
- The storage account key changed during the transfer process

To run the tool in **RepairExport** mode, you first need to connect the drive containing the exported files to your computer. Next, run the Azure Import/Export tool, specifying the path to that drive with the **/d** parameter. You also need to specify the path to the drive's copy log file that you downloaded. The following command line example below runs the tool to repair any files that failed to export:

```
WAIImportExport.exe RepairExport /r:C:\WAIImportExport\9WM35C3U.rep /d:G:\ /sn:bobmediaaccount
/sk:VkGbrUqBWLYJ6zg1m29VOTrxpBgdN0lp+kp0C9MEdx3GEIxmbw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ==
/CopyLogFile:C:\WAIImportExport\9WM35C3U.log
```

The following is an example of a copy log file that shows that one block in the blob failed to export:

```
<?xml version="1.0" encoding="utf-8"?>
<DriveLog>
  <DriveId>9WM35C2V</DriveId>
  <Blob Status="CompletedWithErrors">
    <BlobPath>pictures/wild/desert.jpg</BlobPath>
    <FilePath>\pictures\wild\desert.jpg</FilePath>
    <LastModified>2012-09-18T23:47:08Z</LastModified>
    <Length>163840</Length>
    <BlockList>
      <Block Offset="65536" Length="65536" Id="AQAAAA==" Status="Failed" />
    </BlockList>
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>
```

The copy log file indicates that a failure occurred while the Windows Azure Import/Export service was downloading one of the blob's blocks to the file on the export drive. The other components of the file downloaded successfully, and the file length was correctly set. In this case, the tool will open the file on the drive, download the block from the storage account, and write it to the file range starting from offset 65536 with length 65536.

Using RepairExport to Validate Drive Contents

You can also use Azure Import/Export with the **RepairExport** option to validate the contents on the drive are correct. The manifest file on each export drive contains MD5s for the contents of the drive.

The Azure Import/Export service can also save the manifest files to a storage account during the export process. The location of the manifest files is available via the [Get Job](#) operation when the job has completed. See [Import-Export Service Manifest File Format](#) for more information about the format of a drive manifest file.

The following example shows how to run the Azure Import/Export tool with the **/ManifestFile** and **/CopyLogFile** parameters:

```
WAIImportExport.exe RepairExport /r:C:\WAIImportExport\9WM35C3U.rep /d:G:\ /sn:bobmediaaccount  
/sk:VKGbrUqBWLYJ6zg1m29VOTrxpBgdN0lp+kP0C9MEdx3GELxmBw4hK94f7KysbbeKLdksg7VoN1W/a5UuM2zNgQ==  
/CopyLogFile:C:\WAIImportExport\9WM35C3U.log /ManifestFile:G:\9WM35C3U.manifest
```

The following is an example of a manifest file:

```
<?xml version="1.0" encoding="utf-8"?>  
<DriveManifest Version="2011-10-01">  
  <Drive>  
    <DriveId>9WM35C3U</DriveId>  
    <ClientCreator>Windows Azure Import/Export service</ClientCreator>  
    <BlobList>  
      <Blob>  
        <BlobPath>pictures/city/redmond.jpg</BlobPath>  
        <FilePath>\pictures\city\redmond.jpg</FilePath>  
        <Length>15360</Length>  
        <PageRangeList>  
          <PageRange Offset="0" Length="3584" Hash="72FC55ED9AFDD40A0C8D5C4193208416" />  
          <PageRange Offset="3584" Length="3584" Hash="68B28A561B73D1DA769D4C24AA427DB8" />  
          <PageRange Offset="7168" Length="512" Hash="F521DF2F50C46BC5F9EA9FB787A23EED" />  
        </PageRangeList>  
        <PropertiesPath  
          Hash="E72A22EA959566066AD89E3B49020C0A">\pictures\city\redmond.jpg.properties</PropertiesPath>  
      </Blob>  
      <Blob>  
        <BlobPath>pictures/wild/canyon.jpg</BlobPath>  
        <FilePath>\pictures\wild\canyon.jpg</FilePath>  
        <Length>10884</Length>  
        <BlockList>  
          <Block Offset="0" Length="2721" Id="AAAAAA==" Hash="263DC9C4B99C2177769C5EBE04787037" />  
          <Block Offset="2721" Length="2721" Id="AQAAAA==" Hash="0C52BAE2CC20EFEC15CC1E3045517AA6" />  
          <Block Offset="5442" Length="2721" Id="AgAAAA==" Hash="73D1CB62CB426230C34C9F57B7148F10" />  
          <Block Offset="8163" Length="2721" Id="AwAAAA==" Hash="11210E665C5F8E7E4F136D053B243E6A" />  
        </BlockList>  
        <PropertiesPath  
          Hash="81D7F81B2C29F10D6E123D386C3A4D5A">\pictures\wild\canyon.jpg.properties</PropertiesPath>  
      </Blob>  
    </BlobList>  
  </Drive>  
</DriveManifest>
```

After finishing the repair process, the tool will read through each file referenced in the manifest file and verify the file's integrity with the MD5 hashes. For the manifest above, it will go through the following components.

G:\pictures\city\redmond.jpg, offset 0, length 3584

G:\pictures\city\redmond.jpg, offset 3584, length 3584

G:\pictures\city\redmond.jpg, offset 7168, length 3584

G:\pictures\city\redmond.jpg.properties

G:\pictures\wild\canyon.jpg, offset 0, length 2721

G:\pictures\wild\canyon.jpg, offset 2721, length 2721

G:\pictures\wild\canyon.jpg, offset 5442, length 2721

G:\pictures\wild\canyon.jpg, offset 8163, length 2721

G:\pictures\wild\canyon.jpg.properties

Any component failing the verification will be downloaded by the tool and rewritten to the same file on the drive.

See Also

[Setting Up the Azure Import-Export Tool](#)

[Preparing Hard Drives for an Import Job](#)

[Reviewing Job Status with Copy Log Files](#)

[Repairing an Import Job](#)

[Troubleshooting the Azure Import-Export Tool](#)

Troubleshooting the Azure Import-Export Tool

1/17/2017 • 1 min to read • [Edit on GitHub](#)

The Microsoft Azure Import/Export tool returns error messages if it runs into issues. This topic lists some common issues that users may run into.

A copy session fails, what I should do?

When a copy session fails, there are two options:

If the error is retryable, for example if the network share was offline for a short period and now is back online, you can resume the copy session. If the error is not retryable, for example if you specified the wrong source file directory in the command line parameters, you need to abort the copy session. See [Preparing Hard Drives for an Import Job](#) for more information about resuming and aborting copy sessions.

I can't resume or abort a copy session.

If the copy session is the first copy session for a drive, then the error message should state: "The first copy session cannot be resumed or aborted." In this case, you can delete the old journal file and rerun the command.

If a copy session is not the first one for a drive, it can always be resumed or aborted.

I lost the journal file, can I still create the job?

The journal file for a drive contains the complete information of copying data to this drive, and it is needed to add more files to the drive and will be used to create an import job. If the journal file is lost, you will have to redo all the copy sessions for the drive.

See Also

[Setting Up the Azure Import-Export Tool](#)

[Preparing Hard Drives for an Import Job](#)

[Reviewing Job Status with Copy Log Files](#)

[Repairing an Import Job](#)

[Repairing an Export Job](#)

Import-Export Service Manifest File Format

1/17/2017 • 7 min to read • [Edit on GitHub](#)

The drive manifest file describes the mapping between blobs in Azure Blob storage and files on drive comprising an import or export job. For an import operation, the manifest file is created as a part of the drive preparation process, and is stored on the drive before the drive is sent to the Azure data center. During an export operation, the manifest is created and stored on the drive by the Azure Import/Export service.

For both import and export jobs, the drive manifest file is stored on the import or export drive; it is not transmitted to the service via any API operation.

The following describes the general format of a drive manifest file:

```

<?xml version="1.0" encoding="UTF-8"?>
<DriveManifest Version="2014-11-01">
  <Drive>
    <DriveId>drive-id</DriveId>
    import-export-credential

    <!-- First Blob List -->
    <BlobList>
      <!-- Global properties and metadata that applies to all blobs -->
      [<MetadataPath Hash="md5-hash">global-metadata-file-path</MetadataPath>]
      [<PropertiesPath
        Hash="md5-hash">global-properties-file-path</PropertiesPath>]

      <!-- First Blob -->
      <Blob>
        <BlobPath>blob-path-relative-to-account</BlobPath>
        <FilePath>file-path-relative-to-transfer-disk</FilePath>
        [<ClientData>client-data</ClientData>]
        [<Snapshot>snapshot</Snapshot>]
        <Length>content-length</Length>
        [<ImportDisposition>import-disposition</ImportDisposition>]
        page-range-list-or-block-list
        [<MetadataPath Hash="md5-hash">metadata-file-path</MetadataPath>]
        [<PropertiesPath Hash="md5-hash">properties-file-path</PropertiesPath>]
      </Blob>

      <!-- Second Blob -->
      <Blob>
        . . .
      </Blob>
    </BlobList>

    <!-- Second Blob List -->
    <BlobList>
      . . .
    </BlobList>
  </Drive>
</DriveManifest>

import-export-credential ::=

<StorageAccountKey>storage-account-key</StorageAccountKey> | <ContainerSas>container-sas</ContainerSas>

page-range-list-or-block-list ::=
  page-range-list | block-list

page-range-list ::=
  <PageRangeList>
    [<PageRange Offset="page-range-offset" Length="page-range-length"
      Hash="md5-hash"/>]
    [<PageRange Offset="page-range-offset" Length="page-range-length"
      Hash="md5-hash"/>]
  </PageRangeList>

block-list ::=
  <BlockList>
    [<Block Offset="block-offset" Length="block-length" [Id="block-id"]
      Hash="md5-hash"/>]
    [<Block Offset="block-offset" Length="block-length" [Id="block-id"]
      Hash="md5-hash"/>]
  </BlockList>

```

The data elements and attributes of the drive manifest XML format are specified in the following table.

XML ELEMENT	TYPE	DESCRIPTION
<code>DriveManifest</code>	Root element	The root element of the manifest file. All other elements in the file are beneath this element.
<code>version</code>	Attribute, String	The version of the manifest file.
<code>Drive</code>	Nested XML element	Contains the manifest for each drive.
<code>DriveId</code>	String	The unique drive identifier for the drive. The drive identifier is found by querying the drive for its serial number. The drive serial number is usually printed on the outside of the drive as well. The <code>DriveID</code> element must appear before any <code>BlobList</code> element in the manifest file.
<code>StorageAccountKey</code>	String	Required for import jobs if and only if <code>ContainerSas</code> is not specified. The account key for the Azure storage account associated with the job. This element is omitted from the manifest for an export operation.
<code>ContainerSas</code>	String	Required for import jobs if and only if <code>StorageAccountKey</code> is not specified. The container SAS for accessing the blobs associated with the job. See Put Job for its format. This element is omitted from the manifest for an export operation.
<code>ClientCreator</code>	String	Specifies the client which created the XML file. This value is not interpreted by the Import/Export service.
<code>BlobList</code>	Nested XML element	Contains a list of blobs that are part of the import or export job. Each blob in a blob list shares the same metadata and properties.
<code>BlobList/MetadataPath</code>	String	Optional. Specifies the relative path of a file on the disk that contains the default metadata that will be set on blobs in the blob list for an import operation. This metadata can be optionally overridden on a blob-by-blob basis. This element is omitted from the manifest for an export operation.
<code>BlobList/MetadataPath/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash value for the metadata file.

XML ELEMENT	TYPE	DESCRIPTION
<code>BlobList/PropertiesPath</code>	String	<p>Optional. Specifies the relative path of a file on the disk that contains the default properties that will be set on blobs in the blob list for an import operation. These properties can be optionally overridden on a blob-by-blob basis.</p> <p>This element is omitted from the manifest for an export operation.</p>
<code>BlobList/PropertiesPath/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash value for the properties file.
<code>Blob</code>	Nested XML element	Contains information about each blob in each blob list.
<code>Blob/BlobPath</code>	String	The relative URI to the blob, beginning with the container name. If the blob is in root container, it must begin with <code>\$root</code> .
<code>Blob/FilePath</code>	String	<p>Specifies the relative path to the file on the drive. For export jobs, the blob path will be used for the file path if possible; e.g., <code>pictures/bob/wild/desert.jpg</code> will be exported to <code>\pictures\bob\wild\desert.jpg</code>. However, due to the limitations of NTFS names, a blob may be exported to a file with a path that doesn't resemble the blob path.</p>
<code>Blob/ClientData</code>	String	Optional. Contains comments from the customer. This value is not interpreted by the Import/Export service.
<code>Blob/Snapshot</code>	DateTime	Optional for export jobs. Specifies the snapshot identifier for an exported blob snapshot.
<code>Blob/Length</code>	Integer	Specifies the total length of the blob in bytes. The value may be up to 200 GB for a block blob and up to 1 TB for a page blob. For a page blob, this value must be a multiple of 512.

XML ELEMENT	TYPE	DESCRIPTION
<code>Blob/ImportDisposition</code>	String	<p>Optional for import jobs, omitted for export jobs. This specifies how the Import/Export service should handle the case for an import job where a blob with the same name already exists. If this value is omitted from the import manifest, the default value is <code>rename</code>.</p> <p>The values for this element include:</p> <ul style="list-style-type: none"> - <code>no-overwrite</code>: If a destination blob is already present with the same name, the import operation will skip importing this file. - <code>overwrite</code>: Any existing destination blob is overwritten completely by the newly imported file. - <code>rename</code>: The new blob will be uploaded with a modified name. <p>The renaming rule is as follows:</p> <ul style="list-style-type: none"> - If the blob name doesn't contain a dot, a new name is generated by appending <code>(2)</code> to the original blob name; if this new name also conflicts with an existing blob name, then <code>(3)</code> is appended in place of <code>(2)</code>; and so on. - If the blob name contains a dot, the portion following the last dot is considered the extension name. Similar to the above procedure, <code>(2)</code> is inserted before the last dot to generate a new name; if the new name still conflicts with an existing blob name, then the service tries <code>(3)</code>, <code>(4)</code>, and so on, until a non-conflicting name is found. <p>Some examples:</p> <p>The blob <code>BlobNameWithoutDot</code> will be renamed to:</p> <pre><code>BlobNameWithoutDot (2) // if BlobNameWithoutDot exists</code></pre> <pre><code>BlobNameWithoutDot (3) // if both BlobNameWithoutDot and BlobNameWithoutDot (2) exist</code></pre> <p>The blob <code>Seattle.jpg</code> will be renamed to:</p> <pre><code>Seattle (2).jpg // if Seattle.jpg exists</code></pre> <pre><code>Seattle (3).jpg // if both Seattle.jpg and Seattle (2).jpg exist</code></pre>

XML ELEMENT	TYPE	DESCRIPTION
<code>PageRangeList</code>	Nested XML element	<p>Required for a page blob.</p> <p>For an import operation, specifies a list of byte ranges of a file to be imported. Each page range is described by an offset and length in the source file that describes the page range, together with an MD5 hash of the region. The <code>Hash</code> attribute of a page range is required. The service will validate that the hash of the data in the blob matches the computed MD5 hash from the page range. Any number of page ranges may be used to describe a file for an import, with the total size up to 1 TB. All page ranges must be ordered by offset and no overlaps are allowed.</p> <p>For an export operation, specifies a set of byte ranges of a blob that have been exported to the drive.</p> <p>The page ranges together may cover only sub-ranges of a blob or file. The remaining part of the file not covered by any page range is expected and its content can be undefined.</p>
<code>PageRange</code>	XML element	Represents a page range.
<code>PageRange/@Offset</code>	Attribute, Integer	Specifies the offset in the transfer file and the blob where the specified page range begins. This value must be a multiple of 512.
<code>PageRange/@Length</code>	Attribute, Integer	Specifies the length of the page range. This value must be a multiple of 512 and no more than 4 MB.
<code>PageRange/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash value for the page range.

XML ELEMENT	TYPE	DESCRIPTION
<code>blockList</code>	Nested XML element	<p>Required for a block blob with named blocks.</p> <p>For an import operation, the block list specifies a set of blocks that will be imported into Azure Storage. For an export operation, the block list specifies where each block has been stored in the file on the export disk. Each block is described by an offset in the file and a block length; each block is furthermore named by a block ID attribute, and contains an MD5 hash for the block. Up to 50,000 blocks may be used to describe a blob. All blocks must be ordered by offset, and together should cover the complete range of the file, <i>i.e.</i>, there must be no gap between blocks. If the blob is no more than 64 MB, the block IDs for each block must be either all absent or all present. Block IDs are required to be Base64-encoded strings. See Put Block for further requirements for block IDs.</p>
<code>block</code>	XML element	Represents a block.
<code>block/@Offset</code>	Attribute, Integer	Specifies the offset where the specified block begins.
<code>block/@Length</code>	Attribute, Integer	Specifies the number of bytes in the block; this value must be no more than 4MB.
<code>block/@Id</code>	Attribute, String	Specifies a string representing the block ID for the block.
<code>block/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash of the block.
<code>blob/MetadataPath</code>	String	Optional. Specifies the relative path of a metadata file. During an import, the metadata is set on the destination blob. During an export operation, the blob's metadata is stored in the metadata file on the drive.
<code>blob/MetadataPath/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash of the blob's metadata file.
<code>blob/PropertiesPath</code>	String	Optional. Specifies the relative path of a properties file. During an import, the properties are set on the destination blob. During an export operation, the blob properties are stored in the properties file on the drive.

XML ELEMENT	TYPE	DESCRIPTION
<code>Blob/PropertiesPath/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash of the blob's properties file.

See Also

[Storage Import/Export REST](#)

Import-Export Service Metadata and Properties File Format

1/17/2017 • 1 min to read • [Edit on GitHub](#)

You can specify metadata and properties for one or more blobs as part of an import job or an export job. To set metadata or properties for blobs being created as part of an import job, you provide a metadata or properties file on the hard drive containing the data to be imported. For an export job, metadata and properties are written to a metadata or properties file that is included on the hard drive returned to you.

Metadata File Format

The format of a metadata file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Metadata>
[<metadata-name-1>metadata-value-1</metadata-name-1>]
[<metadata-name-2>metadata-value-2</metadata-name-2>]
...
</Metadata>
```

XML ELEMENT	TYPE	DESCRIPTION
<code>Metadata</code>	Root element	The root element of the metadata file.
<code><metadata-name></code>	String	Optional. The XML element specifies the name of the metadata for the blob, and its value specifies the value of the metadata setting.

Properties File Format

The format of a properties file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Properties>
[<Last-Modified>date-time-value</Last-Modified>]
[<Etag>etag</Etag>]
[<Content-Length>size-in-bytes</Content-Length>]
[<Content-Type>content-type</Content-Type>]
[<Content-MD5>content-md5</Content-MD5>]
[<Content-Encoding>content-encoding</Content-Encoding>]
[<Content-Language>content-language</Content-Language>]
[<Cache-Control>cache-control</Cache-Control>]
</Properties>
```

XML ELEMENT	TYPE	DESCRIPTION
<code>Properties</code>	Root element	The root element of the properties file.
<code><Last-Modified></code>	String	Optional. The last-modified time for the blob. For export jobs only.

XML ELEMENT	TYPE	DESCRIPTION
Etag	String	Optional. The blob's ETag value. For export jobs only.
Content-Length	String	Optional. The size of the blob in bytes. For export jobs only.
Content-Type	String	Optional. The content type of the blob.
Content-MD5	String	Optional. The blob's MD5 hash.
Content-Encoding	String	Optional. The blob's content encoding.
Content-Language	String	Optional. The blob's content language.
Cache-Control	String	Optional. The cache control string for the blob.

See [Set Blob Properties](#), [Set Blob Metadata](#), and [Setting and Retrieving Properties and Metadata for Blob Resources](#) for detailed rules about setting blob metadata and properties.

Import-Export Service Log File Format

1/17/2017 • 10 min to read • [Edit on GitHub](#)

When the Microsoft Azure Import/Export service performs an action on a drive as part of an import job or an export job, logs are written to block blobs in the storage account associated with that job.

There are two logs that may be written by the Import/Export service:

- The error log is always generated in the event of an error.
- The verbose log is not enabled by default, but may be enabled by setting the `EnableVerboseLog` property on a [Put Job or Update Job Properties](#) operation.

Log File Location

The logs are written to block blobs in the container or virtual directory specified by the `ImportExportStatesPath` setting, which you can set on a [Put Job](#) operation. The location to which the logs are written depends on how authentication is specified for the job, together with the value specified for `ImportExportStatesPath`. Authentication for the job may be specified via a storage account key, or a container SAS (shared access signature).

The name of the container or virtual directory may either be the default name of `waimportexport`, or another container or virtual directory name that you specify.

The table below shows the possible options:

AUTHENTICATION METHOD	VALUE OF <code>IMPORTEXPORTSTATESPATH</code> ELEMENT	LOCATION OF LOG BLOBS
Storage account key	Default value	A container named <code>waimportexport</code> , which is the default container. For example: <code>https://myaccount.blob.core.windows.net/waimportexport</code>
Storage account key	User-specified value	A container named by the user. For example: <code>https://myaccount.blob.core.windows.net/mylogcontainer</code>
Container SAS	Default value	A virtual directory named <code>waimportexport</code> , which is the default name, beneath the container specified in the SAS. For example, if the SAS specified for the job is <code>https://myaccount.blob.core.windows.net/mylogcontainer?sv=2012-02-12&se=2015-05-22T06%3A54%3A55Z&sr=c&sp=w&sig=sigvalue</code> , then the log location would be <code>https://myaccount.blob.core.windows.net/mylogcontainer/waimportexport</code>
Container SAS	User-specified value	A virtual directory named by the user, beneath the container specified in the SAS. For example, if the SAS specified for the job is <code>https://myaccount.blob.core.windows.net/mylogcontainer?sv=2012-02-12&se=2015-05-22T06%3A54%3A55Z&sr=c&sp=w&sig=sigvalue</code> , and the specified virtual directory is named <code>mylogblobs</code> , then the log location would be <code>https://myaccount.blob.core.windows.net/mylogcontainer/mylogblobs</code> .

You can retrieve the URL for the error and verbose logs by calling the [Get Job](#) operation. The logs are available after processing of the drive is complete.

Log File Format

The format for both logs is the same: a blob containing XML descriptions of the events that occurred while copying blobs between the hard drive and the customer's account.

The verbose log contains complete information about the status of the copy operation for every blob (for an import job) or file (for an export job), whereas the error log contains only the information for blobs or files that encountered errors during the import or export job.

The verbose log format is shown below. The error log has the same structure, but filters out successful operations.

```

<DriveLog Version="2014-11-01">
  <DriveId>drive-id</DriveId>
  [<Blob Status="blob-status">
    <BlobPath>blob-path</BlobPath>
    <FilePath>file-path</FilePath>
    [<Snapshot>snapshot</Snapshot>]
    <Length>length</Length>
    [<LastModified>last-modified</LastModified>]
    [<ImportDisposition Status="import-disposition-status">import-disposition</ImportDisposition>]
    [page-range-list-or-block-list]
    [metadata-status]
    [properties-status]
  </Blob>]
  [<Blob>
    ...
  </Blob>]
  <Status>drive-status</Status>
</DriveLog>

page-range-list-or-block-list ::= page-range-list | block-list

page-range-list ::= <PageRangeList>
  [<PageRange Offset="page-range-offset" Length="page-range-length"
    [Hash="md5-hash"] Status="page-range-status"/>]
  [<PageRange Offset="page-range-offset" Length="page-range-length"
    [Hash="md5-hash"] Status="page-range-status"/>]
</PageRangeList>

block-list ::= <BlockList>
  [<Block Offset="block-offset" Length="block-length" [Id="block-id"]
    [Hash="md5-hash"] Status="block-status"/>]
  [<Block Offset="block-offset" Length="block-length" [Id="block-id"]
    [Hash="md5-hash"] Status="block-status"/>]
</BlockList>

metadata-status ::= <Metadata Status="metadata-status">
  [<GlobalPath Hash="md5-hash">global-metadata-file-path</GlobalPath>]
  [<Path Hash="md5-hash">metadata-file-path</Path>]
</Metadata>

properties-status ::= <Properties Status="properties-status">
  [<GlobalPath Hash="md5-hash">global-properties-file-path</GlobalPath>]
  [<Path Hash="md5-hash">properties-file-path</Path>]
</Properties>

```

The following table describes the elements of the log file.

XML ELEMENT	TYPE	DESCRIPTION
[DriveLog]	XML Element	Represents a drive log.
[Version]	Attribute, String	The version of the log format.
[DriveId]	String	The drive's hardware serial number.
[Status]	String	Status of the drive processing. See the [Drive Status Codes] table below for more information.
[Blob]	Nested XML element	Represents a blob.
[Blob/BlobPath]	String	The URI of the blob.
[Blob/FilePath]	String	The relative path to the file on the drive.
[Blob/Snapshot]	DateTime	The snapshot version of the blob, for an export job only.
[Blob/Length]	Integer	The total length of the blob in bytes.
[Blob/LastModified]	DateTime	The date/time that the blob was last modified, for an export job only.

XML ELEMENT	TYPE	DESCRIPTION
<code>Blob/ImportDisposition</code>	String	The import disposition of the blob, for an import job only.
<code>Blob/ImportDisposition/@Status</code>	Attribute, String	The status of the import disposition.
<code>PageRangeList</code>	Nested XML element	Represents a list of page ranges for a page blob.
<code>PageRange</code>	XML element	Represents a page range.
<code>PageRange/@Offset</code>	Attribute, Integer	Starting offset of the page range in the blob.
<code>PageRange/@Length</code>	Attribute, Integer	Length in bytes of the page range.
<code>PageRange/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the page range.
<code>PageRange/@Status</code>	Attribute, String	Status of processing the page range.
<code>BlockList</code>	Nested XML element	Represents a list of blocks for a block blob.
<code>Block</code>	XML element	Represents a block.
<code>Block/@Offset</code>	Attribute, Integer	Starting offset of the block in the blob.
<code>Block/@Length</code>	Attribute, Integer	Length in bytes of the block.
<code>Block/@Id</code>	Attribute, String	The block ID.
<code>Block/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the block.
<code>Block/@Status</code>	Attribute, String	Status of processing the block.
<code>Metadata</code>	Nested XML element	Represents the blob's metadata.
<code>Metadata/@Status</code>	Attribute, String	Status of processing of the blob metadata.
<code>Metadata/GlobalPath</code>	String	Relative path to the global metadata file.
<code>Metadata/GlobalPath/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the global metadata file.
<code>Metadata/Path</code>	String	Relative path to the metadata file.
<code>Metadata/Path/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the metadata file.
<code>Properties</code>	Nested XML element	Represents the blob properties.
<code>Properties/@Status</code>	Attribute, String	Status of processing the blob properties, e.g. file not found, completed.
<code>Properties/GlobalPath</code>	String	Relative path to the global properties file.
<code>Properties/GlobalPath/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the global properties file.
<code>Properties/Path</code>	String	Relative path to the properties file.
<code>Properties/Path/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the properties file.
<code>Blob/Status</code>	String	Status of processing the blob.

Drive Status Codes

The following table lists the status codes for processing a drive.

STATUS CODE	DESCRIPTION
<code>Completed</code>	The drive has finished processing without any errors.
<code>CompletedWithWarnings</code>	The drive has finished processing with warnings in one or more blobs per the import dispositions specified for the blobs.
<code>CompletedWithErrors</code>	The drive has finished with errors in one or more blobs or chunks.
<code>DiskNotFound</code>	No disk is found on the drive.
<code>VolumeNotNtfs</code>	The first data volume on the disk is not in NTFS format.
<code>DiskOperationFailed</code>	An unknown failure occurred when performing operations on the drive.
<code>BitLockerVolumeNotFound</code>	No BitLocker encryptable volume is found.
<code>BitLockerNotActivated</code>	BitLocker is not enabled on the volume.
<code>BitLockerProtectorNotFound</code>	The numerical password key protector does not exist on the volume.
<code>BitLockerKeyInvalid</code>	The numerical password provided cannot unlock the volume.
<code>BitLockerUnlockVolumeFailed</code>	Unknown failure has happened when trying to unlock the volume.
<code>BitLockerFailed</code>	An unknown failure occurred while performing BitLocker operations.
<code>ManifestNameInvalid</code>	The manifest file name is invalid.
<code>ManifestNameTooLong</code>	The manifest file name is too long.
<code>ManifestNotFound</code>	The manifest file is not found.
<code>ManifestAccessDenied</code>	Access to the manifest file is denied.
<code>ManifestCorrupted</code>	The manifest file is corrupted (the content does not match its hash).
<code>ManifestFormatInvalid</code>	The manifest content does not conform to the required format.
<code>ManifestDriveIdMismatch</code>	The drive ID in the manifest file does not match the one read from the drive.
<code>ReadManifestFailed</code>	A disk I/O failure occurred while reading from the manifest.
<code>BlobListFormatInvalid</code>	The export blob list blob does not conform to the required format.
<code>BlobRequestForbidden</code>	Access to the blobs in the storage account is forbidden. This might be due to invalid storage account key or container SAS.
<code>InternalError</code>	An internal error occurred while processing the drive.

Blob Status Codes

The following table lists the status codes for processing a blob.

STATUS CODE	DESCRIPTION
<code>Completed</code>	The blob has finished processing without errors.
<code>CompletedWithErrors</code>	The blob has finished processing with errors in one or more page ranges or blocks, metadata, or properties.
<code>FileNameInvalid</code>	The file name is invalid.
<code>FileNameTooLong</code>	The file name is too long.
<code>FileNotFound</code>	The file is not found.
<code>FileAccessDenied</code>	Access to the file is denied.

STATUS CODE	DESCRIPTION
<code>BlobRequestFailed</code>	The Blob service request to access the blob has failed.
<code>BlobRequestForbidden</code>	The Blob service request to access the blob is forbidden. This might be due to invalid storage account key or container SAS.
<code>RenameFailed</code>	Failed to rename the blob (for an import job) or the file (for an export job).
<code>BlobUnexpectedChange</code>	An unexpected change has occurred with the blob (for an export job).
<code>LeasePresent</code>	There is a lease present on the blob.
<code>IOFailed</code>	A disk or network I/O failure occurred while processing the blob.
<code>Failed</code>	An unknown failure occurred while processing the blob.

Import Disposition Status Codes

The following table lists the status codes for resolving an import disposition.

STATUS CODE	DESCRIPTION
<code>Created</code>	The blob has been created.
<code>Renamed</code>	The blob has been renamed per rename import disposition. The <code>Blob/BlobPath</code> element contains the URI for the renamed blob.
<code>Skipped</code>	The blob has been skipped per <code>no-overwrite</code> import disposition.
<code>Overwritten</code>	The blob has overwritten an existing blob per <code>overwrite</code> import disposition.
<code>Cancelled</code>	A prior failure has stopped further processing of the import disposition.

Page Range/Block Status Codes

The following table lists the status codes for processing a page range or a block.

STATUS CODE	DESCRIPTION
<code>Completed</code>	The page range or block has finished processing without any errors.
<code>Committed</code>	The block has been committed, but not in the full block list because other blocks have failed, or put full block list itself has failed.
<code>Uncommitted</code>	The block is uploaded but not committed.
<code>Corrupted</code>	The page range or block is corrupted (the content does not match its hash).
<code>FileUnexpectedEnd</code>	An unexpected end of file has been encountered.
<code>BlobUnexpectedEnd</code>	An unexpected end of blob has been encountered.
<code>BlobRequestFailed</code>	The Blob service request to access the page range or block has failed.
<code>IOFailed</code>	A disk or network I/O failure occurred while processing the page range or block.
<code>Failed</code>	An unknown failure occurred while processing the page range or block.
<code>Cancelled</code>	A prior failure has stopped further processing of the page range or block.

Metadata Status Codes

The following table lists the status codes for processing blob metadata.

STATUS CODE	DESCRIPTION
<code>Completed</code>	The metadata has finished processing without errors.
<code>FileNameInvalid</code>	The metadata file name is invalid.

STATUS CODE	DESCRIPTION
<code>FileNameTooLong</code>	The metadata file name is too long.
<code>FileNotFoundException</code>	The metadata file is not found.
<code>FileAccessDenied</code>	Access to the metadata file is denied.
<code>Corrupted</code>	The metadata file is corrupted (the content does not match its hash).
<code>XmlReadFailed</code>	The metadata content does not conform to the required format.
<code>XmlWriteFailed</code>	Writing the metadata XML has failed.
<code>BlobRequestFailed</code>	The Blob service request to access the metadata has failed.
<code>IOFailed</code>	A disk or network I/O failure occurred while processing the metadata.
<code>Failed</code>	An unknown failure occurred while processing the metadata.
<code>Cancelled</code>	A prior failure has stopped further processing of the metadata.

Properties Status Codes

The following table lists the status codes for processing blob properties.

STATUS CODE	DESCRIPTION
<code>Completed</code>	The properties have finished processing without any errors.
<code>FileNameInvalid</code>	The properties file name is invalid.
<code>FileNameTooLong</code>	The properties file name is too long.
<code>FileNotFoundException</code>	The properties file is not found.
<code>FileAccessDenied</code>	Access to the properties file is denied.
<code>Corrupted</code>	The properties file is corrupted (the content does not match its hash).
<code>XmlReadFailed</code>	The properties content does not conform to the required format.
<code>XmlWriteFailed</code>	Writing the properties XML has failed.
<code>BlobRequestFailed</code>	The Blob service request to access the properties has failed.
<code>IOFailed</code>	A disk or network I/O failure occurred while processing the properties.
<code>Failed</code>	An unknown failure occurred while processing the properties.
<code>Cancelled</code>	A prior failure has stopped further processing of the properties.

Sample Logs

The following is an example of verbose log.

```

<?xml version="1.0" encoding="UTF-8"?>
<DriveLog Version="2014-11-01">
  <DriveId>WD-WMATV123456</DriveId>
  <Blob Status="Completed">
    <BlobPath>pictures/bob/wild/desert.jpg</BlobPath>
    <FilePath>\Users\bob\Pictures\wild\desert.jpg</FilePath>
    <Length>98304</Length>
    <ImportDisposition Status="Created">overwrite</ImportDisposition>
    <BlockList>
      <Block Offset="0" Length="65536" Id="AAAAAA==" Hash=" 9C8AE14A55241F98533C4D80D85CDC68" Status="Completed"/>
      <Block Offset="65536" Length="32768" Id="AQAAAA==" Hash=" DF54C531C9B3CA2570FDDDB3BCD0E27D" Status="Completed"/>
    </BlockList>
    <Metadata Status="Completed">
      <GlobalPath Hash=" E34F54B7086BCF4EC1601D056F4C7E37">\Users\bob\Pictures\wild\metadata.xml</GlobalPath>
    </Metadata>
  </Blob>
  <Blob Status="CompletedWithErrors">
    <BlobPath>pictures/bob/animals/koala.jpg</BlobPath>
    <FilePath>\Users\bob\Pictures\animals\koala.jpg</FilePath>
    <Length>163840</Length>
    <ImportDisposition Status="Overwritten">overwrite</ImportDisposition>
    <PageRangeList>
      <PageRange Offset="0" Length="65536" Hash="19701B8877418393CB3CB567F53EE225" Status="Completed"/>
      <PageRange Offset="65536" Length="65536" Hash="AA2585F6F6FD01C4AD4256E018240CD4" Status="Corrupted"/>
      <PageRange Offset="131072" Length="4096" Hash="9BA552E1C3EEAFFC91B42B979900A996" Status="Completed"/>
    </PageRangeList>
    <Properties Status="Completed">
      <Path Hash="38D7AE80653F47F63C0222FEE90EC4E7">\Users\bob\Pictures\animals\koala.jpg.properties</Path>
    </Properties>
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>

```

The corresponding error log is shown below.

```

<?xml version="1.0" encoding="UTF-8"?>
<DriveLog Version="2014-11-01">
  <DriveId>WD-WMATV6965824</DriveId>
  <Blob Status="CompletedWithErrors">
    <BlobPath>pictures/bob/animals/koala.jpg</BlobPath>
    <FilePath>\Users\bob\Pictures\animals\koala.jpg</FilePath>
    <Length>163840</Length>
    <ImportDisposition Status="Overwritten">overwrite</ImportDisposition>
    <PageRangeList>
      <PageRange Offset="65536" Length="65536" Hash="AA2585F6F6FD01C4AD4256E018240CD4" Status="Corrupted"/>
    </PageRangeList>
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>

```

The follow error log for an import job contains an error about a file not found on the import drive. Note that the status of subsequent components is **Cancelled**.

```

<?xml version="1.0" encoding="utf-8"?>
<DriveLog Version="2014-11-01">
  <DriveId>9WM35C2V</DriveId>
  <Blob Status="FileNotFoundException">
    <BlobPath>pictures/animals/koala.jpg</BlobPath>
    <FilePath>\animals\koala.jpg</FilePath>
    <Length>30310</Length>
    <ImportDisposition Status="Cancelled">rename</ImportDisposition>
    <BlockList>
      <Block Offset="0" Length="6062" Id="MD5/cAzn4h7VVSWXf696qp5Uaw==" Hash="700CE7E21ED55525977FAF7AAA9E546B" Status="Cancelled" />
      <Block Offset="6062" Length="6062" Id="MD5/PEnGwYOI8PLNYdfKr7KAg==" Hash="3C49C6C18388F0B3CB35875F2ABEE402" Status="Cancelled" />
      <Block Offset="12124" Length="6062" Id="MD5/FG4WxqfZKuUWZ2nGTU2qVA==" Hash="146E16C6A7D92AE5166769C64D4DA54" Status="Cancelled" />
      <Block Offset="18186" Length="6062" Id="MD5/ZzibNDzr3IRBQENRygegeXQ==" Hash="67389B343CEBDC8441404351C9E81E5D" Status="Cancelled" />
      <Block Offset="24248" Length="6062" Id="MD5/ZzibNDzr3IRBQENRygegeXQ==" Hash="67389B343CEBDC8441404351C9E81E5D" Status="Cancelled" />
    </BlockList>
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>

```

The following error log for an export job indicates that the blob content has been successfully written to the drive, but that an error occurred while exporting the blob's properties.

```
<?xml version="1.0" encoding="utf-8"?>
<DriveLog Version="2014-11-01">
  <DriveId>9WM35C3U</DriveId>
  <Blob Status="CompletedWithErrors">
    <BlobPath>pictures/wild/canyon.jpg</BlobPath>
    <FilePath>\pictures\wild\canyon.jpg</FilePath>
    <LastModified>2012-09-18T23:47:08Z</LastModified>
    <Length>163840</Length>
    <BlockList />
    <Properties Status="Failed" />
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>
```

See Also

[Storage Import/Export REST](#)

Using the Azure Import/Export Tool (v1)

1/17/2017 • 1 min to read • [Edit on GitHub](#)

The Azure Import/Export Tool (WAImpoertExport.exe) is used to create and manage jobs for the Azure Import/Export Service, enabling you to transfer large amounts of data into or out of Azure Blob Storage.

This documentation is for v1 of the Azure Import/Export Tool. For information about using the most recent version of the tool, please see [Using the Azure Import/Export Tool](#).

In these articles, you will see how to use the tool to do the following:

- Install and set up the Import/Export Tool.
- Prepare your hard drives for a job where you import data from your drives to Azure Blob Storage.
- Review the status of a job with Copy Log Files.
- Repair an import job.
- Repair an export job.
- Troubleshoot the Azure Import/Export Tool, in case you had a problem during process.

Setting Up the Azure Import-Export Tool

1/17/2017 • 6 min to read • [Edit on GitHub](#)

The Microsoft Azure Import/Export tool is the drive preparation and repair tool that you can use with the Microsoft Azure Import/Export Service. You can use the tool for the following functions:

- Before creating an import job, you can use this tool to copy data to the hard drives you are going to ship to a Windows Azure data center.
- After an import job has completed, you can use this tool to repair any blobs that were corrupted, were missing, or conflicted with other blobs.
- After you receive the drives from a completed export job, you can use this tool to repair any files that were corrupted or missing on the drives.

Prerequisites

If you are preparing drives for an import job, you will need to meet the following prerequisites:

- You must have an active Azure subscription.
- Your subscription must include a storage account with enough available space to store the files you are going to import.
- You need at least one of the account keys for the storage account.
- You need a computer (the "copy machine") with Windows 7, Windows Server 2008 R2, or a newer Windows operating system installed.
- The .NET Framework 4 must be installed on the copy machine.
- BitLocker must be enabled on the copy machine.
- You will need one or more drives that contains data to be imported or empty 3.5-inch SATA hard drives connected to the copy machine.
- The files you plan to import must be accessible from the copy machine, whether they are on a network share or a local hard drive.

If you are attempting to repair an import that has partially failed, you will need:

- The copy log files
- The storage account key

If you are attempting to repair an export that has partially failed, you will need:

- The copy log files
- The manifest files (optional)
- The storage account key

Installing the Azure Import/Export Tool

The Azure Import/Export tool consists of the following files:

- WAImportExport.exe
- WAImportExport.exe.config
- WAImportExportCore.dll
- WAImportExportRepair.dll
- Microsoft.WindowsAzure.Storage.dll
- Hddid.dll

Copy these files to a working directory, for example, `c:\WAImportExport`. Next, open a command line window in Administrator mode, and set the above directory as current directory.

To output help for the command, run the tool without parameters:

```
WAImportExport, a client tool for Microsoft Azure Import/Export Service. Microsoft (c) 2013, 2014
```

Copy a Directory:

```
WAImportExport.exe PrepImport
/j:<JournalFile> [/logdir:<LogDirectory>] [/id:<SessionId>] [/resumesession]
[/abortsession] [/sk:<StorageAccountKey>] [/csas:<ContainerSas>]
/t:<TargetDriveLetter> [/format] [/silentmode] [/encrypt]
/bk:<BitLockerKey> [/Disposition:<Disposition>] [/BlobType:<BlobType>]
[/PropertyFile:<PropertyFile>] [/MetadataFile:<MetadataFile>]
/srcdir:<SourceDirectory> /dstdir:<DestinationBlobVirtualDirectory>
```

Copy a File:

```
WAImportExport.exe PrepImport
/j:<JournalFile> [/logdir:<LogDirectory>] [/id:<SessionId>] [/resumesession]
[/abortsession] [/sk:<StorageAccountKey>] [/csas:<ContainerSas>]
/t:<TargetDriveLetter> [/format] [/silentmode] [/encrypt]
/bk:<BitLockerKey> [/Disposition:<Disposition>] [/BlobType:<BlobType>]
[/PropertyFile:<PropertyFile>] [/MetadataFile:<MetadataFile>]
/srcfile:<SourceFilePath> /dstblob:<DestinationBlobPath>
```

Repair a Drive:

```
WAImportExport.exe RepairImport | RepairExport
/r:<RepairFile> [/logdir:<LogDirectory>]
/d:<TargetDirectories>] [/bk:<BitLockerKey>]
/sn:<StorageAccountName> [/sk:<StorageAccountKey> | /csas:<ContainerSas>]
[/CopyLogFile:<DriveCopyLogFile>] [/ManifestFile:<DriveManifestFile>]
[/PathMapFile:<DrivePathMapFile>]
```

Preview an Export Job:

```
WAImportExport.exe PreviewExport
[/logdir:<LogDirectory>]
/sn:<StorageAccountName> [/sk:<StorageAccountKey> | /csas:<ContainerSas>]
/ExportBlobListFile:<ExportBlobListFile> /DriveSize:<DriveSize>
```

Parameters:

```
/j:<JournalFile>
- Required. Path to the journal file. Each drive must have one and only one
journal file. The journal file corresponding to the target drive must always
be specified.
/logdir:<LogDirectory>
- Optional. The log directory. Verbose log files as well as some temporary
files will be written to this directory. If not specified, current directory
will be used as the log directory.
/id:<SessionId>
- Required. The session Id is used to identify a copy session. It is used to
ensure accurate recovery of an interrupted copy session. In addition, files
that are copied in a copy session are stored in a directory named after the
session Id on the target drive.
/resumesession
```

- Optional. If the last copy session was terminated abnormally, this parameter can be specified to resume the session.

/abortsession

- Optional. If the last copy session was terminated abnormally, this parameter can be specified to abort the session.

/sn:<StorageAccountName>

- Required. Only applicable for RepairImport and RepairExport. The name of the storage account.

/sk:<StorageAccountKey>

- Optional. The key of the storage account. One of /sk: and /csas: must be specified.

/csas:<ContainerSas>

- Optional. A container SAS, in format of <ContainerName>?<SasString>, to be used for import the data. One of /sk: and /csas: must be specified.

/t:<TargetDriveLetter>

- Required. Drive letter of the target drive.

/r:<RepairFile>

- Required. Only applicable for RepairImport and RepairExport.
Path to the file for tracking repair progress. Each drive must have one and only one repair file.

/d:<TargetDirectories>

- Required. Only applicable for RepairImport and RepairExport.
For RepairImport, one or more semicolon-separated directories to repair;
For RepairExport, one directory to repair, e.g. root directory of the drive.

/format

- Optional. If specified, the target drive will be formatted. DO NOT specify this parameter if you do not want to format the drive.

/silentmode

- Optional. If not specified, the /format parameter will require a confirmation from console before the tool formats the drive. If this parameter is specified, not confirmation will be given for formatting the drive.

/encrypt

- Optional. If specified, the target drive will be encrypted with BitLocker.
If the drive has already been encrypted with BitLocker, do not specify this parameter and instead specify the BitLocker key using the "/k" parameter.

/bk:<BitLockerKey>

- Optional. The current BitLocker key if the drive has already been encrypted with BitLocker.

/Disposition:<Disposition>

- Optional. Specifies the behavior when a blob with the same path as the one being imported already exists. Valid values are: rename, no-overwrite and overwrite (case-sensitive). If not specified, "rename" will be used as the default value.

/BlobType:<BlobType>

- Optional. The blob type for the imported blob(s). Valid values are BlockBlob and PageBlob. If not specified, BlockBlob will be used as the default value.

/PropertyFile:<PropertyFile>

- Optional. Path to the property file for the file(s) to be imported.

/MetadataFile:<MetadataFile>

- Optional. Path to the metadata file for the file(s) to be imported.

/CopyLogFile:<DriveCopyLogFile>

- Required. Only applicable for RepairImport and RepairExport. Path to the drive copy log file (verbose or error).

/ManifestFile:<DriveManifestFile>

- Required. Only applicable for RepairExport. Path to the drive manifest file.

/PathMapFile:<DrivePathMapFile>

- Optional. Only applicable for RepairImport. Path to the file containing mappings of file paths relative to the drive root to locations of actual files (tab-delimited). When first specified, it will be populated with file paths with empty targets, which means either they are not found in TargetDirectories, access denied, with invalid name, or they exist in multiple directories. The path map file can be manually edited to include the correct target paths and specified again for the tool to resolve the file paths correctly.

/ExportBlobListFile:<ExportBlobListFile>

- Required. Path to the XML file containing list of blob paths or blob path prefixes for the blobs to be exported. The file format is the same as the blob list blob format in the Put Job operation of the Import/Export Service REST API.

/DriveSize:<DriveSize>

```

- Required. Size of drives to be used for export. For example, 500GB, 1.5TB.
  Note: 1 GB = 1,000,000,000 bytes
  1 TB = 1,000,000,000,000 bytes
/srcdir:<SourceDirectory>
- Required. Source directory that contains files to be copied to the
target drives.
/dstdir:<DestinationBlobVirtualDirectory>
- Required. Destination blob virtual directory to which the files will
be imported.
/srcfile:<SourceFilePath>
- Required. Path to the source file to be imported.
/dstblob:<DestinationBlobPath>
- Required. Destination blob path for the file to be imported.
/skipwrite
- Optional. To skip write process. Used for inplace data drive preparation.
  Be sure to reserve enough space (3 GB per 7TB) for drive manifest file!

```

Examples:

Copy a source directory to a drive:

```
WAImportExport.exe PrepImport
/j:9WM35C2V.jrn /id:session#1 /sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN0lp+kp0C9MEdx3GEL
xmBw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ== /t:x /format /encrypt /srcdir:d:\movies\drama
/dstdir:movies/drama/
```

Copy another directory to the same drive following the above command:

```
WAImportExport.exe PrepImport
/j:9WM35C2V.jrn /id:session#2 /srcdir:d:\movies\action /dstdir:movies\action/
```

Copy another file to the same drive following the above commands:

```
WAImportExport.exe PrepImport
/j:9WM35C2V.jrn /id:session#3 /srcfile:d:\movies\dvd.vhd /dstblob:movies/dvd.vhd /BlobType:PageBlob
```

Preview how many 1.5 TB drives are needed for an export job:

```
WAImportExport.exe PreviewExport
/sn:mytestaccount /sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN0lp+kp0C9MEdx3GELxmBw4hK94f7K
ysbbeKLDksg7VoN1W/a5UuM2zNgQ== /ExportBlobListFile:C:\temp\myexportbloblist.xml
/DriveSize:1.5TB
```

Repair an finished import job:

```
WAImportExport.exe RepairImport
/r:9WM35C2V.rep /d:X:\ /bk:442926-020713-108086-436744-137335-435358-242242-2795
98 /sn:mytestaccount /sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN0lp+kp0C9MEdx3GELxmBw4hK94
f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ== /CopyLogFile:C:\temp\9WM35C2V_error.log
```

Skip write process, inplace data drive preparation:

```
WAImportExport.exe PrepImport
/j:9WM35C2V.jrn /id:session#1 /sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN0lp+kp0C9MEdx3GEL
xmBw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ== /t:d /encrypt /srcdir:d:\movies\drama
/dstdir:movies/drama/ /skipwrite
```

See Also

[Preparing Hard Drives for an Import Job](#)

[Previewing Drive Usage for an Export Job](#)

[Reviewing Job Status with Copy Log Files](#)

[Repairing an Import Job](#)

[Repairing an Export Job](#)

[Troubleshooting the Azure Import-Export Tool](#)

Preparing Hard Drives for an Import Job

1/17/2017 • 11 min to read • [Edit on GitHub](#)

To prepare one or more hard drives for an import job, follow these steps:

- Identify the data to import into the Blob service
- Identify target virtual directories and blobs in the Blob service
- Determine how many drives you'll need
- Copy the data to each of your hard drives

For a sample workflow, see [Sample Workflow to Prepare Hard Drives for an Import Job](#).

Identify the Data to Be Imported

The first step to creating an import job is to determine which directories and files you are going to import. This can be a list of directories, a list of unique files, or a combination of those two. When a directory is included, all files in the directory and its subdirectories will be part of the import job.

NOTE

Since subdirectories are included recursively when a parent directory is included, specify only the parent directory. Do not also specify any of its subdirectories.

Currently, the Microsoft Azure Import/Export tool has the following limitation: if a directory contains more data than a hard drive can contain, then the directory needs to be broken into smaller directories. For example, if a directory contains 2.5TB of data and the hard drive's capacity is only 2TB, then you need to break the 2.5TB directory into smaller directories. This limitation will be addressed in a later version of the tool.

Identify the Destination Locations in the Blob Service

For each directory or file that will be imported, you need to identify a destination virtual directory or blob in the Azure Blob service. You will use these targets as inputs to the Azure Import/Export tool. Note that directories should be delimited with the forward slash character "/".

The following table shows some examples of blob targets:

SOURCE FILE OR DIRECTORY	DESTINATION BLOB OR VIRTUAL DIRECTORY
H:\Video	https://mystorageaccount.blob.core.windows.net/video
H:\Photo	https://mystorageaccount.blob.core.windows.net/photo
K:\Temp\FavoriteVideo.ISO	https://mystorageaccount.blob.core.windows.net/favorite/FavoriteVideo.ISO
\myshare\john\music	https://mystorageaccount.blob.core.windows.net/music

Determine How Many Drives Are Needed

Next, you need to determine:

- The number of hard drives needed to store the data.
- The directories and/or standalone files that will be copied to each of your hard drive.

Ensure that you have the number of hard drives you will need to store the data you are transferring.

Copy Data to Your Hard Drive

This section describes how to call the Azure Import/Export tool to copy your data to one or more hard drives. Each time you call the Azure Import/Export tool, you create a new *copy session*. You create at least one copy session for each drive to which you copy data; in

some cases, you may need more than one copy session to copy all of your data to single drive. Here are some reasons that you may need multiple copy sessions:

- You must create a separate copy session for each drive that you copy to.
- A copy session can copy either a single directory or a single blob to the drive. If you are copying multiple directories, multiple blobs, or a combination of both, you'll need to create multiple copy sessions.
- You can specify properties and metadata that will be set on the blobs imported as part of an import job. The properties or metadata that you specify for a copy session will apply to all blobs specified by that copy session. If you want to specify different properties or metadata for some blobs, you'll need to create a separate copy session. See [Setting Properties and Metadata during the Import Process](#)for more information.

NOTE

If you have multiple machines that meet the requirements outlined in [Setting Up the Azure Import-Export Tool](#), you can copy data to multiple hard drives in parallel by running an instance of this tool on each machine.

For each hard drive that you prepare with the Azure Import/Export tool, the tool will create a single journal file. You will need the journal files from all of your drives to create the import job. The journal file can also be used to resume drive preparation if the tool is interrupted.

Azure Import/Export Tool Syntax for an Import Job

To prepare drives for an import job, call the Azure Import/Export tool with the **PrepImport** command. Which parameters you include depends on whether this is the first copy session, or a subsequent copy session.

The first copy session for a drive requires some additional parameters to specify the storage account key; the target drive letter; whether the drive must be formatted; whether the drive must be encrypted and if so, the BitLocker key; and the log directory. Here is the syntax for an initial copy session to copy a directory or a single file:

First Copy Session to Copy a Single Directory

```
wAIImportExport PrepImport /sk:<StorageAccountKey> /csas:<ContainerSas> /t: <TargetDriveLetter> [/format] [/silentmode] [/encrypt]
[/bk:<BitLockerKey>] [/logdir:<LogDirectory>] /j:<JournalFile> /id:<SessionId> /srcdir:<SourceDirectory> /dstdir:
<DestinationBlobVirtualDirectory> [/Disposition:<Disposition>] [/BlobType:<BlockBlob|PageBlob>] [/PropertyFile:<PropertyFile>]
[_METADATAFILE:<MetadataFile>]
```

First Copy Session to Copy a Single File

```
wAIImportExport PrepImport /sk:<StorageAccountKey> /csas:<ContainerSas> /t: <TargetDriveLetter> [/format] [/silentmode] [/encrypt]
[/bk:<BitLockerKey>] [/logdir:<LogDirectory>] /j:<Journalfile> /id:<SessionId> /srcfile:<SourceFile> /dstblob:<DestinationBlobPath>
[/Disposition:<Disposition>] [/BlobType:<BlockBlob|PageBlob>] [/PropertyFile:<PropertyFile>] [/MetadataFile:<MetadataFile>]
```

In subsequent copy sessions, you do not need to specify the initial parameters. Here is the syntax for a subsequent copy session to copy a directory or a single file:

Subsequent Copy Sessions to Copy a Single Directory

```
wAIImportExport PrepImport /j:<JournalFile> /id:<SessionId> /srcdir:<SourceDirectory> /dstdir:<DestinationBlobVirtualDirectory>
[/Disposition:<Disposition>] [/BlobType:<BlockBlob|PageBlob>] [/PropertyFile:<PropertyFile>] [/MetadataFile:<MetadataFile>]
```

Subsequent Copy Sessions to Copy a Single File

```
wAIImportExport PrepImport /j:<JournalFile> /id:<SessionId> /srcfile:<SourceFile> /dstblob:<DestinationBlobPath> [/Disposition:
<Disposition>] [/BlobType:<BlockBlob|PageBlob>] [/PropertyFile:<PropertyFile>] [/MetadataFile:<MetadataFile>]
```

Parameters for the First Copy Session for a Hard Drive

Each time you run the Azure Import/Export tool to copy files to the hard drive, the tool creates a copy session. Each copy session copies a single directory or a single file to a hard drive. The state of the copy session is written to the journal file. If a copy session is interrupted (for example, due to a system power loss), it can be resumed by running the tool again and specifying the journal file on the command line.

WARNING

If you specify the **/format** parameter for the first copy session, the drive will be formatted and all data on the drive will be erased. It's recommended that you use blank drives only for your copy session.

The command used for the first copy session for each drive requires different parameters than the commands for subsequent copy sessions. The following table lists the additional parameters that are available for the first copy session:

COMMAND-LINE PARAMETER	DESCRIPTION
/sk:	Optional. The storage account key for the storage account to which the data will be imported. You must include either /sk: or /csas: in the command.
/csas:	Optional. The container SAS to use to import data to the storage account. You must include either /sk: or /csas: in the command. The value for this parameter must begin with the container name, followed by a question mark (?) and the SAS token. For example: <pre>mycontainer?sv=2014-02-14&sr=c&si=abcde&sig=LiqEmV%2Fs1LF4loC%2FJs9ZM91%2FkfqFqKhnz0JM6bqIqN0%11-20T23%3A54%3A14Z&sp=rwdl</pre> The permissions, whether specified on the URL or in a stored access policy, must include Read, Write, and Delete for import jobs, and Read, Write and List for export jobs. When this parameter is specified, all blobs to be imported or exported must be within the container specified in the shared access signature.
/t:	Required. The drive letter of the target hard drive for the current copy session, without the trailing colon.
/format	Optional. Specify this parameter when the drive needs to be formatted; otherwise, omit it. Before the tool formats the drive, it will prompt for a confirmation from console. To suppress the confirmation, specify the /silentmode parameter.
/silentmode	Optional. Specify this parameter to suppress the confirmation for formatting the target drive.
/encrypt	Optional. Specified this parameter when the drive has not yet been encrypted with BitLocker and needs to be encrypted by the tool. If the drive has already been encrypted with BitLocker, then omit this parameter and specify the /blk parameter, providing the existing BitLocker key. If you specify the /format parameter, then you must also specify the /encrypt parameter.
/blk:	Optional. If /encrypt is specified, omit this parameter. If /encrypt is omitted, you need to have already have encrypted the drive with BitLocker. Use this parameter to specify the BitLocker key. BitLocker encryption is required for all hard drives for import jobs.
/logdir:	Optional. The log directory specifies a directory to be used to store verbose logs as well as temporary manifest files. If not specified, the current directory will be used as the log directory.

Parameters Required for All Copy Sessions

The journal file contains the status for all copy sessions for a hard drive. It also contains the information needed to create the import job. You must always specify a journal file when running the Azure Import/Export tool, as well as a copy session ID:

Command line parameter	Description
/j:	Required. The path to the journal file. Each drive must have exactly one journal file. Note that the journal file must not reside on the target drive. The journal file extension is .jrn .
/id:	Required. The session ID identifies a copy session. It is used to ensure accurate recovery of an interrupted copy session. Files that are copied in a copy session are stored in a directory named after the session ID on the target drive.

Parameters for Copying a Single Directory

When copying a single directory, the following required and optional parameters apply:

COMMAND LINE PARAMETER	DESCRIPTION
/srcdir:	<p>Required. The source directory that contains files to be copied to the target drive. The directory path must be an absolute path (not a relative path).</p>
/dstdir:	<p>Required. The path to the destination virtual directory in your Windows Azure storage account. The virtual directory may or may not already exist.</p> <p>You can specify a container, or a blob prefix like <code>music/70s/</code>. The destination directory must begin with the container name, followed by a forward slash "/", and optionally may include a virtual blob directory that ends with "/".</p> <p>When the destination container is the root container, you must explicitly specify the root container, including the forward slash, as <code>\$blob\$</code>. Since blobs under the root container cannot include "/" in their names, any subdirectories in the source directory will not be copied when the destination directory is the root container.</p> <p>Be sure to use valid container names when specifying destination virtual directories or blobs. Keep in mind that container names must be lowercase. For container naming rules, see Naming and Referencing Containers, Blobs, and Metadata.</p>
/Disposition:	<p>Optional. Specifies the behavior when a blob with the specified address already exists. Valid values for this parameter are: <code>rename</code>, <code>no-overwrite</code> and <code>overwrite</code>. Note that these values are case-sensitive. If no value is specified, the default is <code>rename</code>.</p> <p>The value specified for this parameter affects all the files in the directory specified by the <code>/srcdir</code> parameter.</p>
/BlobType:	<p>Optional. Specifies the blob type for the destination blobs. Valid values are: <code>BlockBlob</code> and <code>PageBlob</code>. Note that these values are case-sensitive. If no value is specified, the default is <code>BlockBlob</code>.</p> <p>In most cases, <code>BlockBlob</code> is recommended. If you specify <code>PageBlob</code>, the length of each file in the directory must be a multiple of 512, the size of a page for page blobs.</p>
/PropertyFile:	<p>Optional. Path to the property file for the destination blobs. See Import-Export Service Metadata and Properties File Format for more information.</p>
/MetadataFile:	<p>Optional. Path to the metadata file for the destination blobs. See Import-Export Service Metadata and Properties File Format for more information.</p>

Parameters for Copying a Single File

When copying a single file, the following required and optional parameters apply:

COMMAND LINE PARAMETER	DESCRIPTION
/srcfile:	<p>Required. The full path to the file to be copied. The directory path must be an absolute path (not a relative path).</p>

COMMAND LINE PARAMETER	DESCRIPTION
/dstblob:	<p>Required. The path to the destination blob in your Windows Azure storage account. The blob may or may not already exist.</p> <p>Specify the blob name beginning with the container name. The blob name cannot start with "/" or the storage account name. For blob naming rules, see Naming and Referencing Containers, Blobs, and Metadata.</p> <p>When the destination container is the root container, you must explicitly specify <code>\$root</code> as the container, such as <code>\$root/sample.txt</code>. Note that blobs under the root container cannot include "/" in their names.</p>
/Disposition:	<p>Optional. Specifies the behavior when a blob with the specified address already exists. Valid values for this parameter are: <code>rename</code>, <code>no-overwrite</code> and <code>overwrite</code>. Note that these values are case-sensitive. If no value is specified, the default is <code>rename</code>.</p>
/BlobType:	<p>Optional. Specifies the blob type for the destination blobs. Valid values are: <code>BlockBlob</code> and <code>PageBlob</code>. Note that these values are case-sensitive. If no value is specified, the default is <code>BlockBlob</code>.</p> <p>In most cases, <code>BlockBlob</code> is recommended. If you specify <code>PageBlob</code>, the length of each file in the directory must be a multiple of 512, the size of a page for page blobs.</p>
/PropertyFile:	<p>Optional. Path to the property file for the destination blobs. See Import-Export Service Metadata and Properties File Format for more information.</p>
/MetadataFile:	<p>Optional. Path to the metadata file for the destination blobs. See Import-Export Service Metadata and Properties File Format for more information.</p>

Resuming an Interrupted Copy Session

If a copy session is interrupted for any reason, you can resume it by running the tool with only the journal file specified:

```
WAImportExport.exe PrepImport /j:<JournalFile> /id:<SessionId> /ResumeSession
```

Only the most recent copy session, if terminated abnormally, can be resumed.

IMPORTANT

When you resume a copy session, do not modify the source data files and directories by adding or removing files.

Aborting an Interrupted Copy Session

If a copy session is interrupted and it is not possible to resume (for example, if a source directory proved inaccessible), you must abort the current session so that it can be rolled back and new copy sessions can be started:

```
WAImportExport.exe PrepImport /j:<JournalFile> /id:<SessionId> /AbortSession
```

Only the last copy session, if terminated abnormally, can be aborted. Note that you cannot abort the first copy session for a drive. Instead you must restart the copy session with a new journal file.

See Also

[Setting Up the Azure Import-Export Tool Setting Properties and Metadata during the Import Process Sample Workflow to Prepare Hard Drives for an Import Job](#) [Quick Reference for Frequently Used Commands](#) [Reviewing Job Status with Copy Log Files](#) [Repairing an Import Job](#) [Repairing an Export Job](#) [Troubleshooting the Azure Import-Export Tool](#)

Setting Properties and Metadata during the Import Process

1/17/2017 • 1 min to read • [Edit on GitHub](#)

When you run the Microsoft Azure Import/Export tool to prepare your drives, you can specify properties and metadata to be set on the destination blobs. Follow these steps:

1. To set blob properties, create a text file on your local computer that specifies property names and values.
2. To set blob metadata, create a text file on your local computer that specifies metadata names and values.
3. Pass the full path to one or both of these files to the Azure Import/Export tool as part of the `PrepImport` operation.

NOTE

When you specify a properties or metadata file as part of a copy session, those properties or metadata are set for every blob that is imported as part of that copy session. If you want to specify a different set of properties or metadata for some of the blobs being imported, you'll need to create a separate copy session with different properties or metadata files.

Specify Blob Properties in a Text File

To specify blob properties, create a local text file, and include XML that specifies property names as elements, and property values as values. Here's an example that specifies some property values:

```
<?xml version="1.0" encoding="UTF-8"?>
<Properties>
    <Content-Type>application/octet-stream</Content-Type>
    <Content-MD5>Q2h1Y2sgSW50ZWdyaxR5IQ==</Content-MD5>
    <Cache-Control>no-cache</Cache-Control>
</Properties>
```

Save the file to a local location like `C:\WAImportExport\ImportProperties.txt`.

Specify Blob Metadata in a Text File

Similarly, to specify blob metadata, create a local text file that specifies metadata names as elements, and metadata values as values. Here's an example that specifies some metadata values:

```
<?xml version="1.0" encoding="UTF-8"?>
<Metadata>
    <UploadMethod>Windows Azure Import/Export Service</UploadMethod>
    <DataSetName>SampleData</DataSetName>
    <CreationDate>10/1/2013</CreationDate>
</Metadata>
```

Save the file to a local location like `C:\WAImportExport\ImportMetadata.txt`.

Create a Copy Session Including the Properties or Metadata Files

When you run the Azure Import/Export tool to prepare the import job, specify the properties file on the command

line using the `PropertyFile` parameter. Specify the metadata file on the command line using the `/MetadataFile` parameter. Here's an example that specifies both files:

```
WAIImportExport.exe PrepImport /j:SecondDrive.jrn /id:BlueRayIso /srcfile:K:\Temp\BlueRay.ISO  
/dstblob:favorite/BlueRay.ISO /MetadataFile:c:\WAIImportExport\SampleMetadata.txt  
/PropertyFile:c:\WAIImportExport\SampleProperties.txt
```

See Also

[Import-Export Service Metadata and Properties File Format](#)

Sample Workflow to Prepare Hard Drives for an Import Job

1/17/2017 • 3 min to read • [Edit on GitHub](#)

This topic walks you through the complete process of preparing drives for an import job.

This example imports the following data into a Window Azure storage account named `mystorageaccount`:

LOCATION	DESCRIPTION
H:\Video	A collection of videos, 5 TB in total.
H:\Photo	A collection of photos, 30 GB in total.
K:\Temp\FavoriteMovie.ISO	A Blu-Ray™ disk image, 25 GB.
\bigshare\john\music	A collection of music files on a network share, 10 GB in total.

The import job will import this data into the following destinations in the storage account:

SOURCE	DESTINATION VIRTUAL DIRECTORY OR BLOB
H:\Video	https://mystorageaccount.blob.core.windows.net/video
H:\Photo	https://mystorageaccount.blob.core.windows.net/photo
K:\Temp\FavoriteMovie.ISO	https://mystorageaccount.blob.core.windows.net/favorite/FavoriteMovies.ISO
\bigshare\john\music	https://mystorageaccount.blob.core.windows.net/music

With this mapping, the file `H:\Video\ Drama\GreatMovie.mov` will be imported to the blob

`https://mystorageaccount.blob.core.windows.net/video/Drama/GreatMovie.mov`.

Next, to determine how many hard drives are needed, compute the size of the data:

$$5\text{TB} + 30\text{GB} + 25\text{GB} + 10\text{GB} = 5\text{TB} + 65\text{GB}$$

For this example, two 3TB hard drives should be sufficient. However, since the source directory `H:\Video` has 5TB of data and your single hard drive's capacity is only 3TB, it's necessary to break `H:\Video` into two smaller directories before running the Microsoft Azure Import/Export tool: `H:\Video1` and `H:\Video2`. This step yields the following source directories:

LOCATION	SIZE	DESTINATION VIRTUAL DIRECTORY OR BLOB
H:\Video1	2.5TB	https://mystorageaccount.blob.core.windows.net/video

LOCATION	SIZE	DESTINATION VIRTUAL DIRECTORY OR BLOB
H:\Video2	2.5TB	https://mystorageaccount.blob.core.windows.net/video
H:\Photo	30GB	https://mystorageaccount.blob.core.windows.net/photo
K:\Temp\FavoriteMovies.ISO	25GB	https://mystorageaccount.blob.core.windows.net/favorite/FavoriteMovies.ISO
\bigshare\john\music	10GB	https://mystorageaccount.blob.core.windows.net/music

Note that even though the `H:\Video` directory has been split to two directories, they point to the same destination virtual directory in the storage account. This way, all video files are maintained under a single `video` container in the storage account.

Next, the above source directories are evenly distributed to the two hard drives:

Hard drive	Source directories	Total size
First Drive	H:\Video1	2.5TB + 30GB
	H:\Photo	
Second Drive	H:\Video2	2.5TB + 35GB
	K:\Temp\BlueRay.ISO	
	\bigshare\john\music	

In addition, you can set the following metadata for all files:

- **UploadMethod:** Windows Azure Import/Export Service
- **DataSetName:** SampleData
- **CreationDate:** 10/1/2013

To set metadata for the imported files, create a text file, `c:\WAImportExport\SampleMetadata.txt`, with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<Metadata>
    <UploadMethod>Windows Azure Import/Export Service</UploadMethod>
    <DataSetName>SampleData</DataSetName>
    <CreationDate>10/1/2013</CreationDate>
</Metadata>
```

You can also set some properties for the `FavoriteMovie.ISO` blob:

- **Content-Type:** application/octet-stream
- **Content-MD5:** Q2hIY2sgSW50ZWdyaXR5IQ==

- **Cache-Control:** no-cache

To set these properties, create a text file, `c:\WAIImportExport\SampleProperties.txt` :

```
<?xml version="1.0" encoding="UTF-8"?>
<Properties>
    <Content-Type>application/octet-stream</Content-Type>
    <Content-MD5>Q2h1Y2sgSW50ZWdyaxR5IQ==</Content-MD5>
    <Cache-Control>no-cache</Cache-Control>
</Properties>
```

Now you are ready to run the Azure Import/Export tool to prepare the two hard drives. Note that:

- The first drive is mounted as drive X.
- The second drive is mounted as drive Y.
- The key for the storage account `mystorageaccount` is
`8ImTigJhIwvL9VEIQKB/zbqcXbxrIHbBjLIf0t0tyR98TxtFvUM/7T0KVNR6KRkJrh26u5I8hTxTLM201aDVqg==`.

Preparing disk for import when data is pre-loaded

If the data to be imported is already present on the disk, use the flag `/skipwrite`. Value of `/t` and `/srcdir` both should point to the disk being prepared for import. If not all the data on the disk needs to go to the same destination virtual directory or root of the storage account, run the same command for each directory separately keeping the value of `/id` same across all runs.

NOTE

Do not specify `/format` as it will wipe the data on the disk. You can specify `/encrypt` or `/bk` depending on whether the disk is already encrypted or not.

When data is already present on the disk for each drive run the following command.

```
WAIImportExport.exe PrepImport /j:FirstDrive.jrn /id:Video1 /logdir:c:\logs
/sk:8ImTigJhIwvL9VEIQKB/zbqcXbxrIHbBjLIf0t0tyR98TxtFvUM/7T0KVNR6KRkJrh26u5I8hTxTLM201aDVqg== /t:x /format
/encrypt /srcdir:x:\Video1 /dstdir:video/ /MetadataFile:c:\WAIImportExport\SampleMetadata.txt /skipwrite
```

For the first drive, run the Azure Import/Export tool twice to copy the two source directories:

```
## First copy session for first drive
WAIImportExport.exe PrepImport /j:FirstDrive.jrn /id:Video1 /logdir:c:\logs
/sk:8ImTigJhIwvL9VEIQKB/zbqcXbxrIHbBjLIf0t0tyR98TxtFvUM/7T0KVNR6KRkJrh26u5I8hTxTLM201aDVqg== /t:x /format
/encrypt /srcdir:H:\Video1 /dstdir:video/ /MetadataFile:c:\WAIImportExport\SampleMetadata.txt
```

```
## Second copy session for first drive
WAIImportExport.exe PrepImport /j:FirstDrive.jrn /id:Photo /srcdir:H:\Photo /dstdir:photo/
/MetadataFile:c:\WAIImportExport\SampleMetadata.txt
```

For the second drive, run the Azure Import/Export tool three times, once each for the source directories and once for the standalone Blu-Ray™ image file):

```
## First copy session
WAIImportExport.exe PrepImport /j:SecondDrive.jrn /id:Video2 /logdir:c:\logs
/sk:8ImTigJhIwvL9VEIQKB/zbqcXbxrIHbBjLIf0t0tyR98TxtFvUM/7T0KVNR6KRkJrh26u5I8hTxTLM201aDVqg== /t:y /format
/encrypt /srcdir:H:\Video2 /dstdir:video/ /MetadataFile:c:\WAIImportExport\SampleMetadata.txt
```

```
## Second copy session
WAIImportExport.exe PrepImport /j:SecondDrive.jrn /id:Music /srcdir:\\bigshare\john\music /dstdir:music/
/MetadataFile:c:\WAIImportExport\SampleMetadata.txt
```

```
## Third copy session
WAIImportExport.exe PrepImport /j:SecondDrive.jrn /id:BlueRayIso /srcfile:K:\Temp\BlueRay.ISO
/dstblob:favorite/BlueRay.ISO /MetadataFile:c:\WAIImportExport\SampleMetadata.txt
/PropertyFile:c:\WAIImportExport\SampleProperties.txt
```

Once the copy sessions have completed, you can disconnect the two drives from the copy computer and ship them to the appropriate Windows Azure data center. You'll upload the two journal files, `FirstDrive.jrn` and `SecondDrive.jrn`, when you create the import job in the [Windows Azure Management Portal](#).

See Also

[Preparing Hard Drives for an Import Job](#)

[Quick Reference for Frequently Used Commands](#)

Quick reference for frequently used commands for import jobs

1/17/2017 • 1 min to read • [Edit on GitHub](#)

This section provides a quick references for some frequently used commands. For detailed usage, see [Preparing Hard Drives for an Import Job](#).

Prepare the disks when data already copied to the disks

Here is a sample command to prepare a disks when data already copied to the hard drive that hasn't been yet been encrypted with BitLocker:

```
WAIImportExport.exe PrepImport /j:9WM35C2V.jrn /id:session#1  
/sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN0lp+kp0C9MEDx3GELxmBw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ== /t:d /encrypt  
/srcdir:d:\movies\drama /dstdir:movies/drama/ /skipwrite
```

Copy a single directory to a hard drive

Here is a sample command to copy a single source directory to a hard drive that hasn't been yet been encrypted with BitLocker:

```
WAIImportExport.exe PrepImport /j:FirstDrive.jrn /id:movies /logdir:c:\logs  
/sk:8ImTigJhIwl9VEIQKB/zbqcXbxrIHbBjLIf0t0tyR98TxtFvUM/7T0KVNR6KRkJrh26u5I8hTxTLM201aDVqg== /t:x /format  
/encrypt /srcdir:d:\Movies /dstdir:entertainment/movies/
```

Copy two directories to a hard drive

To copy two source directories to a drive, you will need two commands.

The first command specifies the log directory, storage account key, target drive letter and **Format/Encrypt** requirements, in addition to the common parameters:

```
WAIImportExport.exe PrepImport /j:FirstDrive.jrn /id:movies /logdir:c:\logs  
/sk:8ImTigJhIwl9VEIQKB/zbqcXbxrIHbBjLIf0t0tyR98TxtFvUM/7T0KVNR6KRkJrh26u5I8hTxTLM201aDVqg== /t:x /format  
/encrypt /srcdir:d:\Movies /dstdir:entertainment/movies/
```

The second command specifies the journal file, a new session ID, and the source and destination locations:

```
WAIImportExport.exe PrepImport /j:FirstDrive.jrn /id:music /srcdir:d:\Music /dstdir:entertainment/music/
```

Copy a large file to a hard drive in a second copy session

Here is a sample command that copies a single large file to a drive that has been prepared in a previous copy session:

```
WAIImportExport.exe PrepImport /j:FirstDrive.jrn /id:dvd /srcfile:d:\dvd\favoritemovie.vhd  
/dstblob:dvd/favoritemovie.vhd
```

See also

[Sample Workflow to Prepare Hard Drives for an Import Job](#)

Previewing Drive Usage for an Export Job

1/17/2017 • 2 min to read • [Edit on GitHub](#)

Before you create an export job, you need to choose a set of blobs that are to be exported. The Microsoft Azure Import/Export service allows you to use a list of blob paths or blob prefixes to represent the blobs you have selected.

Next you need to determine how many drives you need to send. The Microsoft Azure Import/Export tool provides the `PreviewExport` command to preview drive usage for the blobs you selected, based on the size of the drives you are going to use. You can specify the following parameters:

COMMAND-LINE OPTION	DESCRIPTION
<code>/logdir:</code>	Optional. The log directory. Verbose log files will be written to this directory. If no log directory is specified, the current directory will be used as the log directory.
<code>/sn:</code>	Required. The name of the storage account for the export job.
<code>/sk:</code>	Required if and only if a container SAS is not specified. The account key for the storage account for the export job.
<code>/csas:</code>	Required if and only if a storage account key is not specified. The container SAS for listing the blobs to be exported in the export job.
<code>/ExportBlobListFile:</code>	Required. Path to the XML file containing list of blob paths or blob path prefixes for the blobs to be exported. The file format used in the <code>BlobListBlobPath</code> element in the Put Job operation of the Import/Export Service REST API.
<code>/DriveSize:</code>	Required. The size of drives to use for an export job, e.g., 500GB, 1.5TB.

The following example demonstrates the `PreviewExport` command:

```
WAImportExport.exe PreviewExport /sn:bobmediaaccount  
/sk:VkGbrUqBWLYJ6zg1m29V0TrxpBgdN0lp+kp0C9MEDx3GELxmBw4hK94f7KysbbeKLDksg7VoN1w/a5UuM2zNgQ==  
/ExportBlobListFile:C:\WAImportExport\mybloblist.xml /DriveSize:500GB
```

The export blob list file may contain blob names and blob prefixes, as shown here:

```
<?xml version="1.0" encoding="utf-8"?>  
<BlobList>  
<BlobPath>pictures/animals/koala.jpg</BlobPath>  
<BlobPathPrefix>/vhds/</BlobPathPrefix>  
<BlobPathPrefix>/movies/</BlobPathPrefix>  
</BlobList>
```

The Azure Import/Export tool lists all blobs to be exported and calculates how to pack them into drives of the specified size, taking into account any necessary overhead, then estimates the number of drives needed to hold the blobs and drive usage information.

Here is an example of the output, with informational logs omitted:

```
Number of unique blob paths/prefixes: 3
Number of duplicate blob paths/prefixes: 0
Number of nonexistent blob paths/prefixes: 1

Drive size: 500.00 GB
Number of blobs that can be exported: 6
Number of blobs that cannot be exported: 2
Number of drives needed: 3
  Drive #1: blobs = 1, occupied space = 454.74 GB
  Drive #2: blobs = 3, occupied space = 441.37 GB
  Drive #3: blobs = 2, occupied space = 131.28 GB
```

See Also

[Azure Import-Export Tool Reference](#)

Reviewing Job Status with Copy Log Files

1/17/2017 • 1 min to read • [Edit on GitHub](#)

When the Microsoft Azure Import/Export service processes drives associated with an import or export job, it writes copy log files to the storage account to or from which you are importing or exporting blobs. The log file contains detailed status about each file that was imported or exported. The URL to each copy log file is returned when you query the status of a completed job; see [Get Job](#) for more information.

The following are example URLs for copy log files for an import job with two drives:

```
http://myaccount.blob.core.windows.net/ImportExportStatesPath/waies/myjob_9WM35C2V_20130921-034307-902_error.xml
```

```
http://myaccount.blob.core.windows.net/ImportExportStatesPath /waies/myjob_9WM45A6Q_20130921-042122-021_error.xml
```

See [Import-Export Service Log File Format](#) for the format of copy logs and the full list of status codes.

See Also

[Setting Up the Azure Import-Export Tool](#)

[Preparing Hard Drives for an Import Job](#)

[Repairing an Import Job](#)

[Repairing an Export Job](#)

[Troubleshooting the Azure Import-Export Tool](#)

Repairing an Import Job

1/17/2017 • 4 min to read • [Edit on GitHub](#)

The Microsoft Azure Import/Export service may fail to copy some of your files or parts of a file to the Windows Azure Blob service. Some reasons for failures include:

- Corrupted files
- Damaged drives
- The storage account key changed while the file was being transferred.

You can run the Microsoft Azure Import/Export tool with the import job's copy log files, and the tool will upload the missing files (or parts of a file) to your Windows Azure storage account to complete import job.

The command for repairing an import job is **RepairImport**. The following parameters can be specified:

/r:	Required. Path to the repair file, which tracks the progress of the repair, and allows you to resume an interrupted repair. Each drive must have one and only one repair file. When you start a repair for a given drive, you will pass in the path to a repair file which does not yet exist. To resume an interrupted repair, you should pass in the name of an existing repair file. The repair file corresponding to the target drive must always be specified.
/logdir:	Optional. The log directory. Verbose log files will be written to this directory. If no log directory is specified, the current directory will be used as the log directory.
/d:	Required. One or more semicolon-separated directories that contain the original files that were imported. The import drive may also be used, but is not required if alternate locations of original files are available.
/bk:	Optional. You should specify the BitLocker key if you want the tool to unlock an encrypted drive where the original files are available.
/sn:	Required. The name of the storage account for the import job.
/sk:	Required if and only if a container SAS is not specified. The account key for the storage account for the import job.
/csas:	Required if and only if the storage account key is not specified. The container SAS for accessing the blobs associated with the import job.

/CopyLogFile:	Required. Path to the drive copy log file (either verbose log or error log). The file is generated by the Windows Azure Import/Export service and can be downloaded from the blob storage associated with the job. The copy log file contains information about failed blobs or files which are to be repaired.
/PathMapFile:	Optional. Path to a text file that can be used to resolve ambiguities if you have multiple files with the same name that you were importing in the same job. The first time the tool is run, it can populate this file with all of the ambiguous names. Subsequent runs of the tool will use this file to resolve the ambiguities.

Using the RepairImport Command

To repair import data by streaming the data over the network, you must specify the directories that contain the original files you were importing using the `/d` parameter. You must also specify the copy log file that you downloaded from your storage account. A typical command line to repair an import job with partial failures looks like:

```
WAImportExport.exe RepairImport /r:C:\WAImportExport\9WM35C2V.rep /d:C:\Users\bob\Pictures;X:\BobBackup\photos
/sn:bobmediaaccount
/sk:VkGbrUqBWLJ6zg1m29VOTrxpBgdN0lp+kp0C9MEdx3GELxmBw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ==
/CopyLogFile:C:\WAImportExport\9WM35C2V.log
```

The following is an example of a copy log file. In this case, one 64K piece of a file was corrupted on the drive that was shipped for the import job. Since this is the only failure indicated, the rest of the blobs in the job were successfully imported.

```
<?xml version="1.0" encoding="utf-8"?>
<DriveLog>
  <DriveId>9WM35C2V</DriveId>
  <Blob Status="CompletedWithErrors">
    <BlobPath>pictures/animals/koala.jpg</BlobPath>
    <FilePath>\animals\koala.jpg</FilePath>
    <Length>163840</Length>
    <ImportDisposition Status="Overwritten">overwrite</ImportDisposition>
    <PageRangeList>
      <PageRange Offset="65536" Length="65536" Hash="AA2585F6F6FD01C4AD4256E018240CD4" Status="Corrupted" />
    </PageRangeList>
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>
```

When this copy log is passed to the Azure Import/Export tool, the tool will try to finish the import for this file by copying the missing contents across the network. Following the example above, the tool will look for the original file `\animals\koala.jpg` within the two directories `C:\Users\bob\Pictures` and `X:\BobBackup\photos`. If the file `C:\Users\bob\Pictures\animals\koala.jpg` exists, the Azure Import/Export tool will copy the missing range of data to the corresponding blob `http://bobmediaaccount.blob.core.windows.net/pictures/animals/koala.jpg`.

Resolving Conflicts When Using RepairImport

In some situations, the tool may not be able to find or open the necessary file for one of the following reasons: the file could not be found, the file is not accessible, the file name is ambiguous, or the content of the file is no longer correct.

An ambiguous error could occur if the tool is trying to locate `\animals\koala.jpg` and there is a file with that name under both `C:\Users\bob\pictures` and `X:\BobBackup\photos`. That is, both `C:\Users\bob\pictures\animals\koala.jpg` and `X:\BobBackup\photos\animals\koala.jpg` exist on the import job drives.

The `/PathMapFile` option will allow you to resolve these errors. You can specify the name of the file which will contain the list of files that the tool was not able to correctly identify. The following is an example command line that would populate `9WM35C2V_pathmap.txt`:

```
WAIImportExport.exe RepairImport /r:C:\WAIImportExport\9WM35C2V.rep /d:C:\Users\bob\Pictures;X:\BobBackup\photos  
/sn:bobmediaaccount  
/sk:VkGbrUqBWLYJ6zg1m29VOTrxpBgdN0lp+kp0C9MEdx3GELxmBw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ==  
/CopyLogFile:C:\WAIImportExport\9WM35C2V.log /PathMapFile:C:\WAIImportExport\9WM35C2V_pathmap.txt
```

The tool will then write the problematic file paths to `9WM35C2V_pathmap.txt`, one on each line. For instance, the file may contain the following entries after running the command:

```
\animals\koala.jpg  
\animals\kangaroo.jpg
```

For each file in the list, you should attempt to locate and open the file to ensure it is available to the tool. If you wish to tell the tool explicitly where to find a file, you can modify the path map file and add the path to each file on the same line, separated by a tab character:

```
\animals\koala.jpg      C:\Users\bob\Pictures\animals\koala.jpg  
\animals\kangaroo.jpg   X:\BobBackup\photos\animals\kangaroo.jpg
```

After making the necessary files available to the tool, or updating the path map file, you can rerun the tool to complete the import process.

See Also

- [Setting Up the Azure Import-Export Tool](#)
- [Preparing Hard Drives for an Import Job](#)
- [Reviewing Job Status with Copy Log Files](#)
- [Repairing an Export Job](#)
- [Troubleshooting the Azure Import-Export Tool](#)

Repairing an Export Job

1/17/2017 • 4 min to read • [Edit on GitHub](#)

After an export job has completed, you can run the Microsoft Azure Import/Export tool on premise to:

1. Download any files that the Azure Import/Export service was unable to export.
2. Validate that the files on the drive were correctly exported.

You must have connectivity to Azure Storage to use this functionality.

The command for repairing an import job is **RepairExport**. The following parameters can be specified:

PARAMETER	DESCRIPTION
/r:	Required. Path to the repair file, which tracks the progress of the repair, and allows you to resume an interrupted repair. Each drive must have one and only one repair file. When you start a repair for a given drive, you will pass in the path to a repair file which does not yet exist. To resume an interrupted repair, you should pass in the name of an existing repair file. The repair file corresponding to the target drive must always be specified.
/logdir:	Optional. The log directory. Verbose log files will be written to this directory. If no log directory is specified, the current directory will be used as the log directory.
/d:	Required. The directory to validate and repair. This is usually the root directory of the export drive, but could also be a network file share containing a copy of the exported files.
/blk:	Optional. You should specify the BitLocker key if you want the tool to unlock an encrypted where the exported files are stored.
/sn:	Required. The name of the storage account for the export job.
/sk:	Required if and only if a container SAS is not specified. The account key for the storage account for the export job.
/csas:	Required if and only if the storage account key is not specified. The container SAS for accessing the blobs associated with the export job.
/CopyLogFile:	Required. The path to the drive copy log file. The file is generated by the Windows Azure Import/Export service and can be downloaded from the blob storage associated with the job. The copy log file contains information about failed blobs or files which are to be repaired.

PARAMETER	DESCRIPTION
/ManifestFile:	<p>Optional. The path to the export drive's manifest file. This file is generated by the Windows Azure Import/Export service and stored on the export drive, and optionally in a blob in the storage account associated with the job.</p> <p>The content of the files on the export drive will be verified with the MD5 hashes contained in this file. Any files that are determined to be corrupted will be downloaded and rewritten to the target directories.</p>

Using RepairExport Mode to Correct Failed Exports

You can use the Azure Import/Export tool to download files that failed to export. The copy log file will contain a list of files that failed to export.

The causes of export failures include the following possibilities:

- Damaged drives
- The storage account key changed during the transfer process

To run the tool in **RepairExport** mode, you first need to connect the drive containing the exported files to your computer. Next, run the Azure Import/Export tool, specifying the path to that drive with the **/d** parameter. You also need to specify the path to the drive's copy log file that you downloaded. The following command line example below runs the tool to repair any files that failed to export:

```
WAImportExport.exe RepairExport /r:C:\WAImportExport\9WM35C3U.rep /d:G:\ /sn:bobmediaaccount
/sk:VkGbrUqBWLYJ6zg1m29VOTrxpBgdN0lp+kp0C9MEdx3GEIxmbw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ==
/CopyLogFile:C:\WAImportExport\9WM35C3U.log
```

The following is an example of a copy log file that shows that one block in the blob failed to export:

```
<?xml version="1.0" encoding="utf-8"?>
<DriveLog>
  <DriveId>9WM35C2V</DriveId>
  <Blob Status="CompletedWithErrors">
    <BlobPath>pictures/wild/desert.jpg</BlobPath>
    <FilePath>\pictures\wild\desert.jpg</FilePath>
    <LastModified>2012-09-18T23:47:08Z</LastModified>
    <Length>163840</Length>
    <BlockList>
      <Block Offset="65536" Length="65536" Id="AQAAAA==" Status="Failed" />
    </BlockList>
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>
```

The copy log file indicates that a failure occurred while the Windows Azure Import/Export service was downloading one of the blob's blocks to the file on the export drive. The other components of the file downloaded successfully, and the file length was correctly set. In this case, the tool will open the file on the drive, download the block from the storage account, and write it to the file range starting from offset 65536 with length 65536.

Using RepairExport to Validate Drive Contents

You can also use Azure Import/Export with the **RepairExport** option to validate the contents on the drive are

correct. The manifest file on each export drive contains MD5s for the contents of the drive.

The Azure Import/Export service can also save the manifest files to a storage account during the export process. The location of the manifest files is available via the [Get Job](#) operation when the job has completed. See [Import-Export Service Manifest File Format](#) for more information about the format of a drive manifest file.

The following example shows how to run the Azure Import/Export tool with the **/ManifestFile** and **/CopyLogFile** parameters:

```
WAImportExport.exe RepairExport /r:C:\WAImportExport\9WM35C3U.rep /d:G:\ /sn:bobmediaaccount  
/sk:VkGbrUqBWLJ6zg1m29V0TrxpBgdN0lp+kp0C9MEdx3GELxmBw4hK94f7KysbbeKLDksg7VoN1W/a5UuM2zNgQ==  
/CopyLogFile:C:\WAImportExport\9WM35C3U.log /ManifestFile:G:\9WM35C3U.manifest
```

The following is an example of a manifest file:

```
<?xml version="1.0" encoding="utf-8"?>  
<DriveManifest Version="2011-10-01">  
  <Drive>  
    <DriveId>9WM35C3U</DriveId>  
    <ClientCreator>Windows Azure Import/Export service</ClientCreator>  
    <BlobList>  
      <Blob>  
        <BlobPath>pictures/city/redmond.jpg</BlobPath>  
        <FilePath>\pictures\city\redmond.jpg</FilePath>  
        <Length>15360</Length>  
        <PageRangeList>  
          <PageRange Offset="0" Length="3584" Hash="72FC55ED9AFDD40A0C8D5C4193208416" />  
          <PageRange Offset="3584" Length="3584" Hash="68B28A561B73D1DA769D4C24AA427DB8" />  
          <PageRange Offset="7168" Length="512" Hash="F521DF2F50C46BC5F9EA9FB787A23EED" />  
        </PageRangeList>  
        <PropertiesPath  
          Hash="E72A22EA959566066AD89E3B49020C0A">\pictures\city\redmond.jpg.properties</PropertiesPath>  
      </Blob>  
      <Blob>  
        <BlobPath>pictures/wild/canyon.jpg</BlobPath>  
        <FilePath>\pictures\wild\canyon.jpg</FilePath>  
        <Length>10884</Length>  
        <BlockList>  
          <Block Offset="0" Length="2721" Id="AAAAAA==" Hash="263DC9C4B99C2177769C5EBE04787037" />  
          <Block Offset="2721" Length="2721" Id="AQAAAA==" Hash="0C52BAE2CC20EFEC15CC1E3045517AA6" />  
          <Block Offset="5442" Length="2721" Id="AgAAAA==" Hash="73D1CB62CB426230C34C9F57B7148F10" />  
          <Block Offset="8163" Length="2721" Id="AwAAAA==" Hash="11210E665C5F8E7E4F136D053B243E6A" />  
        </BlockList>  
        <PropertiesPath  
          Hash="81D7F81B2C29F10D6E123D386C3A4D5A">\pictures\wild\canyon.jpg.properties</PropertiesPath>  
      </Blob>  
    </BlobList>  
  </Drive>  
</DriveManifest>
```

After finishing the repair process, the tool will read through each file referenced in the manifest file and verify the file's integrity with the MD5 hashes. For the manifest above, it will go through the following components.

G:\pictures\city\redmond.jpg, offset 0, length 3584

G:\pictures\city\redmond.jpg, offset 3584, length 3584

G:\pictures\city\redmond.jpg, offset 7168, length 3584

G:\pictures\city\redmond.jpg.properties

G:\pictures\wild\canyon.jpg, offset 0, length 2721

G:\pictures\wild\canyon.jpg, offset 2721, length 2721

G:\pictures\wild\canyon.jpg, offset 5442, length 2721

G:\pictures\wild\canyon.jpg, offset 8163, length 2721

G:\pictures\wild\canyon.jpg.properties

Any component failing the verification will be downloaded by the tool and rewritten to the same file on the drive.

See Also

[Setting Up the Azure Import-Export Tool](#)

[Preparing Hard Drives for an Import Job](#)

[Reviewing Job Status with Copy Log Files](#)

[Repairing an Import Job](#)

[Troubleshooting the Azure Import-Export Tool](#)

Troubleshooting the Azure Import-Export Tool

1/17/2017 • 1 min to read • [Edit on GitHub](#)

The Microsoft Azure Import/Export tool returns error messages if it runs into issues. This topic lists some common issues that users may run into.

A copy session fails, what I should do?

When a copy session fails, there are two options:

If the error is retryable, for example if the network share was offline for a short period and now is back online, you can resume the copy session. If the error is not retryable, for example if you specified the wrong source file directory in the command line parameters, you need to abort the copy session. See [Preparing Hard Drives for an Import Job](#) for more information about resuming and aborting copy sessions.

I can't resume or abort a copy session.

If the copy session is the first copy session for a drive, then the error message should state: "The first copy session cannot be resumed or aborted." In this case, you can delete the old journal file and rerun the command.

If a copy session is not the first one for a drive, it can always be resumed or aborted.

I lost the journal file, can I still create the job?

The journal file for a drive contains the complete information of copying data to this drive, and it is needed to add more files to the drive and will be used to create an import job. If the journal file is lost, you will have to redo all the copy sessions for the drive.

See Also

[Setting Up the Azure Import-Export Tool](#)

[Preparing Hard Drives for an Import Job](#)

[Reviewing Job Status with Copy Log Files](#)

[Repairing an Import Job](#)

[Repairing an Export Job](#)

Import-Export Service Manifest File Format

1/17/2017 • 7 min to read • [Edit on GitHub](#)

The drive manifest file describes the mapping between blobs in Azure Blob storage and files on drive comprising an import or export job. For an import operation, the manifest file is created as a part of the drive preparation process, and is stored on the drive before the drive is sent to the Azure data center. During an export operation, the manifest is created and stored on the drive by the Azure Import/Export service.

For both import and export jobs, the drive manifest file is stored on the import or export drive; it is not transmitted to the service via any API operation.

The following describes the general format of a drive manifest file:

```

<?xml version="1.0" encoding="UTF-8"?>
<DriveManifest Version="2014-11-01">
  <Drive>
    <DriveId>drive-id</DriveId>
    import-export-credential

    <!-- First Blob List -->
    <BlobList>
      <!-- Global properties and metadata that applies to all blobs -->
      [<MetadataPath Hash="md5-hash">global-metadata-file-path</MetadataPath>]
      [<PropertiesPath
        Hash="md5-hash">global-properties-file-path</PropertiesPath>]

      <!-- First Blob -->
      <Blob>
        <BlobPath>blob-path-relative-to-account</BlobPath>
        <FilePath>file-path-relative-to-transfer-disk</FilePath>
        [<ClientData>client-data</ClientData>]
        [<Snapshot>snapshot</Snapshot>]
        <Length>content-length</Length>
        [<ImportDisposition>import-disposition</ImportDisposition>]
        page-range-list-or-block-list
        [<MetadataPath Hash="md5-hash">metadata-file-path</MetadataPath>]
        [<PropertiesPath Hash="md5-hash">properties-file-path</PropertiesPath>]
      </Blob>

      <!-- Second Blob -->
      <Blob>
        . . .
      </Blob>
    </BlobList>

    <!-- Second Blob List -->
    <BlobList>
      . . .
    </BlobList>
  </Drive>
</DriveManifest>

import-export-credential ::=

<StorageAccountKey>storage-account-key</StorageAccountKey> | <ContainerSas>container-sas</ContainerSas>

page-range-list-or-block-list ::=
  page-range-list | block-list

page-range-list ::=
  <PageRangeList>
    [<PageRange Offset="page-range-offset" Length="page-range-length"
      Hash="md5-hash"/>]
    [<PageRange Offset="page-range-offset" Length="page-range-length"
      Hash="md5-hash"/>]
  </PageRangeList>

block-list ::=
  <BlockList>
    [<Block Offset="block-offset" Length="block-length" [Id="block-id"]
      Hash="md5-hash"/>]
    [<Block Offset="block-offset" Length="block-length" [Id="block-id"]
      Hash="md5-hash"/>]
  </BlockList>

```

The data elements and attributes of the drive manifest XML format are specified in the following table.

XML ELEMENT	TYPE	DESCRIPTION
<code>DriveManifest</code>	Root element	The root element of the manifest file. All other elements in the file are beneath this element.
<code>version</code>	Attribute, String	The version of the manifest file.
<code>Drive</code>	Nested XML element	Contains the manifest for each drive.
<code>DriveId</code>	String	The unique drive identifier for the drive. The drive identifier is found by querying the drive for its serial number. The drive serial number is usually printed on the outside of the drive as well. The <code>DriveID</code> element must appear before any <code>BlobList</code> element in the manifest file.
<code>StorageAccountKey</code>	String	Required for import jobs if and only if <code>ContainerSas</code> is not specified. The account key for the Azure storage account associated with the job. This element is omitted from the manifest for an export operation.
<code>ContainerSas</code>	String	Required for import jobs if and only if <code>StorageAccountKey</code> is not specified. The container SAS for accessing the blobs associated with the job. See Put Job for its format. This element is omitted from the manifest for an export operation.
<code>ClientCreator</code>	String	Specifies the client which created the XML file. This value is not interpreted by the Import/Export service.
<code>BlobList</code>	Nested XML element	Contains a list of blobs that are part of the import or export job. Each blob in a blob list shares the same metadata and properties.
<code>BlobList/MetadataPath</code>	String	Optional. Specifies the relative path of a file on the disk that contains the default metadata that will be set on blobs in the blob list for an import operation. This metadata can be optionally overridden on a blob-by-blob basis. This element is omitted from the manifest for an export operation.
<code>BlobList/MetadataPath/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash value for the metadata file.

XML ELEMENT	TYPE	DESCRIPTION
<code>BlobList/PropertiesPath</code>	String	<p>Optional. Specifies the relative path of a file on the disk that contains the default properties that will be set on blobs in the blob list for an import operation. These properties can be optionally overridden on a blob-by-blob basis.</p> <p>This element is omitted from the manifest for an export operation.</p>
<code>BlobList/PropertiesPath/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash value for the properties file.
<code>Blob</code>	Nested XML element	Contains information about each blob in each blob list.
<code>Blob/BlobPath</code>	String	The relative URI to the blob, beginning with the container name. If the blob is in root container, it must begin with <code>\$root</code> .
<code>Blob/FilePath</code>	String	<p>Specifies the relative path to the file on the drive. For export jobs, the blob path will be used for the file path if possible; e.g., <code>pictures/bob/wild/desert.jpg</code> will be exported to <code>\pictures\bob\wild\desert.jpg</code>. However, due to the limitations of NTFS names, a blob may be exported to a file with a path that doesn't resemble the blob path.</p>
<code>Blob/ClientData</code>	String	Optional. Contains comments from the customer. This value is not interpreted by the Import/Export service.
<code>Blob/Snapshot</code>	DateTime	Optional for export jobs. Specifies the snapshot identifier for an exported blob snapshot.
<code>Blob/Length</code>	Integer	Specifies the total length of the blob in bytes. The value may be up to 200 GB for a block blob and up to 1 TB for a page blob. For a page blob, this value must be a multiple of 512.

XML ELEMENT	TYPE	DESCRIPTION
<code>Blob/ImportDisposition</code>	String	<p>Optional for import jobs, omitted for export jobs. This specifies how the Import/Export service should handle the case for an import job where a blob with the same name already exists. If this value is omitted from the import manifest, the default value is <code>rename</code>.</p> <p>The values for this element include:</p> <ul style="list-style-type: none"> - <code>no-overwrite</code>: If a destination blob is already present with the same name, the import operation will skip importing this file. - <code>overwrite</code>: Any existing destination blob is overwritten completely by the newly imported file. - <code>rename</code>: The new blob will be uploaded with a modified name. <p>The renaming rule is as follows:</p> <ul style="list-style-type: none"> - If the blob name doesn't contain a dot, a new name is generated by appending <code>(2)</code> to the original blob name; if this new name also conflicts with an existing blob name, then <code>(3)</code> is appended in place of <code>(2)</code>; and so on. - If the blob name contains a dot, the portion following the last dot is considered the extension name. Similar to the above procedure, <code>(2)</code> is inserted before the last dot to generate a new name; if the new name still conflicts with an existing blob name, then the service tries <code>(3)</code>, <code>(4)</code>, and so on, until a non-conflicting name is found. <p>Some examples:</p> <p>The blob <code>BlobNameWithoutDot</code> will be renamed to:</p> <pre><code>BlobNameWithoutDot (2) // if BlobNameWithoutDot exists</code></pre> <pre><code>BlobNameWithoutDot (3) // if both BlobNameWithoutDot and BlobNameWithoutDot (2) exist</code></pre> <p>The blob <code>Seattle.jpg</code> will be renamed to:</p> <pre><code>Seattle (2).jpg // if Seattle.jpg exists</code></pre> <pre><code>Seattle (3).jpg // if both Seattle.jpg and Seattle (2).jpg exist</code></pre>

XML ELEMENT	TYPE	DESCRIPTION
<code>PageRangeList</code>	Nested XML element	<p>Required for a page blob.</p> <p>For an import operation, specifies a list of byte ranges of a file to be imported. Each page range is described by an offset and length in the source file that describes the page range, together with an MD5 hash of the region. The <code>Hash</code> attribute of a page range is required. The service will validate that the hash of the data in the blob matches the computed MD5 hash from the page range. Any number of page ranges may be used to describe a file for an import, with the total size up to 1 TB. All page ranges must be ordered by offset and no overlaps are allowed.</p> <p>For an export operation, specifies a set of byte ranges of a blob that have been exported to the drive.</p> <p>The page ranges together may cover only sub-ranges of a blob or file. The remaining part of the file not covered by any page range is expected and its content can be undefined.</p>
<code>PageRange</code>	XML element	Represents a page range.
<code>PageRange/@Offset</code>	Attribute, Integer	Specifies the offset in the transfer file and the blob where the specified page range begins. This value must be a multiple of 512.
<code>PageRange/@Length</code>	Attribute, Integer	Specifies the length of the page range. This value must be a multiple of 512 and no more than 4 MB.
<code>PageRange/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash value for the page range.

XML ELEMENT	TYPE	DESCRIPTION
<code>blockList</code>	Nested XML element	<p>Required for a block blob with named blocks.</p> <p>For an import operation, the block list specifies a set of blocks that will be imported into Azure Storage. For an export operation, the block list specifies where each block has been stored in the file on the export disk. Each block is described by an offset in the file and a block length; each block is furthermore named by a block ID attribute, and contains an MD5 hash for the block. Up to 50,000 blocks may be used to describe a blob. All blocks must be ordered by offset, and together should cover the complete range of the file, <i>i.e.</i>, there must be no gap between blocks. If the blob is no more than 64 MB, the block IDs for each block must be either all absent or all present. Block IDs are required to be Base64-encoded strings. See Put Block for further requirements for block IDs.</p>
<code>block</code>	XML element	Represents a block.
<code>block/@Offset</code>	Attribute, Integer	Specifies the offset where the specified block begins.
<code>block/@Length</code>	Attribute, Integer	Specifies the number of bytes in the block; this value must be no more than 4MB.
<code>block/@Id</code>	Attribute, String	Specifies a string representing the block ID for the block.
<code>block/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash of the block.
<code>blob/MetadataPath</code>	String	Optional. Specifies the relative path of a metadata file. During an import, the metadata is set on the destination blob. During an export operation, the blob's metadata is stored in the metadata file on the drive.
<code>blob/MetadataPath/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash of the blob's metadata file.
<code>blob/PropertiesPath</code>	String	Optional. Specifies the relative path of a properties file. During an import, the properties are set on the destination blob. During an export operation, the blob properties are stored in the properties file on the drive.

XML ELEMENT	TYPE	DESCRIPTION
<code>Blob/PropertiesPath/@Hash</code>	Attribute, String	Specifies the Base16-encoded MD5 hash of the blob's properties file.

See Also

[Storage Import/Export REST](#)

Import-Export Service Metadata and Properties File Format

1/17/2017 • 1 min to read • [Edit on GitHub](#)

You can specify metadata and properties for one or more blobs as part of an import job or an export job. To set metadata or properties for blobs being created as part of an import job, you provide a metadata or properties file on the hard drive containing the data to be imported. For an export job, metadata and properties are written to a metadata or properties file that is included on the hard drive returned to you.

Metadata File Format

The format of a metadata file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Metadata>
[<metadata-name-1>metadata-value-1</metadata-name-1>]
[<metadata-name-2>metadata-value-2</metadata-name-2>]
...
</Metadata>
```

XML ELEMENT	TYPE	DESCRIPTION
<code><Metadata></code>	Root element	The root element of the metadata file.
<code>[<metadata-name></code>	String	Optional. The XML element specifies the name of the metadata for the blob, and its value specifies the value of the metadata setting.

Properties File Format

The format of a properties file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Properties>
[<Last-Modified>date-time-value</Last-Modified>]
[<Etag>etag</Etag>]
[<Content-Length>size-in-bytes</Content-Length>]
[<Content-Type>content-type</Content-Type>]
[<Content-MD5>content-md5</Content-MD5>]
[<Content-Encoding>content-encoding</Content-Encoding>]
[<Content-Language>content-language</Content-Language>]
[<Cache-Control>cache-control</Cache-Control>]
</Properties>
```

XML ELEMENT	TYPE	DESCRIPTION
<code><Properties></code>	Root element	The root element of the properties file.
<code>[<Last-Modified></code>	String	Optional. The last-modified time for the blob. For export jobs only.

XML ELEMENT	TYPE	DESCRIPTION
<code>Etag</code>	String	Optional. The blob's ETag value. For export jobs only.
<code>Content-Length</code>	String	Optional. The size of the blob in bytes. For export jobs only.
<code>Content-Type</code>	String	Optional. The content type of the blob.
<code>Content-MD5</code>	String	Optional. The blob's MD5 hash.
<code>Content-Encoding</code>	String	Optional. The blob's content encoding.
<code>Content-Language</code>	String	Optional. The blob's content language.
<code>Cache-Control</code>	String	Optional. The cache control string for the blob.

See [Set Blob Properties](#), [Set Blob Metadata](#), and [Setting and Retrieving Properties and Metadata for Blob Resources](#) for detailed rules about setting blob metadata and properties.

Import-Export Service Log File Format

1/17/2017 • 10 min to read • [Edit on GitHub](#)

When the Microsoft Azure Import/Export service performs an action on a drive as part of an import job or an export job, logs are written to block blobs in the storage account associated with that job.

There are two logs that may be written by the Import/Export service:

- The error log is always generated in the event of an error.
- The verbose log is not enabled by default, but may be enabled by setting the `EnableVerboseLog` property on a [Put Job](#) or [Update Job Properties](#) operation.

Log File Location

The logs are written to block blobs in the container or virtual directory specified by the `ImportExportStatesPath` setting, which you can set on a [Put Job](#) operation. The location to which the logs are written depends on how authentication is specified for the job, together with the value specified for `ImportExportStatesPath`. Authentication for the job may be specified via a storage account key, or a container SAS (shared access signature).

The name of the container or virtual directory may either be the default name of `waimportexport`, or another container or virtual directory name that you specify.

The table below shows the possible options:

AUTHENTICATION METHOD	VALUE OF <code>IMPORTEXPORTSTATESPATH</code> ELEMENT	LOCATION OF LOG BLOBS
Storage account key	Default value	A container named <code>waimportexport</code> , which is the default container. For example: <code>https://myaccount.blob.core.windows.net/waimportexport</code>
Storage account key	User-specified value	A container named by the user. For example: <code>https://myaccount.blob.core.windows.net/mylogcontainer</code>
Container SAS	Default value	A virtual directory named <code>waimportexport</code> , which is the default name, beneath the container specified in the SAS. For example, if the SAS specified for the job is <code>https://myaccount.blob.core.windows.net/mylogcontainer?sv=2012-02-12&se=2015-05-22T06%3A54%3A55Z&sr=c&sp=w&sig=sigvalue</code> , then the log location would be <code>https://myaccount.blob.core.windows.net/mylogcontainer/waimportexport</code>
Container SAS	User-specified value	A virtual directory named by the user, beneath the container specified in the SAS. For example, if the SAS specified for the job is <code>https://myaccount.blob.core.windows.net/mylogcontainer?sv=2012-02-12&se=2015-05-22T06%3A54%3A55Z&sr=c&sp=w&sig=sigvalue</code> , and the specified virtual directory is named <code>mylogblobs</code> , then the log location would be <code>https://myaccount.blob.core.windows.net/mylogcontainer/mylogblobs/waimportexport</code> .

You can retrieve the URL for the error and verbose logs by calling the [Get Job](#) operation. The logs are available after processing of the drive is complete.

Log File Format

The format for both logs is the same: a blob containing XML descriptions of the events that occurred while copying blobs between the hard drive and the customer's account.

The verbose log contains complete information about the status of the copy operation for every blob (for an import job) or file (for an export job), whereas the error log contains only the information for blobs or files that encountered errors during the import or export job.

The verbose log format is shown below. The error log has the same structure, but filters out successful operations.

```

<DriveLog Version="2014-11-01">
  <DriveId>drive-id</DriveId>
  [<Blob Status="blob-status">
    <BlobPath>blob-path</BlobPath>
    <FilePath>file-path</FilePath>
    [<Snapshot>snapshot</Snapshot>]
    <Length>length</Length>
    [<LastModified>last-modified</LastModified>]
    [<ImportDisposition Status="import-disposition-status">import-disposition</ImportDisposition>]
    [page-range-list-or-block-list]
    [metadata-status]
    [properties-status]
  </Blob>
  [<Blob>
    ...
  </Blob>]
  <Status>drive-status</Status>
</DriveLog>

page-range-list-or-block-list ::= 
  page-range-list | block-list

page-range-list ::= 
<PageRangeList>
  [<PageRange Offset="page-range-offset" Length="page-range-length"
    [Hash="md5-hash"] Status="page-range-status"/>]
  [<PageRange Offset="page-range-offset" Length="page-range-length"
    [Hash="md5-hash"] Status="page-range-status"/>]
</PageRangeList>

block-list ::= 
<BlockList>
  [<Block Offset="block-offset" Length="block-length" [Id="block-id"]
    [Hash="md5-hash"] Status="block-status"/>]
  [<Block Offset="block-offset" Length="block-length" [Id="block-id"]
    [Hash="md5-hash"] Status="block-status"/>]
</BlockList>

metadata-status ::= 
<Metadata Status="metadata-status">
  [<GlobalPath Hash="md5-hash">global-metadata-file-path</GlobalPath>]
  [<Path Hash="md5-hash">metadata-file-path</Path>]
</Metadata>

properties-status ::= 
<Properties Status="properties-status">
  [<GlobalPath Hash="md5-hash">global-properties-file-path</GlobalPath>]
  [<Path Hash="md5-hash">properties-file-path</Path>]
</Properties>

```

The following table describes the elements of the log file.

XML ELEMENT	TYPE	DESCRIPTION
<code>DriveLog</code>	XML Element	Represents a drive log.
<code>Version</code>	Attribute, String	The version of the log format.
<code>DriveId</code>	String	The drive's hardware serial number.
<code>status</code>	String	Status of the drive processing. See the Drive Status Codes table below for more information.
<code>blob</code>	Nested XML element	Represents a blob.
<code>Blob/BlobPath</code>	String	The URI of the blob.
<code>Blob/FilePath</code>	String	The relative path to the file on the drive.
<code>Blob/Snapshot</code>	DateTime	The snapshot version of the blob, for an export job only.
<code>Blob/Length</code>	Integer	The total length of the blob in bytes.
<code>Blob/LastModified</code>	DateTime	The date/time that the blob was last modified, for an export job only.

XML ELEMENT	TYPE	DESCRIPTION
<code>blob/importDisposition</code>	String	The import disposition of the blob, for an import job only.
<code>Blob/importDisposition/@Status</code>	Attribute, String	The status of the import disposition.
<code>PageRangeList</code>	Nested XML element	Represents a list of page ranges for a page blob.
<code>PageRange</code>	XML element	Represents a page range.
<code>PageRange/@Offset</code>	Attribute, Integer	Starting offset of the page range in the blob.
<code>PageRange/@Length</code>	Attribute, Integer	Length in bytes of the page range.
<code>PageRange/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the page range.
<code>PageRange/@Status</code>	Attribute, String	Status of processing the page range.
<code>BlockList</code>	Nested XML element	Represents a list of blocks for a block blob.
<code>Block</code>	XML element	Represents a block.
<code>Block/@Offset</code>	Attribute, Integer	Starting offset of the block in the blob.
<code>Block/@Length</code>	Attribute, Integer	Length in bytes of the block.
<code>Block/@Id</code>	Attribute, String	The block ID.
<code>Block/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the block.
<code>Block/@Status</code>	Attribute, String	Status of processing the block.
<code>Metadata</code>	Nested XML element	Represents the blob's metadata.
<code>Metadata/@Status</code>	Attribute, String	Status of processing of the blob metadata.
<code>Metadata/GlobalPath</code>	String	Relative path to the global metadata file.
<code>Metadata/GlobalPath/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the global metadata file.
<code>Metadata/Path</code>	String	Relative path to the metadata file.
<code>Metadata/Path/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the metadata file.
<code>Properties</code>	Nested XML element	Represents the blob properties.
<code>Properties/@Status</code>	Attribute, String	Status of processing the blob properties, e.g. file not found, completed.
<code>Properties/GlobalPath</code>	String	Relative path to the global properties file.
<code>Properties/GlobalPath/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the global properties file.
<code>Properties/Path</code>	String	Relative path to the properties file.
<code>Properties/Path/@Hash</code>	Attribute, String	Base16-encoded MD5 hash of the properties file.
<code>Blob/Status</code>	String	Status of processing the blob.

Drive Status Codes

The following table lists the status codes for processing a drive.

STATUS CODE	DESCRIPTION
Completed	The drive has finished processing without any errors.
CompletedWithWarning	The drive has finished processing with warnings in one or more blobs per the import dispositions specified for the blobs.
CompletedWithErrors	The drive has finished with errors in one or more blobs or chunks.
DiskNotFound	No disk is found on the drive.
VolumeNotNtfs	The first data volume on the disk is not in NTFS format.
DiskOperationFailed	An unknown failure occurred when performing operations on the drive.
BitLockerVolumeNotFound	No BitLocker encryptable volume is found.
BitLockerNotActivated	BitLocker is not enabled on the volume.
BitLockerProtectorNotFound	The numerical password key protector does not exist on the volume.
BitLockerKeyInvalid	The numerical password provided cannot unlock the volume.
BitLockerUnlockVolumeFailed	Unknown failure has happened when trying to unlock the volume.
BitLockerFailed	An unknown failure occurred while performing BitLocker operations.
ManifestNameInvalid	The manifest file name is invalid.
ManifestNameTooLong	The manifest file name is too long.
ManifestNotFound	The manifest file is not found.
ManifestAccessDenied	Access to the manifest file is denied.
ManifestCorrupted	The manifest file is corrupted (the content does not match its hash).
ManifestFormatInvalid	The manifest content does not conform to the required format.
ManifestDriveIdMismatch	The drive ID in the manifest file does not match the one read from the drive.
ReadManifestFailed	A disk I/O failure occurred while reading from the manifest.
BlobListFormatInvalid	The export blob list blob does not conform to the required format.
BlobRequestForbidden	Access to the blobs in the storage account is forbidden. This might be due to invalid storage account key or container SAS.
InternalError	An internal error occurred while processing the drive.

Blob Status Codes

The following table lists the status codes for processing a blob.

STATUS CODE	DESCRIPTION
Completed	The blob has finished processing without errors.
CompletedWithErrors	The blob has finished processing with errors in one or more page ranges or blocks, metadata, or properties.
FileNameInvalid	The file name is invalid.
FileNameTooLong	The file name is too long.
FileNotFound	The file is not found.
FileAccessDenied	Access to the file is denied.

STATUS CODE	DESCRIPTION
<code>BlobRequestFailed</code>	The Blob service request to access the blob has failed.
<code>BlobRequestForbidden</code>	The Blob service request to access the blob is forbidden. This might be due to invalid storage account key or container SAS.
<code>RenameFailed</code>	Failed to rename the blob (for an import job) or the file (for an export job).
<code>BlobUnexpectedChange</code>	An unexpected change has occurred with the blob (for an export job).
<code>LeasePresent</code>	There is a lease present on the blob.
<code>IOFailed</code>	A disk or network I/O failure occurred while processing the blob.
<code>Failed</code>	An unknown failure occurred while processing the blob.

Import Disposition Status Codes

The following table lists the status codes for resolving an import disposition.

STATUS CODE	DESCRIPTION
<code>Created</code>	The blob has been created.
<code>Renamed</code>	The blob has been renamed per rename import disposition. The <code>Blob/BlobPath</code> element contains the URI for the renamed blob.
<code>Skipped</code>	The blob has been skipped per <code>no-overwrite</code> import disposition.
<code>Overwritten</code>	The blob has overwritten an existing blob per <code>overwrite</code> import disposition.
<code>Canceled</code>	A prior failure has stopped further processing of the import disposition.

Page Range/Block Status Codes

The following table lists the status codes for processing a page range or a block.

STATUS CODE	DESCRIPTION
<code>Completed</code>	The page range or block has finished processing without any errors.
<code>Committed</code>	The block has been committed, but not in the full block list because other blocks have failed, or put full block list itself has failed.
<code>Uncommitted</code>	The block is uploaded but not committed.
<code>Corrupted</code>	The page range or block is corrupted (the content does not match its hash).
<code>FileUnexpectedEnd</code>	An unexpected end of file has been encountered.
<code>BlobUnexpectedEnd</code>	An unexpected end of blob has been encountered.
<code>BlobRequestFailed</code>	The Blob service request to access the page range or block has failed.
<code>IOFailed</code>	A disk or network I/O failure occurred while processing the page range or block.
<code>Failed</code>	An unknown failure occurred while processing the page range or block.
<code>Canceled</code>	A prior failure has stopped further processing of the page range or block.

Metadata Status Codes

The following table lists the status codes for processing blob metadata.

STATUS CODE	DESCRIPTION
<code>Completed</code>	The metadata has finished processing without errors.
<code>fileNameInvalid</code>	The metadata file name is invalid.

STATUS CODE	DESCRIPTION
FileNameTooLong	The metadata file name is too long.
FileNotFoundException	The metadata file is not found.
FileAccessDenied	Access to the metadata file is denied.
Corrupted	The metadata file is corrupted (the content does not match its hash).
XmlReadFailed	The metadata content does not conform to the required format.
XmlWriteFailed	Writing the metadata XML has failed.
BlobRequestFailed	The Blob service request to access the metadata has failed.
IOPFailure	A disk or network I/O failure occurred while processing the metadata.
Failed	An unknown failure occurred while processing the metadata.
Cancelled	A prior failure has stopped further processing of the metadata.

Properties Status Codes

The following table lists the status codes for processing blob properties.

STATUS CODE	DESCRIPTION
Completed	The properties have finished processing without any errors.
FileNameInvalid	The properties file name is invalid.
FileNameTooLong	The properties file name is too long.
FileNotFoundException	The properties file is not found.
FileAccessDenied	Access to the properties file is denied.
Corrupted	The properties file is corrupted (the content does not match its hash).
XmlReadFailed	The properties content does not conform to the required format.
XmlWriteFailed	Writing the properties XML has failed.
BlobRequestFailed	The Blob service request to access the properties has failed.
IOPFailure	A disk or network I/O failure occurred while processing the properties.
Failed	An unknown failure occurred while processing the properties.
Cancelled	A prior failure has stopped further processing of the properties.

Sample Logs

The following is an example of verbose log.

```

<?xml version="1.0" encoding="UTF-8"?>
<DriveLog Version="2014-11-01">
  <DriveId>WD-WMATV123456</DriveId>
  <Blob Status="Completed">
    <BlobPath>pictures\bob\wild\desert.jpg</BlobPath>
    <FilePath>C:\Users\bob\Pictures\wild\desert.jpg</FilePath>
    <Length>98304</Length>
    <ImportDisposition Status="Created">overwrite</ImportDisposition>
    <BlockList>
      <Block Offset="0" Length="65536" Id="AAAAAA==" Hash=" 9C8AE14A55241F98533C4D80D85CDC68" Status="Completed"/>
      <Block Offset="65536" Length="32768" Id="AQAAAA==" Hash=" DF54C531C9B3CA2570FDDDB3BCD0E27D" Status="Completed"/>
    </BlockList>
    <Metadata Status="Completed">
      <GlobalPath Hash=" E34F54B7086BCF4EC1601D056F4C7E37">\Users\bob\Pictures\wild\metadata.xml</GlobalPath>
    </Metadata>
  </Blob>
  <Blob Status="CompletedWithErrors">
    <BlobPath>pictures\bob\animals\koala.jpg</BlobPath>
    <FilePath>C:\Users\bob\Pictures\animals\koala.jpg</FilePath>
    <Length>163840</Length>
    <ImportDisposition Status="Overwritten">overwrite</ImportDisposition>
    <PageRangeList>
      <PageRange Offset="0" Length="65536" Hash="19701B8877418393CB3CB567F53EE225" Status="Completed"/>
      <PageRange Offset="65536" Length="65536" Hash="AA2585F6F6FD01C4AD4256E018240CD4" Status="Corrupted"/>
      <PageRange Offset="131072" Length="4096" Hash="9BA552E1C3EEAFFC91B42B979900A996" Status="Completed"/>
    </PageRangeList>
    <Properties Status="Completed">
      <Path Hash="38D7AE80653F47F63C0222FEE90EC4E7">\Users\bob\Pictures\animals\koala.jpg.properties</Path>
    </Properties>
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>

```

The corresponding error log is shown below.

```

<?xml version="1.0" encoding="UTF-8"?>
<DriveLog Version="2014-11-01">
  <DriveId>WD-WMATV6965824</DriveId>
  <Blob Status="CompletedWithErrors">
    <BlobPath>pictures\bob\animals\koala.jpg</BlobPath>
    <FilePath>C:\Users\bob\Pictures\animals\koala.jpg</FilePath>
    <Length>163840</Length>
    <ImportDisposition Status="Overwritten">overwrite</ImportDisposition>
    <PageRangeList>
      <PageRange Offset="65536" Length="65536" Hash="AA2585F6F6FD01C4AD4256E018240CD4" Status="Corrupted"/>
    </PageRangeList>
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>

```

The follow error log for an import job contains an error about a file not found on the import drive. Note that the status of subsequent components is

Cancelled

```

<?xml version="1.0" encoding="utf-8"?>
<DriveLog Version="2014-11-01">
  <DriveId>9WM35C2V</DriveId>
  <Blob Status="FileNotFound">
    <BlobPath>pictures\animals\koala.jpg</BlobPath>
    <FilePath>\animals\koala.jpg</FilePath>
    <Length>30310</Length>
    <ImportDisposition Status="Cancelled">rename</ImportDisposition>
    <BlockList>
      <Block Offset="0" Length="6062" Id="MD5/cAzn4h7VVSXF696qp5Uaw==" Hash="700CE7E21ED55525977FAF7AAA9E546B" Status="Cancelled" />
      <Block Offset="6062" Length="6062" Id="MD5/PEnGwY0T8LPLNYdfKr7kAg==" Hash="3C49C6C18388F0B3CB35875F2ABEE402" Status="Cancelled" />
      <Block Offset="12124" Length="6062" Id="MD5/FG4WxqfZKuUWZ2nGTU2qVA==" Hash="146E16C6A7D92AE5166769C64D4DA54" Status="Cancelled" />
      <Block Offset="18186" Length="6062" Id="MD5/ZzbNDzr3IRBQENRygeXQ==" Hash="67389B343CEBDC8441404351C9E81E5D" Status="Cancelled" />
      <Block Offset="24248" Length="6062" Id="MD5/ZzbNDzr3IRBQENRygeXQ==" Hash="67389B343CEBDC8441404351C9E81E5D" Status="Cancelled" />
    </BlockList>
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>

```

The following error log for an export job indicates that the blob content has been successfully written to the drive, but that an error occurred while exporting the blob's properties.

```
<?xml version="1.0" encoding="utf-8"?>
<DriveLog Version="2014-11-01">
  <DriveId>9WM35C3U</DriveId>
  <Blob Status="CompletedWithErrors">
    <BlobPath>pictures/wild/canyon.jpg</BlobPath>
    <FilePath>\pictures\wild\canyon.jpg</FilePath>
    <LastModified>2012-09-18T23:47:08Z</LastModified>
    <Length>163840</Length>
    <BlockList />
    <Properties Status="Failed" />
  </Blob>
  <Status>CompletedWithErrors</Status>
</DriveLog>
```

See Also

[Storage Import/Export REST](#)

Using the Azure Import/Export service REST API

1/17/2017 • 1 min to read • [Edit on GitHub](#)

The Microsoft Azure Import/Export service exposes a REST API to enable programmatic control of import/export jobs. You can use the REST API to perform all of the import/export operations that you can perform with the [Azure portal](#). Additionally, you can use the REST API to perform certain granular operations, such as querying the percentage completion of a job, which are not currently available in the classic portal.

See [Using the Microsoft Azure Import/Export service to Transfer Data to Blob Storage](#) for an overview of the Import/Export service and a tutorial that demonstrates how to use the classic portal to create and manage import and export jobs.

Service endpoints

The Azure Import/Export service is a resource provider for Azure Resource Manager and provides a set of REST APIs at the following HTTPS endpoint for managing import/export jobs:

```
https://management.azure.com/subscriptions/<subscription-id>/resourceGroups/<resource-group>/providers/Microsoft.ImportExport/jobs/<job-name>
```

Versioning

Requests to the Import/Export service must specify the `api-version` parameter and set its value to `2016-11-01`.

In this section

[Creating an Import Job](#)

[Creating an Export Job](#)

[Retrieving State Information for a Job](#)

[Enumerating Jobs](#)

[Cancelling and Deleting Jobs](#)

[Backing Up Drive Manifests](#)

[Diagnostics and Error Recovery for Import-Export Jobs](#)

See Also

[Storage Import/Export REST](#)

Creating an Import Job

1/17/2017 • 3 min to read • [Edit on GitHub](#)

Creating an import job for the Microsoft Azure Import/Export service using the REST API involves the following steps:

- Preparing drives with the Azure Import/Export tool.
- Obtaining the location to which to ship the drive.
- Creating the import job.
- Shipping the drives to Microsoft via a supported carrier service.
- Updating the import job with the shipping details.

See [Using the Microsoft Azure Import/Export service to Transfer Data to Blob Storage](#) for an overview of the Import/Export service and a tutorial that demonstrates how to use the [Azure portal](#) to create and manage import and export jobs.

Preparing drives with the Azure Import/Export Tool

The steps to prepare drives for an import job are the same whether you create the job via the portal or via the REST API.

Below is a brief overview of drive preparation. Refer to the [Azure Import-Export Tool Reference](#) for complete instructions. You can download the Azure Import/Export tool [here](#).

Preparing your drive involves:

- Identifying the data to be imported.
- Identifying the destination blobs in Windows Azure Storage.
- Using the Azure Import/Export tool to copy your data to one or more hard drives.

The Azure Import/Export tool will also generate a manifest file for each of the drives as it is prepared. A manifest file contains:

- An enumeration of all the files intended for upload and the mappings from these files to blobs.
- Checksums of the segments of each file.
- Information about the metadata and properties to associate with each blob.
- A listing of the action to take if a blob that is being uploaded has the same name as an existing blob in the container. Possible options are: a) overwrite the blob with the file, b) keep the existing blob and skip uploading the file, c) append a suffix to the name so that it does not conflict with other files.

Obtaining Your Shipping Location

Before creating an import job, you need to obtain a shipping location name and address by calling the [List Locations](#) operation. `List Locations` will return a list of locations and their mailing addresses. You can select a location from the returned list and ship your hard drives to that address. You can also use the `Get Location` operation to obtain the shipping address for a specific location directly.

Follow the steps below to obtain the shipping location:

- Identify the name of the location of your storage account. This value can be found under the **Location** field on the storage account's **Dashboard** in the Azure portal or queried for by using the service management API operation [Get Storage Account Properties](#).
- Retrieve the location that is available to process this storage account by calling the [Get Location](#) operation.
- If the `AlternateLocations` property of the location contains the location itself, then it is okay to use this location. Otherwise, call the [Get Location](#) operation again with one of the alternate locations. The original location might be temporarily closed for maintenance.

Creating the Import Job

To create the import job, call the [Put Job](#) operation. You will need to provide the following information:

- A name for the job.
- The storage account name.
- The shipping location name, obtained from the previous step.
- A job type (Import).
- The return address where the drives should be sent after the import job has completed.
- The list of drives in the job. For each drive, you must include the following information that was obtained during the drive preparation step:
 - The drive Id
 - The BitLocker key
 - The manifest file relative path on the hard drive
 - The Base16 encoded manifest file MD5 hash

Shipping Your Drives

You must ship your drives to the address that you obtained from the previous step, and you must provide the Import/Export service with the tracking number of the package.

NOTE

You must ship your drives via a supported carrier service, which will provide a tracking number for your package.

Updating the Import Job with Your Shipping Information

After you have your tracking number, call the [Update Job Properties](#) operation to update the shipping carrier name, the tracking number for the job, and the carrier account number for return shipping. You can optionally specify the number of drives and the shipping date as well.

See Also

[Using the Import/Export service REST API](#)

Creating an Export Job

1/17/2017 • 3 min to read • [Edit on GitHub](#)

Creating an export job for the Microsoft Azure Import/Export service using the REST API involves the following steps:

- Selecting the blobs to export.
- Obtaining a shipping location.
- Creating the export job.
- Shipping your empty drives to Microsoft via a supported carrier service.
- Updating the export job with the package information.
- Receiving the drives back from Microsoft.

See [Using the Windows Azure Import/Export service to Transfer Data to Blob Storage](#) for an overview of the Import/Export service and a tutorial that demonstrates how to use the [Azure portal](#) to create and manage import and export jobs.

Selecting Blobs to Export

To create an export job, you will need to provide a list of blobs that you want to export from your storage account.

There are a few ways to select blobs to be exported:

- You can use a relative blob path to select a single blob and all of its snapshots.
- You can use a relative blob path to select a single blob excluding its snapshots.
- You can use a relative blob path and a snapshot time to select a single snapshot.
- You can use a blob prefix to select all blobs and snapshots with the given prefix.
- You can export all blobs and snapshots in the storage account.

For more information about specifying blobs to export, see the [Put Job](#) operation.

Obtaining Your Shipping Location

Before creating an export job, you need to obtain a shipping location name and address by calling the [Get Location](#) or [List Locations](#) operation. `List Locations` will return a list of locations and their mailing addresses. You can select a location from the returned list and ship your hard drives to that address. You can also use the `Get Location` operation to obtain the shipping address for a specific location directly.

Follow the steps below to obtain the shipping location:

- Identify the name of the location of your storage account. This value can be found under the **Location** field on the storage account's **Dashboard** in the classic portal or queried for by using the service management API operation [Get Storage Account Properties](#).
- Retrieve the location that are available to process this storage account by calling the `Get Location` operation.
- If the `AlternateLocations` property of the location contains the location itself, then it is okay to use this

location. Otherwise, call the [Get Location](#) operation again with one of the alternate locations. The original location might be temporarily closed for maintenance.

Creating the Export Job

To create the export job, call the [Put Job](#) operation. You will need to provide the following information:

- A name for the job.
- The storage account name.
- The shipping location name, obtained in the previous step.
- A job type (Export).
- The return address where the drives should be sent after the export job has completed.
- The list of blobs (or blob prefixes) to be exported.

Shipping Your Drives

Next, use the Azure Import/Export tool to determine the number of drives you need to send, based on the blobs you have selected to be exported and the drive size. See the [Azure Import-Export Tool Reference](#) for details.

Package the drives in a single package and ship them to the address obtained in the earlier step. Note the tracking number of your package for the next step.

NOTE

You must ship your drives via a supported carrier service, which will provide a tracking number for your package.

Updating the Export Job with Your Package Information

After you have your tracking number, call the [Update Job Properties](#) operation to updated the carrier name and tracking number for the job. You can optionally specify the number of drives, the return address, and the shipping date as well.

Receiving the Package

After your export job has been processed, your drives will be returned to you with your encrypted data. You can retrieve the BitLocker key for each of the drives by calling the [Get Job](#) operation. You can then unlock the drive using the key. The drive manifest file on each drive contains the list of files on the drive, as well as the original blob address for each file.

See Also

[Using the Import/Export service REST API](#)

Retrieving State Information for a Job

1/17/2017 • 6 min to read • [Edit on GitHub](#)

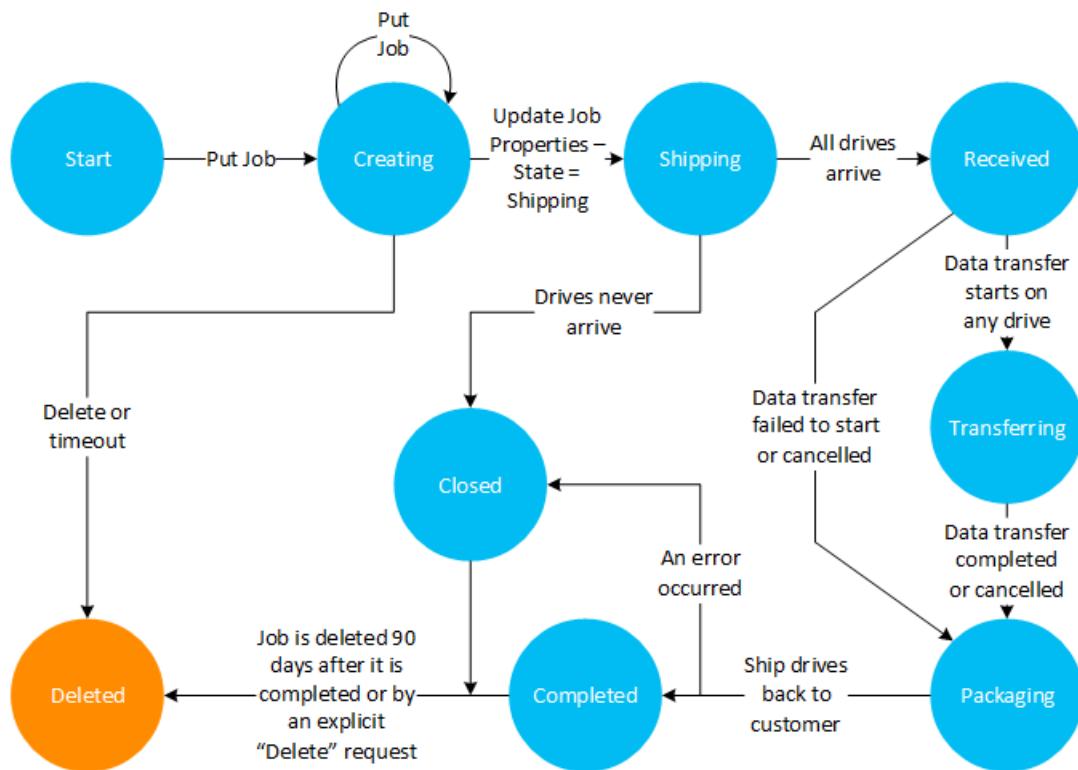
You can call the [Get Job](#) operation to retrieve information about both import and export jobs. The information returned includes:

- The current status of the job.
- The approximate percentage that each job has been completed.
- The current state of each drive.
- URLs for blobs containing error logs and verbose logging information (if enabled).

The following sections explain the information returned by the [Get Job](#) operation.

Job States

The table and the state diagram below describe the states that a job transitions through during its life cycle. The current state of the job can be determined by calling the [Get Job](#) operation.



The following table describes each state that a job may pass through.

JOB STATE	DESCRIPTION
-----------	-------------

JOB STATE	DESCRIPTION
Creating	<p>After you call the Put Job operation, a job is created and its state is set to <code>Creating</code>. While the job is in the <code>Creating</code> state, the Import/Export service assumes the drives have not been shipped to the data center. A job may remain in the <code>Creating</code> state for up to two weeks, after which it is automatically deleted by the service.</p> <p>If you call the Update Job Properties operation while the job is in the <code>Creating</code> state, the job remains in the <code>Creating</code> state, and the timeout interval is reset to two weeks.</p>
Shipping	<p>After you ship your package, you should call the Update Job Properties operation update the state of the job to <code>Shipping</code>. The shipping state can be set only if the <code>DeliveryPackage</code> (postal carrier and tracking number) and the <code>ReturnAddress</code> properties have been set for the job.</p> <p>The job will remain in the <code>Shipping</code> state for up to two weeks. If two weeks have passed and the drives have not been received, the Import/Export service operators will be notified.</p>
Received	<p>After all drives have been received at the data center, the job state will be set to the <code>Received</code> state.</p>
Transferring	<p>After the drives have been received at the data center and at least one drive has begun processing, the job state will be set to the <code>Transferring</code> state. See the Drive States section below for detailed information.</p>
Packaging	<p>After all drives have completed processing, the job will be placed in the <code>Packaging</code> state until the drives are shipped back to the customer.</p>
Completed	<p>After all drives have been shipped back to the customer, if the job has completed without errors, then the job will be set to the <code>Completed</code> state. The job will be automatically deleted after 90 days in the <code>Completed</code> state.</p>
Closed	<p>After all drives have been shipped back to the customer, if there have been any errors during the processing of the job, then the job will be set to the <code>Closed</code> state. The job will be automatically deleted after 90 days in the <code>Closed</code> state.</p>

You can cancel a job only at certain states. A cancelled job skips the data copy step, but otherwise it follows the same state transitions as a job that was not cancelled.

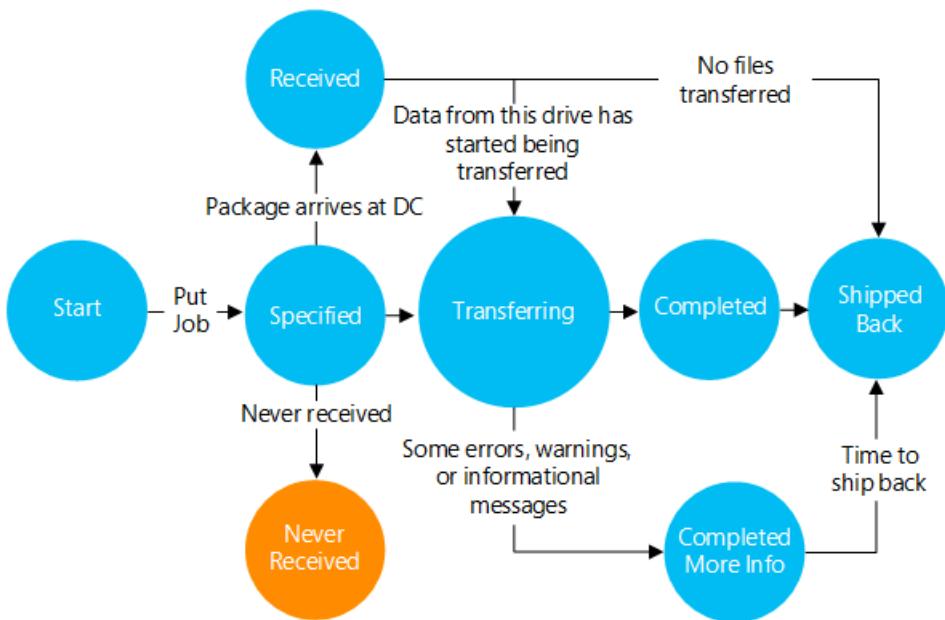
The following table describes errors that can occur for each job state, as well as the effect on the job when an error occurs.

JOB STATE	EVENT	RESOLUTION / NEXT STEPS

JOB STATE	EVENT	RESOLUTION / NEXT STEPS
Creating or Undefined	One or more drives for a job arrived, but the job is not in the Shipping state or there is no record of the job in the service.	<p>The Import/Export service operations team will attempt to contact the customer to create or update the job with necessary information to move the job forward.</p> <p>If the operations team is unable to contact the customer within two weeks, the operations team will attempt to return the drives.</p> <p>In the event that the drives cannot be returned and the customer cannot be reached, the drives will be securely destroyed in 90 days.</p> <p>Note that a job cannot be processed until its state is updated to Shipping.</p>
Shipping	The package for a job has not arrived for over two weeks.	<p>The operations team will notify the customer of the missing package. Based on the customer's response, the operations team will either extend the interval to wait for the package to arrive, or cancel the job.</p> <p>In the event that the customer cannot be contacted or does not respond within 30 days, the operations team will initiate action to move the job from the Shipping state directly to the Closed state.</p>
Completed/Closed	The drives never reached the return address or were damaged in shipment (applies only to an export job).	<p>If the drives do not reach the return address, the customer should first call the Get Job operation or check the job status in the portal to ensure that the drives have been shipped. If the drives have been shipped, then the customer should contact the shipping provider to try and locate the drives.</p> <p>If the drives are damaged during shipment, the customer may want to request another export job, or download the missing blobs.</p>
Transferring/Packaging	Job has an incorrect or missing return address.	<p>The operations team will reach out to the contact person for the job to obtain the correct address.</p> <p>In the event that the customer cannot be reached, the drives will be securely destroyed in 90 days.</p>
Creating / Shipping/ Transferring	A drive that does not appear in the list of drives to be imported is included in the shipping package.	<p>The extra drives will not be processed, and will be returned to the customer when the job is completed.</p>

Drive States

The table and the diagram below describe the life cycle of an individual drive as it transitions through an import or export job. You can retrieve the current drive state by calling the `Get Job` operation and inspecting the `State` element of the `DriveList` property.



The following table describes each state that a drive may pass through.

DRIVE STATE	DESCRIPTION
<code>Specified</code>	For an import job, when the job is created with the <code>Put Job</code> operation, the initial state for a drive is the <code>Specified</code> state. For an export job, since no drive is specified when the job is created, the initial drive state is the <code>Received</code> state.
<code>Received</code>	The drive transitions to the <code>Received</code> state when the Import/Export service operator has processed the drives that were received from the shipping company for an import job. For an export job, the initial drive state is the <code>Received</code> state.
<code>NeverReceived</code>	The drive will move to the <code>NeverReceived</code> state when the package for a job arrives but the package doesn't contain the drive. A drive can also move into this state if it has been two weeks since the service received the shipping information, but the package has not yet arrived at the data center.
<code>Transferring</code>	A drive will move to the <code>Transferring</code> state when the service begins to transfer data from the drive to Windows Azure Storage.
<code>Completed</code>	A drive will move to the <code>Completed</code> state when the service has successfully transferred all the data with no errors.
<code>CompletedMoreInfo</code>	A drive will move to the <code>CompletedMoreInfo</code> state when the service has encountered some issues while copying data either from or to the drive. The information can include errors, warnings, or informational messages about overwriting blobs.

DRIVE STATE	DESCRIPTION
<code>ShippedBack</code>	The drive will move to the <code>ShippedBack</code> state when it has been shipped from the data center back to the return address.

The following table describes the drive failure states and the actions taken for each state.

DRIVE STATE	EVENT	RESOLUTION / NEXT STEP
<code>NeverReceived</code>	A drive that is marked as <code>NeverReceived</code> (because it was not received as part of the job's shipment) arrives in another shipment.	The operations team will move the drive to the <code>Received</code> state.
<code>N/A</code>	A drive that is not part of any job arrives at the data center as part of another job.	The drive will be marked as an extra drive and will be returned to the customer when the job associated with the original package is completed.

Faulted States

When a job or drive fails to progress normally through its expected life cycle, the job or drive will be moved into a `Faulted` state. At that point, the operations team will contact the customer by email or phone. Once the issue is resolved, the faulted job or drive will be taken out of the `Faulted` state and moved into the appropriate state.

See Also

[Using the Import/Export service REST API](#)

Enumerating Jobs

1/17/2017 • 1 min to read • [Edit on GitHub](#)

To enumerate all jobs in a subscription, call the [List Jobs](#) operation. `List Jobs` returns a list of jobs as well as the following attributes:

- The type of job (Import or Export)
- The current job state
- The job's associated storage account

See Also

[Using the Import/Export service REST API](#)

Cancelling and Deleting Jobs

1/17/2017 • 1 min to read • [Edit on GitHub](#)

You can request that a job be cancelled before it is in the `Packaging` state by calling the [Update Job Properties](#) operation and setting the `cancelRequested` element to `true`. The job will be cancelled on a best-effort basis. If drives are in the process of transferring data, data may continue to be transferred even after cancellation has been requested.

A cancelled job will move to the `Completed` state and be kept for 90 days, at which point it will be deleted.

To delete a job, call the [Delete Job](#) operation before the job has shipped (*i.e.*, while the job is in the `Creating` state). You can also delete a job when it is in the `Completed` state. After a job has been deleted, its information and status are no longer accessible via the REST API or the Azure portal.

See Also

[Using the Import/Export service REST API](#)

Backing Up Drive Manifests

1/17/2017 • 1 min to read • [Edit on GitHub](#)

Drive manifests can be automatically backed-up to blobs by setting the `BackupDriveManifest` property to `true` in the [Put Job](#) or [Update Job Properties](#) operations. By default the drive manifests are not backed up. The drive manifest backups are stored as block blobs in a container within the storage account associated with the job. By default, the container name is `waimportexport`, but you can specify a different name in the `DiagnosticsPath` property when calling the `Put Job` or `Update Job Properties` operations. The backup manifest blob are named in the following format: `waies/jobname_driveid_timestamp_manifest.xml`.

You can retrieve the URI of the backup drive manifests for a job by calling the [Get Job](#) operation. The blob URI is returned in the `ManifestUri` property for each drive.

See Also

[Using the Import/Export service REST API](#)

Diagnostics and Error Recovery for Import-Export Jobs

1/17/2017 • 1 min to read • [Edit on GitHub](#)

For each drive processed, the Azure Import/Export service creates an error log in the associated storage account. You can also enable verbose logging by setting the `LogLevel` property to `Verbose` when calling the [Put Job](#) or [Update Job Properties](#) operations.

By default, logs are written to a container named `waimportexport`. You can specify a different name by setting the `DiagnosticsPath` property when calling the [Put Job](#) or [Update Job Properties](#) operations. The logs are stored as block blobs with the following naming convention: `waires/jobname_driveid_timestamp_logtype.xml`.

You can retrieve the URI of the logs for a job by calling the [Get Job](#) operation. The URI for the verbose log is returned in the `VerboseLogUri` property for each drive, while the URI for the error log is returned in the `ErrorLogUri` property.

You can use the logging data to identify the following issues:

Drive Errors

- Errors in accessing or reading the manifest file
- Incorrect BitLocker keys
- Drive read/write errors

Blob Errors

- Incorrect or conflicting blob or names
- Missing files
- Blob not found
- Truncated files (the files on the disk are smaller than specified in the manifest)
- Corrupted file content (for import jobs, detected with an MD5 checksum mismatch)
- Corrupted blob metadata and property files (detected with an MD5 checksum mismatch)
- Incorrect schema for the blob properties and/or metadata files

There may be cases where some parts of an import or export job do not complete successfully, while the overall job still completes. In this case, you can either upload or download the missing pieces of the data over network, or you can create a new job to transfer the data. See the [Azure Import-Export Tool Reference](#) to learn how to repair the data over network.

See Also

[Using the Import/Export service REST API](#)

About Azure storage accounts

1/17/2017 • 9 min to read • [Edit on GitHub](#)

TIP

Try the Microsoft Azure Storage Explorer

[Microsoft Azure Storage Explorer](#) is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Overview

An Azure storage account gives you access to the Azure Blob, Queue, Table, and File services in Azure Storage. Your storage account provides the unique namespace for your Azure Storage data objects. By default, the data in your account is available only to you, the account owner.

There are two types of storage accounts:

- A standard storage account includes Blob, Table, Queue, and File storage.
- A premium storage account currently supports Azure virtual machine disks only. See [Premium Storage: High-performance Storage for Azure Virtual Machine Workloads](#) for an in-depth overview of Premium Storage.

Storage account billing

You are billed for Azure Storage usage based on your storage account. Storage costs are based on four factors: storage capacity, replication scheme, storage transactions, and data egress.

- Storage capacity refers to how much of your storage account allotment you are using to store data. The cost of simply storing your data is determined by how much data you are storing, and how it is replicated.
- Replication determines how many copies of your data are maintained at once, and in what locations.
- Transactions refer to all read and write operations to Azure Storage.
- Data egress refers to data transferred out of an Azure region. When the data in your storage account is accessed by an application that is not running in the same region, whether that application is a cloud service or some other type of application, then you are charged for data egress. (For Azure services, you can take steps to group your data and services in the same data centers to reduce or eliminate data egress charges.)

The [Azure Storage Pricing](#) page provides detailed pricing information for storage capacity, replication, and transactions. The [Data Transfers Pricing Details](#) page provides detailed pricing information for data egress.

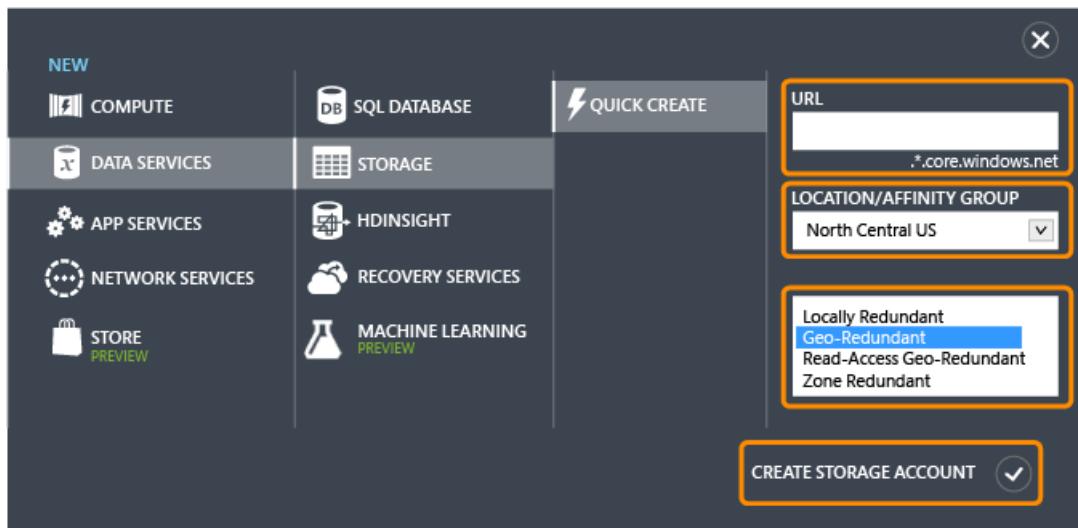
For details about storage account capacity and performance targets, see [Azure Storage Scalability and Performance Targets](#).

NOTE

When you create an Azure virtual machine, a storage account is created for you automatically in the deployment location if you do not already have a storage account in that location. So it's not necessary to follow the steps below to create a storage account for your virtual machine disks. The storage account name will be based on the virtual machine name. See the [Azure Virtual Machines documentation](#) for more details.

Create a storage account

1. Sign in to the [Azure Classic Portal](#).
2. Click **New** in the taskbar at the bottom of the page. Choose **Data Services | Storage**, and then click **Quick Create**.



3. In **URL**, enter a name for your storage account.

NOTE

Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only.

Your storage account name must be unique within Azure. The Azure Classic Portal will indicate if the storage account name you select is already taken.

See [Storage account endpoints](#) below for details about how the storage account name will be used to address your objects in Azure Storage.

4. In **Location/Affinity Group**, select a location for your storage account that is close to you or to your customers. If data in your storage account will be accessed from another Azure service, such as an Azure virtual machine or cloud service, you may want to select an affinity group from the list to group your storage account in the same data center with other Azure services that you are using to improve performance and lower costs.

Note that you must select an affinity group when your storage account is created. You cannot move an existing account to an affinity group. For more information on affinity groups, see [Service co-location with an affinity group](#) below.

IMPORTANT

To determine which locations are available for your subscription, you can call the [List all resource providers](#) operation. To list providers from PowerShell, call `Get-AzureLocation`. From .NET, use the `List` method of the `ProviderOperationsExtensions` class.

Additionally, see [Azure Regions](#) for more information about what services are available in which region.

5. If you have more than one Azure subscription, then the **Subscription** field is displayed. In **Subscription**, enter the Azure subscription that you want to use the storage account with.
6. In **Replication**, select the desired level of replication for your storage account. The recommended replication

option is geo-redundant replication, which provides maximum durability for your data. For more details on Azure Storage replication options, see [Azure Storage replication](#).

7. Click **Create Storage Account**.

It may take a few minutes to create your storage account. To check the status, you can monitor the notifications at the bottom of the Azure Classic Portal. After the storage account has been created, your new storage account has **Online** status and is ready for use.

The screenshot shows the Windows Azure Storage dashboard. On the left, there's a sidebar with icons for ALL, WEB SITES, CLOUD SERVICES, VIRTUAL MACHINES, STORAGE (which is selected), SQL DATABASES, and NETWORKS. The main area is titled 'storage' and lists two entries:

NAME	STATUS	LOCATION	SUBSCRIPTION
storagewestus	✓ Online	West US	myaccountSubscription
storagewestusaffinitygrp	✓ Online	West US	myaccountSubscription

At the bottom of the screen, a notification bar displays the message: "✓ Storage Account 'storagewestusaffinitygrp' created successfully".

Storage account endpoints

Every object that you store in Azure Storage has a unique URL address. The storage account name forms the subdomain of that address. The combination of subdomain and domain name, which is specific to each service, forms an *endpoint* for your storage account.

For example, if your storage account is named *mystorageaccount*, then the default endpoints for your storage account are:

- Blob service: <http://mystorageaccount.blob.core.windows.net>
- Table service: <http://mystorageaccount.table.core.windows.net>
- Queue service: <http://mystorageaccount.queue.core.windows.net>
- File service: <http://mystorageaccount.file.core.windows.net>

You can see the endpoints for your storage account on the storage dashboard in the [Azure Classic Portal](#) once the account has been created.

The URL for accessing an object in a storage account is built by appending the object's location in the storage account to the endpoint. For example, a blob address might have this format:
<http://mystorageaccount.blob.core.windows.net/mycontainer/myblob>.

You can also configure a custom domain name to use with your storage account. See [Configure a custom domain name for your blob storage endpoint](#) for details.

Service co-location with an affinity group

An *affinity group* is a geographic grouping of your Azure services and VMs with your Azure storage account. An affinity group can improve service performance by locating computer workloads in the same data center or near the target user audience. Also, no billing charges are incurred for egress when data in a storage account is

accessed from another service that is part of the same affinity group.

NOTE

To create an affinity group, open the **Settings** area of the [Azure Classic Portal](#), click **Affinity Groups**, and then click either **Add an affinity group** or the **Add** button. You can also create and manage affinity groups by using the Azure Service Management API. See [Operations on affinity groups](#) for more information.

View, copy, and regenerate storage access keys

When you create a storage account, Azure generates two 512-bit storage access keys, which are used for authentication when the storage account is accessed. By providing two storage access keys, Azure enables you to regenerate the keys with no interruption to your storage service or access to that service.

NOTE

We recommend that you avoid sharing your storage access keys with anyone else. To permit access to storage resources without giving out your access keys, you can use a *shared access signature*. A shared access signature provides access to a resource in your account for an interval that you define and with the permissions that you specify. See [Using Shared Access Signatures \(SAS\)](#) for more information.

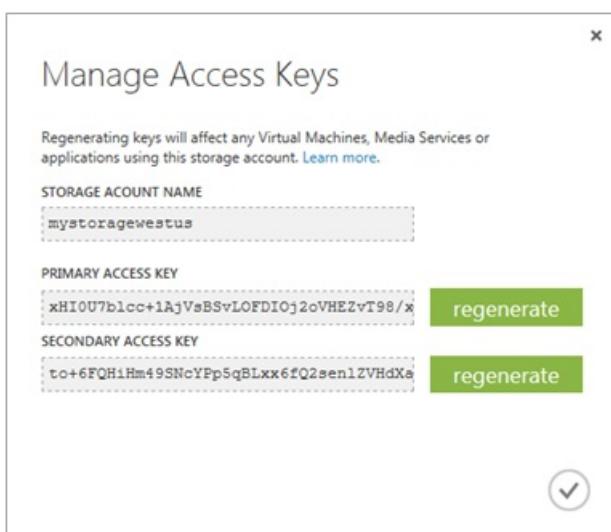
In the [Azure Classic Portal](#), use **Manage Keys** on the dashboard or the **Storage** page to view, copy, and regenerate the storage access keys that are used to access the Blob, Table, and Queue services.

Copy a storage access key

You can use **Manage Keys** to copy a storage access key to use in a connection string. The connection string requires the storage account name and a key to use in authentication. For information about configuring connection strings to access Azure storage services, see [Configure Azure Storage Connection Strings](#).

1. In the [Azure Classic Portal](#), click **Storage**, and then click the name of the storage account to open the dashboard.
2. Click **Manage Keys**.

Manage Access Keys opens.



3. To copy a storage access key, select the key text. Then right-click, and click **Copy**.

Regenerate storage access keys

We recommend that you change the access keys to your storage account periodically to help keep your storage connections secure. Two access keys are assigned so that you can maintain connections to the storage account by

using one access key while you regenerate the other access key.

WARNING

Regenerating your access keys can affect services in Azure as well as your own applications that are dependent on the storage account. All clients that use the access key to access the storage account must be updated to use the new key.

Media services - If you have media services that are dependent on your storage account, you must re-sync the access keys with your media service after you regenerate the keys.

Applications - If you have web applications or cloud services that use the storage account, you will lose the connections if you regenerate keys, unless you roll your keys.

Storage Explorers - If you are using any [storage explorer applications](#), you will probably need to update the storage key used by those applications.

Here is the process for rotating your storage access keys:

1. Update the connection strings in your application code to reference the secondary access key of the storage account.
2. Regenerate the primary access key for your storage account. In the [Azure Classic Portal](#), from the dashboard or the **Configure** page, click **Manage Keys**. Click **Regenerate** under the primary access key, and then click **Yes** to confirm that you want to generate a new key.
3. Update the connection strings in your code to reference the new primary access key.
4. Regenerate the secondary access key.

Delete a storage account

To remove a storage account that you are no longer using, use **Delete** on the dashboard or the **Configure** page.

Delete deletes the entire storage account, including all of the blobs, tables, and queues in the account.

WARNING

It's not possible to restore a deleted storage account or retrieve any of the content that it contained before deletion. Be sure to back up anything you want to save before you delete the account. This also holds true for any resources in the account—once you delete a blob, table, queue, or file, it is permanently deleted.

If your storage account contains VHD files for an Azure virtual machine, then you must delete any images and disks that are using those VHD files before you can delete the storage account. First, stop the virtual machine if it is running, and then delete it. To delete disks, navigate to the **Disks** tab and delete any disks there. To delete images, navigate to the **Images** tab and delete any images that are stored in the account.

1. In the [Azure Classic Portal](#), click **Storage**.
2. Click anywhere in the storage account entry except the name, and then click **Delete**.

-Or-

Click the name of the storage account to open the dashboard, and then click **Delete**.

3. Click **Yes** to confirm that you want to delete the storage account.

Next steps

- To learn more about Azure Storage, see the [Azure Storage documentation](#).
- Visit the [Azure Storage Team Blog](#).

- Transfer data with the AzCopy Command-Line Utility

Enabling Storage metrics and viewing metrics data

1/17/2017 • 8 min to read • [Edit on GitHub](#)

Overview

By default, Storage Metrics is not enabled for your storage services. You can enable monitoring using either the [Azure Classic Portal](#), Windows PowerShell, or programmatically through a storage API.

When you enable Storage Metrics, you must choose a retention period for the data: this period determines for how long the storage service keeps the metrics and charges you for the space required to store them. Typically, you should use a shorter retention period for minute metrics than hourly metrics because of the significant extra space required for minute metrics. You should choose a retention period such that you have sufficient time to analyze the data and download any metrics you wish to keep for off-line analysis or reporting purposes. Remember that you will also be billed for downloading metrics data from your storage account.

How to enable Storage metrics using the Azure Classic Portal

In the [Azure Classic Portal](#), you use the Configure page for a storage account to control Storage Metrics. For monitoring, you can set a level and a retention period in days for each of blobs, tables, and queues. In each case, the level is one of the following:

- Off — No metrics are collected.
- Minimal — Storage Metrics collects a basic set of metrics such as ingress/egress, availability, latency, and success percentages, which are aggregated for the Blob, Table, and Queue services.
- Verbose — Storage Metrics collects a full set of metrics that includes the same metrics for each storage API operation, in addition to the service-level metrics. Verbose metrics enable closer analysis of issues that occur during application operations.

Note that the Azure Classic Portal does not currently enable you to configure minute metrics in your storage account; you must enable minute metrics using PowerShell or programmatically.

How to enable Storage metrics using PowerShell

You can use PowerShell on your local machine to configure Storage Metrics in your storage account by using the Azure PowerShell cmdlet `Get-AzureStorageServiceMetricsProperty` to retrieve the current settings, and the cmdlet `Set-AzureStorageServiceMetricsProperty` to change the current settings.

The cmdlets that control Storage Metrics use the following parameters:

- `MetricsType` possible values are Hour and Minute.
- `ServiceType` possible values are Blob, Queue, and Table.
- `MetricsLevel` possible values are None (equivalent to Off in the Azure Classic Portal), Service (equivalent to Minimal in the Azure Classic Portal), and ServiceAndApi (equivalent to Verbose in the Azure Classic Portal).

For example, the following command switches on minute metrics for the blob service in your default storage account with the retention period set to five days:

```
Set-AzureStorageServiceMetricsProperty -MetricsType Minute -ServiceType Blob -MetricsLevel ServiceAndApi -RetentionDays 5
```

The following command retrieves the current hourly metrics level and retention days for the blob service in your

default storage account:

```
Get-AzureStorageServiceMetricsProperty -MetricsType Hour -ServiceType Blob
```

For information about how to configure the Azure PowerShell cmdlets to work with your Azure subscription and how to select the default storage account to use, see: [How to install and configure Azure PowerShell](#).

How to enable Storage metrics programmatically

The following C# snippet shows how to enable metrics and logging for the Blob service using the storage client library for .NET:

```
//Parse the connection string for the storage account.
const string ConnectionString = "DefaultEndpointsProtocol=https;AccountName=account-name;AccountKey=account-
key";
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(ConnectionString);

// Create service client for credentialed access to the Blob service.
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Enable Storage Analytics logging and set retention policy to 10 days.
ServiceProperties properties = new ServiceProperties();
properties.Logging.LoggingOperations = LoggingOperations.All;
properties.Logging.RetentionDays = 10;
properties.Logging.Version = "1.0";

// Configure service properties for metrics. Both metrics and logging must be set at the same time.
properties.HourMetrics.MetricsLevel = MetricsLevel.ServiceAndApi;
properties.HourMetrics.RetentionDays = 10;
properties.HourMetrics.Version = "1.0";

properties.MinuteMetrics.MetricsLevel = MetricsLevel.ServiceAndApi;
properties.MinuteMetrics.RetentionDays = 10;
properties.MinuteMetrics.Version = "1.0";

// Set the default service version to be used for anonymous requests.
properties.DefaultServiceVersion = "2015-04-05";

// Set the service properties.
blobClient.SetServiceProperties(properties);
```

Viewing Storage metrics

When you have configured Storage Metrics to monitor your storage account, it records the metrics in a set of well-known tables in your storage account. You can use the Monitor page for your storage account in the Azure Classic Portal to view the hourly metrics as they become available on a chart. On this page in the Azure Classic Portal, you can:

- Select which metrics to plot on the chart (the choice of available metrics will depend on whether you chose verbose or minimal monitoring for the service on the Configure page).
- Select the time range for the metrics displayed on the chart.
- Choose to use an absolute or relative scale to plot the metrics.
- Configure email alerts to notify you when a specific metric reaches a certain value.

If you want to download the metrics for long-term storage or to analyze them locally, you will need to use a tool or write some code to read the tables. You must download the minute metrics for analysis. The tables do not appear if you list all the tables in your storage account, but you can access them directly by name. Many third-party storage-browsing tools are aware of these tables and enable you to view them directly (see the blog post [Microsoft Azure Storage Explorers](#) for a list of available tools).

Hourly metrics

- \$MetricsHourPrimaryTransactionsBlob
- \$MetricsHourPrimaryTransactionsTable
- \$MetricsHourPrimaryTransactionsQueue

Minute metrics

- \$MetricsMinutePrimaryTransactionsBlob
- \$MetricsMinutePrimaryTransactionsTable
- \$MetricsMinutePrimaryTransactionsQueue

Capacity

- \$MetricsCapacityBlob

You can find full details of the schemas for these tables at [Storage Analytics Metrics Table Schema](#). The sample rows below show only a subset of the columns available, but illustrate some important features of the way Storage Metrics saves these metrics:

PARTITIONKEY	ROWKEY	TIMESTAMP	TOTALREQUESTS	TOTALBILLABLEREQUESTS	TOTALINGRESS	TOTALEGRESS	AVAILABILITY	AVERAGE2ELATENCY	AVERAGESERVEABILITY	PERCENTSUCCESS
20140522T1100	user;All	2014-05-22T11:01:16.7650250Z	7	7	4003	46801	100	104.4286	6.857143	100
20140522T1100	user;QueryEntities	2014-05-22T11:01:16.7640250Z	5	5	2694	45951	100	143.8	7.8	100
20140522T1100	user;QueryEntity	2014-05-22T11:01:16.7650250Z	1	1	538	633	100	3	3	100
20140522T1100	user;UpdateEntity	2014-05-22T11:01:16.7650250Z	1	1	771	217	100	9	6	100

In this example minute metrics data, the partition key uses the time at minute resolution. The row key identifies the type of information that is stored in the row and this is composed of two pieces of information, the access type, and the request type:

- The access type is either user or system, where user refers to all user requests to the storage service, and system refers to requests made by Storage Analytics.
- The request type is either all in which case it is a summary line, or it identifies the specific API such as QueryEntity or UpdateEntity.

The sample data above shows all the records for a single minute (starting at 11:00AM), so the number of QueryEntities requests plus the number of QueryEntity requests plus the number of UpdateEntity requests add up to seven, which is the total shown on the user>All row. Similarly, you can derive the average end-to-end latency 104.4286 on the user>All row by calculating $((143.8 * 5) + 3 + 9)/7$.

You should consider setting up alerts in the Azure Classic Portal on the Monitor page so that Storage Metrics can automatically notify you of any important changes in the behavior of your storage services. If you use a storage explorer tool to download this metrics data in a delimited format, you can use Microsoft Excel to analyze the data. See the blog post [Microsoft Azure Storage Explorers](#) for a list of available storage explorer tools.

Accessing metrics data programmatically

The following listing shows sample C# code that accesses the minute metrics for a range of minutes and displays the results in a console Window. It uses the Azure Storage Library version 4 that includes the CloudAnalyticsClient class that simplifies accessing the metrics tables in storage.

```
private static void PrintMinuteMetrics(CloudAnalyticsClient analyticsClient, DateTimeOffset startDateTime,
DateTimeOffset endDateTime)
{
    // Convert the dates to the format used in the PartitionKey
    var start = startDateTime.ToUniversalTime().ToString("yyyyMMdd'T'HHmm");
    var end = endDateTime.ToUniversalTime().ToString("yyyyMMdd'T'HHmm");

    var services = Enum.GetValues(typeof(StorageService));
    foreach (StorageService service in services)
    {
        Console.WriteLine("Minute Metrics for Service {0} from {1} to {2} UTC", service, start, end);
        var metricsQuery = analyticsClient.CreateMinuteMetricsQuery(service, StorageLocation.Primary);
        var t = analyticsClient.GetMinuteMetricsTable(service);
        var opContext = new OperationContext();
        var query =
            from entity in metricsQuery
            // Note, you can't filter using the entity properties Time, AccessType, or TransactionType
            // because they are calculated fields in the MetricsEntity class.
            // The PartitionKey identifies the DateTime of the metrics.
            where entity.PartitionKey.CompareTo(start) >= 0 && entity.PartitionKey.CompareTo(end) <= 0
            select entity;

        // Filter on "user" transactions after fetching the metrics from Table Storage.
        // (StartsWith is not supported using LINQ with Azure table storage)
        var results = query.ToList().Where(m => m.RowKey.StartsWith("user"));
        var resultString = results.Aggregate(new StringBuilder(), (builder, metrics) =>
builder.AppendLine(MetricsString(metrics, opContext))).ToString();
        Console.WriteLine(resultString);
    }
}

private static string MetricsString(MetricsEntity entity, OperationContext opContext)
{
    var entityProperties = entity.WriteEntity(opContext);
    var entityString =
        string.Format("Time: {0}, ", entity.Time) +
        string.Format("AccessType: {0}, ", entity.AccessType) +
        string.Format("TransactionType: {0}, ", entity.TransactionType) +
        string.Join(", ", entityProperties.Select(e => new KeyValuePair<string, string>(e.Key.ToString(),
e.Value.PropertyAsObject.ToString())));
    return entityString;
}
```

What charges do you incur when you enable storage metrics?

Write requests to create table entities for metrics are charged at the standard rates applicable to all Azure Storage

operations.

Read and delete requests by a client to metrics data are also billable at standard rates. If you have configured a data retention policy, you are not charged when Azure Storage deletes old metrics data. However, if you delete analytics data, your account is charged for the delete operations.

The capacity used by the metrics tables is also billable: you can use the following to estimate the amount of capacity used for storing metrics data:

- If each hour a service utilizes every API in every service, then approximately 148KB of data is stored every hour in the metrics transaction tables if you have enabled both service and API level summary.
- If each hour a service utilizes every API in every service, then approximately 12KB of data is stored every hour in the metrics transaction tables if you have enabled just service level summary.
- The capacity table for blobs has two rows added each day (provided user has opted in for logs): this implies that every day the size of this table increases by up to approximately 300 bytes.

Next steps:

[Enabling Storage Analytics Logging and Accessing Log Data](#)

Monitor, diagnose, and troubleshoot Microsoft Azure Storage

1/17/2017 • 56 min to read • [Edit on GitHub](#)

Overview

Diagnosing and troubleshooting issues in a distributed application hosted in a cloud environment can be more complex than in traditional environments. Applications can be deployed in a PaaS or IaaS infrastructure, on-premises, on a mobile device, or in some combination of these. Typically, your application's network traffic may traverse public and private networks and your application may use multiple storage technologies such as Microsoft Azure Storage Tables, Blobs, Queues, or Files in addition to other data stores such as relational and document databases.

To manage such applications successfully you should monitor them proactively and understand how to diagnose and troubleshoot all aspects of them and their dependent technologies. As a user of Azure Storage services, you should continuously monitor the Storage services your application uses for any unexpected changes in behavior (such as slower than usual response times), and use logging to collect more detailed data and to analyze a problem in depth. The diagnostics information you obtain from both monitoring and logging will help you to determine the root cause of the issue your application encountered. Then you can troubleshoot the issue and determine the appropriate steps you can take to remediate it. Azure Storage is a core Azure service, and forms an important part of the majority of solutions that customers deploy to the Azure infrastructure. Azure Storage includes capabilities to simplify monitoring, diagnosing, and troubleshooting storage issues in your cloud-based applications.

NOTE

Storage accounts with a replication type of Zone-Redundant Storage (ZRS) do not have the metrics or logging capability enabled at this time.

For a hands-on guide to end-to-end troubleshooting in Azure Storage applications, see [End-to-End Troubleshooting using Azure Storage Metrics and Logging, AzCopy, and Message Analyzer](#).

- [Introduction](#)
 - [How this guide is organized](#)
- [Monitoring your storage service](#)
 - [Monitoring service health](#)
 - [Monitoring capacity](#)
 - [Monitoring availability](#)
 - [Monitoring performance](#)
- [Diagnosing storage issues](#)
 - [Service health issues](#)
 - [Performance issues](#)
 - [Diagnosing errors](#)
 - [Storage emulator issues](#)
 - [Storage logging tools](#)
 - [Using network logging tools](#)
- [End-to-end tracing](#)
 - [Correlating log data](#)

- [Client request ID](#)
 - [Server request ID](#)
 - [Timestamps](#)
- [Troubleshooting guidance](#)
 - Metrics show high AverageE2ELatency and low AverageServerLatency
 - Metrics show low AverageE2ELatency and low AverageServerLatency but the client is experiencing high latency
 - Metrics show high AverageServerLatency
 - You are experiencing unexpected delays in message delivery on a queue
 - Metrics show an increase in PercentThrottlingError
 - Metrics show an increase in PercentTimeoutError
 - Metrics show an increase in PercentNetworkError
 - The client is receiving HTTP 403 (Forbidden) messages
 - The client is receiving HTTP 404 (Not found) messages
 - The client is receiving HTTP 409 (Conflict) messages
 - Metrics show low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors
 - Capacity metrics show an unexpected increase in storage capacity usage
 - You are experiencing unexpected reboots of Virtual Machines that have a large number of attached VHDs
 - Your issue arises from using the storage emulator for development or test
 - You are encountering problems installing the Azure SDK for .NET
 - You have a different issue with a storage service
- [Appendices](#)
 - [Appendix 1: Using Fiddler to capture HTTP and HTTPS traffic](#)
 - [Appendix 2: Using Wireshark to capture network traffic](#)
 - [Appendix 3: Using Microsoft Message Analyzer to capture network traffic](#)
 - [Appendix 4: Using Excel to view metrics and log data](#)
 - [Appendix 5: Monitoring with Application Insights for Visual Studio Team Services](#)

Introduction

This guide shows you how to use features such as Azure Storage Analytics, client-side logging in the Azure Storage Client Library, and other third-party tools to identify, diagnose, and troubleshoot Azure Storage related issues.

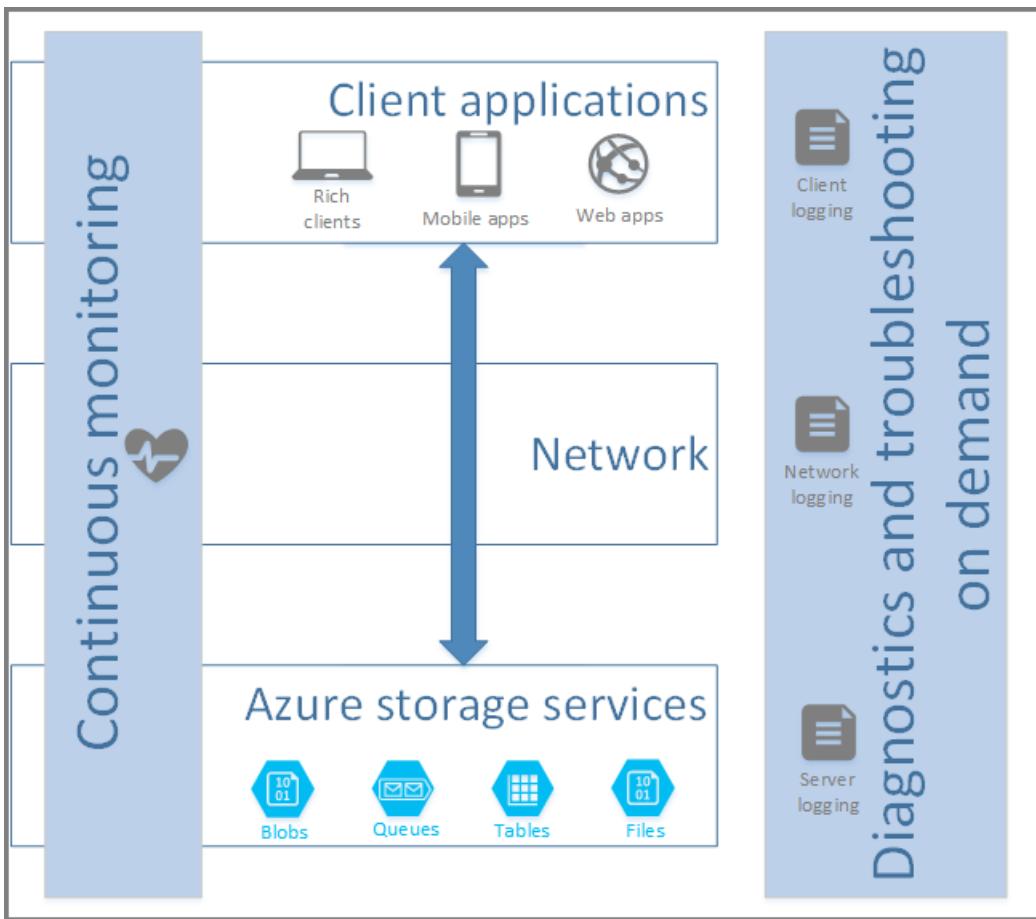


Figure 1 Monitoring, Diagnostics, and Troubleshooting

This guide is intended to be read primarily by developers of online services that use Azure Storage Services and IT Pros responsible for managing such online services. The goals of this guide are:

- To help you maintain the health and performance of your Azure Storage accounts.
- To provide you with the necessary processes and tools to help you decide if an issue or problem in an application relates to Azure Storage.
- To provide you with actionable guidance for resolving problems related to Azure Storage.

How this guide is organized

The section "[Monitoring your storage service](#)" describes how to monitor the health and performance of your Azure Storage services using Azure Storage Analytics Metrics (Storage Metrics).

The section "[Diagnosing storage issues](#)" describes how to diagnose issues using Azure Storage Analytics Logging (Storage Logging). It also describes how to enable client-side logging using the facilities in one of the client libraries such as the Storage Client Library for .NET or the Azure SDK for Java.

The section "[End-to-end tracing](#)" describes how you can correlate the information contained in various log files and metrics data.

The section "[Troubleshooting guidance](#)" provides troubleshooting guidance for some of the common storage-related issues you might encounter.

The "[Appendices](#)" include information about using other tools such as Wireshark and Netmon for analyzing network packet data, Fiddler for analyzing HTTP/HTTPS messages, and Microsoft Message Analyzer for correlating log data.

Monitoring your storage service

If you are familiar with Windows performance monitoring, you can think of Storage Metrics as being an Azure

Storage equivalent of Windows Performance Monitor counters. In Storage Metrics you will find a comprehensive set of metrics (counters in Windows Performance Monitor terminology) such as service availability, total number of requests to service, or percentage of successful requests to service (for a full list of the available metrics, see [Storage Analytics Metrics Table Schema](#) on MSDN). You can specify whether you want the storage service to collect and aggregate metrics every hour or every minute. For more information about how to enable metrics and monitor your storage accounts, see [Enabling storage metrics](#) on MSDN.

You can choose which hourly metrics you want to display in the Azure Classic Portal and configure rules that notify administrators by email whenever an hourly metric exceeds a particular threshold (for more information, see the page [How to: Receive Alert Notifications and Manage Alert Rules in Azure](#)). The storage service collects metrics using a best effort, but may not record every storage operation.

Figure 2 below shows the Monitor page in the Azure Classic Portal where you can view metrics such as availability, total requests, and average latency numbers for a storage account. A notification rule has also been set up to alert an administrator if availability drops below a certain level. From viewing this data, one possible area for investigation is the table service success percentage being below 100% (for more information, see the section "Metrics show low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors").

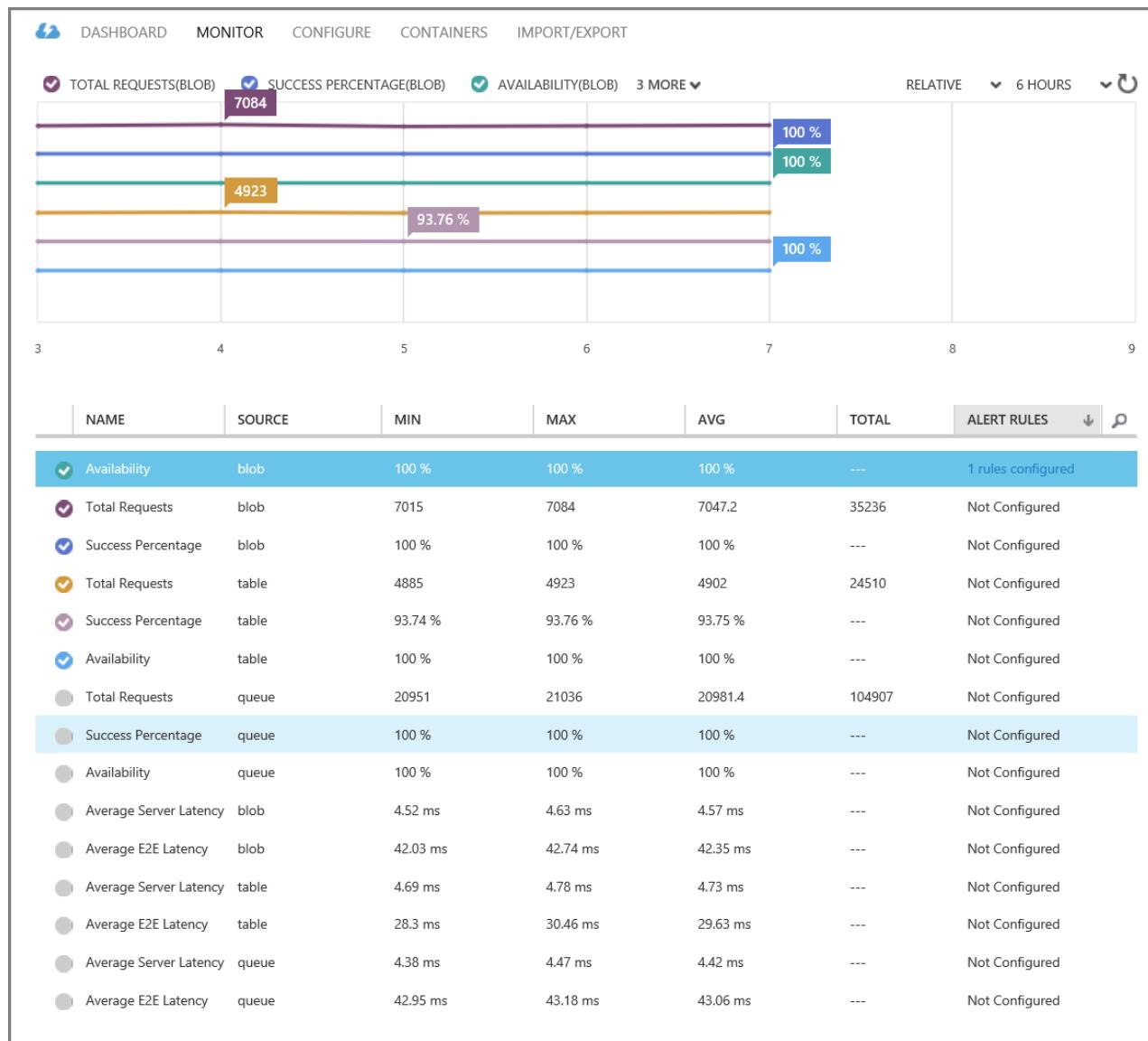


Figure 2 Viewing storage metrics in the Azure Classic Portal

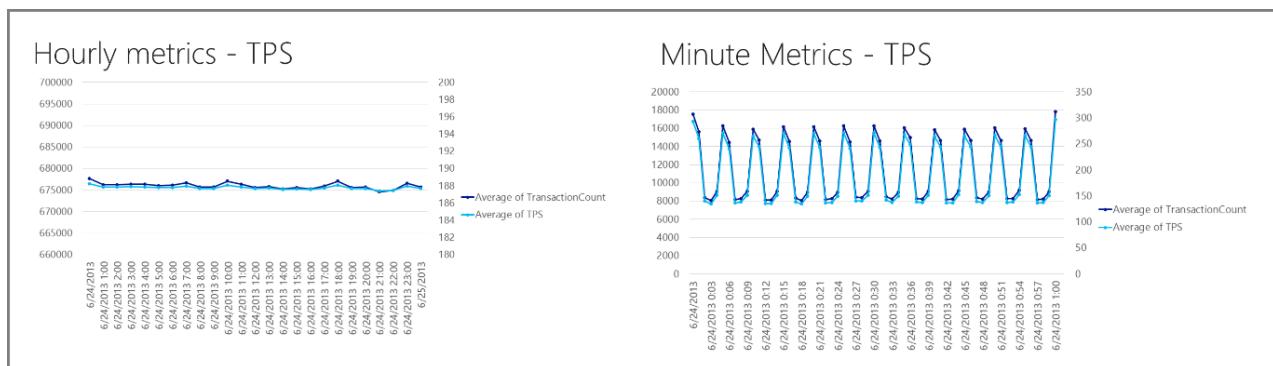
You should continuously monitor your Azure applications to ensure they are healthy and performing as expected by:

- Establishing some baseline metrics for application that will enable you to compare current data and identify any

significant changes in the behavior of Azure storage and your application. The values of your baseline metrics will, in many cases, be application specific and you should establish them when you are performance testing your application.

- Recording minute metrics and using them to monitor actively for unexpected errors and anomalies such as spikes in error counts or request rates.
- Recording hourly metrics and using them to monitor average values such as average error counts and request rates.
- Investigating potential issues using diagnostics tools as discussed later in the section "[Diagnosing storage issues](#)."

The charts in Figure 3 below illustrate how the averaging that occurs for hourly metrics can hide spikes in activity. The hourly metrics appear to show a steady rate of requests, while the minute metrics reveal the fluctuations that are really taking place.



The remainder of this section describes what metrics you should monitor and why.

Monitoring service health

You can use the [Azure Classic Portal](#) to view the health of the Storage service (and other Azure services) in all the Azure regions around the world. This enables you to see immediately if an issue outside of your control is affecting the Storage service in the region you use for your application.

The Azure Classic Portal can also provide with notifications of incidents that affect the various Azure services. Note: This information was previously available, along with historical data, on the Azure Service Dashboard at <http://status.azure.com>.

While the Azure Classic Portal collects health information from inside the Azure datacenters (inside-out monitoring), you could also consider adopting an outside-in approach to generate synthetic transactions that periodically access your Azure-hosted web application from multiple locations. The services offered by [Keynote](#), [Gomez](#), and Application Insights for Visual Studio Team Services are examples of this outside-in approach. For more information about Application Insights for Visual Studio Team Services, see the appendix "[Appendix 5: Monitoring with Application Insights for Visual Studio Team Services](#)."

Monitoring capacity

Storage Metrics only stores capacity metrics for the blob service because blobs typically account for the largest proportion of stored data (at the time of writing, it is not possible to use Storage Metrics to monitor the capacity of your tables and queues). You can find this data in the **\$MetricsCapacityBlob** table if you have enabled monitoring for the Blob service. Storage Metrics records this data once per day, and you can use the value of the **RowKey** to determine whether the row contains an entity that relates to user data (value **data**) or analytics data (value **analytics**). Each stored entity contains information about the amount of storage used (**Capacity** measured in bytes) and the current number of containers (**ContainerCount**) and blobs (**ObjectCount**) in use in the storage account. For more information about the capacity metrics stored in the **\$MetricsCapacityBlob** table, see [Storage Analytics Metrics Table Schema](#) on MSDN.

NOTE

You should monitor these values for an early warning that you are approaching the capacity limits of your storage account. In the Azure Classic Portal, on the **Monitor** page for your storage account, you can add alert rules to notify you if aggregate storage use exceeds or falls below thresholds that you specify.

For help estimating the size of various storage objects such as blobs, see the blog post [Understanding Azure Storage Billing – Bandwidth, Transactions, and Capacity](#).

Monitoring availability

You should monitor the availability of the storage services in your storage account by monitoring the value in the **Availability** column in the hourly or minute metrics tables — **\$MetricsHourPrimaryTransactionsBlob**, **\$MetricsHourPrimaryTransactionsTable**, **\$MetricsHourPrimaryTransactionsQueue**, **\$MetricsMinutePrimaryTransactionsBlob**, **\$MetricsMinutePrimaryTransactionsTable**, **\$MetricsMinutePrimaryTransactionsQueue**, **\$MetricsCapacityBlob**. The **Availability** column contains a percentage value that indicates the availability of the service or the API operation represented by the row (the **RowKey** shows if the row contains metrics for the service as a whole or for a specific API operation).

Any value less than 100% indicates that some storage requests are failing. You can see why they are failing by examining the other columns in the metrics data that show the numbers of requests with different error types such as **ServerTimeoutError**. You should expect to see **Availability** fall temporarily below 100% for reasons such as transient server timeouts while the service moves partitions to better load-balance request; the retry logic in your client application should handle such intermittent conditions. The page lists the transaction types that Storage Metrics includes in its **Availability** calculation.

In the Azure Classic Portal, on the **Monitor** page for your storage account, you can add alert rules to notify you if **Availability** for a service falls below a threshold that you specify.

The "[Troubleshooting guidance](#)" section of this guide describes some common storage service issues related to availability.

Monitoring performance

To monitor the performance of the storage services, you can use the following metrics from the hourly and minute metrics tables.

- The values in the **AverageE2ELatency** and **AverageServerLatency** show the average time the storage service or API operation type is taking to process requests. **AverageE2ELatency** is a measure of end-to-end latency that includes the time taken to read the request and send the response in addition to the time taken to process the request (therefore includes network latency once the request reaches the storage service); **AverageServerLatency** is a measure of just the processing time and therefore excludes any network latency related to communicating with the client. See the section "[Metrics show high AverageE2ELatency and low AverageServerLatency](#)" later in this guide for a discussion of why there might be a significant difference between these two values.
- The values in the **TotalIngress** and **TotalEgress** columns show the total amount of data, in bytes, coming in to and going out of your storage service or through a specific API operation type.
- The values in the **TotalRequests** column show the total number of requests that the storage service or API operation is receiving. **TotalRequests** is the total number of requests that the storage service receives.

Typically, you will monitor for unexpected changes in any of these values as an indicator that you have an issue that requires investigation.

In the Azure Classic Portal, on the **Monitor** page for your storage account, you can add alert rules to notify you if any of the performance metrics for this service fall below or exceed a threshold that you specify.

The "[Troubleshooting guidance](#)" section of this guide describes some common storage service issues related to

performance.

Diagnosing storage issues

There are a number of ways that you might become aware of a problem or issue in your application, these include:

- A major failure that causes the application to crash or to stop working.
- Significant changes from baseline values in the metrics you are monitoring as described in the previous section "[Monitoring your storage service](#)."
- Reports from users of your application that some particular operation didn't complete as expected or that some feature is not working.
- Errors generated within your application that appear in log files or through some other notification method.

Typically, issues related to Azure storage services fall into one of four broad categories:

- Your application has a performance issue, either reported by your users, or revealed by changes in the performance metrics.
- There is a problem with the Azure Storage infrastructure in one or more regions.
- Your application is encountering an error, either reported by your users, or revealed by an increase in one of the error count metrics you monitor.
- During development and test, you may be using the local storage emulator; you may encounter some issues that relate specifically to usage of the storage emulator.

The following sections outline the steps you should follow to diagnose and troubleshoot issues in each of these four categories. The section "[Troubleshooting guidance](#)" later in this guide provides more detail for some common issues you may encounter.

Service health issues

Service health issues are typically outside of your control. The Azure Classic Portal provides information about any ongoing issues with Azure services including storage services. If you opted for Read-Access Geo-Redundant Storage when you created your storage account, then in the event of your data being unavailable in the primary location, your application could switch temporarily to the read-only copy in the secondary location. To do this, your application must be able to switch between using the primary and secondary storage locations, and be able to work in a reduced functionality mode with read-only data. The Azure Storage Client libraries allow you to define a retry policy that can read from secondary storage in case a read from primary storage fails. Your application also needs to be aware that the data in the secondary location is eventually consistent. For more information, see the blog post [Azure Storage Redundancy Options and Read Access Geo Redundant Storage](#).

Performance issues

The performance of an application can be subjective, especially from a user perspective. Therefore, it is important to have baseline metrics available to help you identify where there might be a performance issue. Many factors might affect the performance of an Azure storage service from the client application perspective. These factors might operate in the storage service, in the client, or in the network infrastructure; therefore it is important to have a strategy for identifying the origin of the performance issue.

After you have identified the likely location of the cause of the performance issue from the metrics, you can then use the log files to find detailed information to diagnose and troubleshoot the problem further.

The section "[Troubleshooting guidance](#)" later in this guide provides more information about some common performance related issues you may encounter.

Diagnosing errors

Users of your application may notify you of errors reported by the client application. Storage Metrics also records counts of different error types from your storage services such as **NetworkError**, **ClientTimeoutError**, or **AuthorizationError**. While Storage Metrics only records counts of different error types, you can obtain more

detail about individual requests by examining server-side, client-side, and network logs. Typically, the HTTP status code returned by the storage service will give an indication of why the request failed.

NOTE

Remember that you should expect to see some intermittent errors: for example, errors due to transient network conditions, or application errors.

The following resources on MSDN are useful for understanding storage-related status and error codes:

- [Common REST API Error Codes](#)
- [Blob Service Error Codes](#)
- [Queue Service Error Codes](#)
- [Table Service Error Codes](#)

Storage emulator issues

The Azure SDK includes a storage emulator you can run on a development workstation. This emulator simulates most of the behavior of the Azure storage services and is useful during development and test, enabling you to run applications that use Azure storage services without the need for an Azure subscription and an Azure storage account.

The "[Troubleshooting guidance](#)" section of this guide describes some common issues encountered using the storage emulator.

Storage logging tools

Storage Logging provides server-side logging of storage requests in your Azure storage account. For more information about how to enable server-side logging and access the log data, see [Using server-side logging](#) on MSDN.

The Storage Client Library for .NET enables you to collect client-side log data that relates to storage operations performed by your application. For more information about how to enable client-side logging and access the log data, see [Client-side logging using the Storage Client Library](#) on MSDN.

NOTE

In some circumstances (such as SAS authorization failures), a user may report an error for which you can find no request data in the server-side Storage logs. You can use the logging capabilities of the Storage Client Library to investigate if the cause of the issue is on the client or use network monitoring tools to investigate the network.

Using network logging tools

You can capture the traffic between the client and server to provide detailed information about the data the client and server are exchanging and the underlying network conditions. Useful network logging tools include:

- Fiddler (<http://www.telerik.com/fiddler>) is a free web debugging proxy that enables you to examine the headers and payload data of HTTP and HTTPS request and response messages. For more information, see "[Appendix 1: Using Fiddler to capture HTTP and HTTPS traffic](#)".
- Microsoft Network Monitor (Netmon) (<http://www.microsoft.com/download/details.aspx?id=4865>) and Wireshark (<http://www.wireshark.org/>) are free network protocol analyzers that enable you to view detailed packet information for a wide range of network protocols. For more information about Wireshark, see "[Appendix 2: Using Wireshark to capture network traffic](#)".
- Microsoft Message Analyzer is a tool from Microsoft that supersedes Netmon and that in addition to capturing network packet data, helps you to view and analyze the log data captured from other tools. For more information, see "[Appendix 3: Using Microsoft Message Analyzer to capture network traffic](#)".

- If you want to perform a basic connectivity test to check that your client machine can connect to the Azure storage service over the network, you cannot do this using the standard **ping** tool on the client. However, you can use the **tcping** tool to check connectivity. **Tcping** is available for download at <http://www.elifulkerson.com/projects/tcping.php>.

In many cases, the log data from Storage Logging and the Storage Client Library will be sufficient to diagnose an issue, but in some scenarios, you may need the more detailed information that these network logging tools can provide. For example, using Fiddler to view HTTP and HTTPS messages enables you to view header and payload data sent to and from the storage services, which would enable you to examine how a client application retries storage operations. Protocol analyzers such as Wireshark operate at the packet level enabling you to view TCP data, which would enable you to troubleshoot lost packets and connectivity issues. Message Analyzer can operate at both HTTP and TCP layers.

End-to-end tracing

End-to-end tracing using a variety of log files is a useful technique for investigating potential issues. You can use the date/time information from your metrics data as an indication of where to start looking in the log files for the detailed information that will help you troubleshoot the issue.

Correlating log data

When viewing logs from client applications, network traces, and server-side storage logging it is critical to be able to correlate requests across the different log files. The log files include a number of different fields that are useful as correlation identifiers. The client request id is the most useful field to use to correlate entries in the different logs. However sometimes, it can be useful to use either the server request id or timestamps. The following sections provide more details about these options.

Client request ID

The Storage Client Library automatically generates a unique client request id for every request.

- In the client-side log that the Storage Client Library creates, the client request id appears in the **Client Request ID** field of every log entry relating to the request.
- In a network trace such as one captured by Fiddler, the client request id is visible in request messages as the **x-ms-client-request-id** HTTP header value.
- In the server-side Storage Logging log, the client request id appears in the Client request ID column.

NOTE

It is possible for multiple requests to share the same client request id because the client can assign this value (although the Storage Client Library assigns a new value automatically). In the case of retries from the client, all attempts share the same client request id. In the case of a batch sent from the client, the batch has a single client request id.

Server request ID

The storage service automatically generates server request ids.

- In the server-side Storage Logging log, the server request id appears the **Request ID header** column.
- In a network trace such as one captured by Fiddler, the server request id appears in response messages as the **x-ms-request-id** HTTP header value.
- In the client-side log that the Storage Client Library creates, the server request id appears in the **Operation Text** column for the log entry showing details of the server response.

NOTE

The storage service always assigns a unique server request id to every request it receives, so every retry attempt from the client and every operation included in a batch has a unique server request id.

If the Storage Client Library throws a **StorageException** in the client, the **RequestInformation** property contains a **RequestResult** object that includes a **ServiceRequestId** property. You can also access a **RequestResult** object from an **OperationContext** instance.

The code sample below demonstrates how to set a custom **ClientRequestId** value by attaching an **OperationContext** object the request to the storage service. It also shows how to retrieve the **ServerRequestId** value from the response message.

```
//Parse the connection string for the storage account.  
const string ConnectionString = "DefaultEndpointsProtocol=https;AccountName=account-name;AccountKey=account-key";  
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(ConnectionString);  
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();  
  
// Create an Operation Context that includes custom ClientRequestId string based on constants defined within the  
application along with a Guid.  
OperationContext oc = new OperationContext();  
oc.ClientRequestId = String.Format("{0} {1} {2} {3}", HOSTNAME, APPNAME, USERID, Guid.NewGuid().ToString());  
  
try  
{  
    CloudBlobContainer container = blobClient.GetContainerReference("democontainer");  
    ICloudBlob blob = container.GetBlobReferenceFromServer("testImage.jpg", null, null, oc);  
    var downloadToPath = string.Format("./{0}", blob.Name);  
    using (var fs = File.OpenWrite(downloadToPath))  
    {  
        blob.DownloadToStream(fs, null, null, oc);  
        Console.WriteLine("\t Blob downloaded to file: {0}", downloadToPath);  
    }  
}  
catch (StorageException storageException)  
{  
    Console.WriteLine("Storage exception {0} occurred", storageException.Message);  
    // Multiple results may exist due to client side retry logic - each retried operation will have a unique  
ServiceRequestId  
    foreach (var result in oc.RequestResults)  
    {  
        Console.WriteLine("HttpStatus: {0}, ServiceRequestId {1}", result.HttpStatusCode,  
result.ServiceRequestId);  
    }  
}
```

Timestamps

You can also use timestamps to locate related log entries, but be careful of any clock skew between the client and server that may exist. You should search plus or minus 15 minutes for matching server-side entries based on the timestamp on the client. Remember that the blob metadata for the blobs containing metrics indicates the time range for the metrics stored in the blob; this is useful if you have many metrics blobs for the same minute or hour.

Troubleshooting guidance

This section will help you with the diagnosis and troubleshooting of some of the common issues your application may encounter when using the Azure storage services. Use the list below to locate the information relevant to your specific issue.

Troubleshooting Decision Tree

Does your issue relate to the performance of one of the storage services?

- Metrics show high AverageE2ELatency and low AverageServerLatency
- Metrics show low AverageE2ELatency and low AverageServerLatency but the client is experiencing high latency
- Metrics show high AverageServerLatency
- You are experiencing unexpected delays in message delivery on a queue

Does your issue relate to the availability of one of the storage services?

- Metrics show an increase in PercentThrottlingError
- Metrics show an increase in PercentTimeoutError
- Metrics show an increase in PercentNetworkError

Is your client application receiving an HTTP 4XX (such as 404) response from a storage service?

- The client is receiving HTTP 403 (Forbidden) messages
- The client is receiving HTTP 404 (Not found) messages
- The client is receiving HTTP 409 (Conflict) messages

Metrics show low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors

Capacity metrics show an unexpected increase in storage capacity usage

You are experiencing unexpected reboots of Virtual Machines that have a large number of attached VHDs

Your issue arises from using the storage emulator for development or test

You are encountering problems installing the Azure SDK for .NET

You have a different issue with a storage service

Metrics show high AverageE2ELatency and low AverageServerLatency

The illustration below from the Azure Classic Portal monitoring tool shows an example where the **AverageE2ELatency** is significantly higher than the **AverageServerLatency**.

NAME	SOURCE	MIN	MAX	AVG	TOTAL	ALERT RULES
Average E2E Latency	queue	0 ms	139.65 ms	49.47 ms	---	Not Configured
Average Server Latency	queue	0 ms	4.59 ms	2.49 ms	---	Not Configured

Note that the storage service only calculates the metric **AverageE2ELatency** for successful requests and, unlike **AverageServerLatency**, includes the time the client takes to send the data and receive acknowledgement from the storage service. Therefore, a difference between **AverageE2ELatency** and **AverageServerLatency** could be either due to the client application being slow to respond, or due to conditions on the network.

NOTE

You can also view **E2ELatency** and **ServerLatency** for individual storage operations in the Storage Logging log data.

Investigating client performance issues

Possible reasons for the client responding slowly include having a limited number of available connections or

threads. You may be able to resolve the issue by modifying the client code to be more efficient (for example by using asynchronous calls to the storage service), or by using a larger Virtual Machine (with more cores and more memory).

For the table and queue services, the Nagle algorithm can also cause high **AverageE2ELatency** as compared to **AverageServerLatency**: for more information see the post [Nagle's Algorithm is Not Friendly towards Small Requests](#) on the Microsoft Azure Storage Team Blog. You can disable the Nagle algorithm in code by using the **ServicePointManager** class in the **System.Net** namespace. You should do this before you make any calls to the table or queue services in your application since this does not affect connections that are already open. The following example comes from the **Application_Start** method in a worker role.

```
var storageAccount = CloudStorageAccount.Parse(connStr);
ServicePoint tableServicePoint = ServicePointManager.FindServicePoint(storageAccount.TableEndpoint);
tableServicePoint.UseNagleAlgorithm = false;
ServicePoint queueServicePoint = ServicePointManager.FindServicePoint(storageAccount.QueueEndpoint);
queueServicePoint.UseNagleAlgorithm = false;
```

You should check the client-side logs to see how many requests your client application is submitting, and check for general .NET related performance bottlenecks in your client such as CPU, .NET garbage collection, network utilization, or memory (as a starting point for troubleshooting .NET client applications, see [Debugging, Tracing, and Profiling](#) on MSDN).

Investigating network latency issues

Typically, high end-to-end latency caused by the network is due to transient conditions. You can investigate both transient and persistent network issues such as dropped packets by using tools such as Wireshark or Microsoft Message Analyzer.

For more information about using Wireshark to troubleshoot network issues, see "[Appendix 2: Using Wireshark to capture network traffic](#)."

For more information about using Microsoft Message Analyzer to troubleshoot network issues, see "[Appendix 3: Using Microsoft Message Analyzer to capture network traffic](#)."

Metrics show low AverageE2ELatency and low AverageServerLatency but the client is experiencing high latency

In this scenario, the most likely cause is a delay in the storage requests reaching the storage service. You should investigate why requests from the client are not making it through to the blob service.

Possible reasons for the client delaying sending requests include having a limited number of available connections or threads. You should also check if the client is performing multiple retries, and investigate the reason if this is the case. You can do this programmatically by looking in the **OperationContext** object associated with the request and retrieving the **ServerRequestId** value. For more information, see the code sample in the section "[Server request ID](#)."

If there are no issues in the client, you should investigate potential network issues such as packet loss. You can use tools such as Wireshark or Microsoft Message Analyzer to investigate network issues.

For more information about using Wireshark to troubleshoot network issues, see "[Appendix 2: Using Wireshark to capture network traffic](#)."

For more information about using Microsoft Message Analyzer to troubleshoot network issues, see "[Appendix 3: Using Microsoft Message Analyzer to capture network traffic](#)."

Metrics show high AverageServerLatency

In the case of high **AverageServerLatency** for blob download requests, you should use the Storage Logging logs to see if there are repeated requests for the same blob (or set of blobs). For blob upload requests, you should investigate what block size the client is using (for example, blocks less than 64K in size can result in overheads unless the reads are also in less than 64K chunks), and if multiple clients are uploading blocks to the same blob in

parallel. You should also check the per-minute metrics for spikes in the number of requests that result in exceeding the per second scalability targets: also see "[Metrics show an increase in PercentTimeoutError](#)."

If you are seeing high **AverageServerLatency** for blob download requests when there are repeated requests the same blob or set of blobs, then you should consider caching these blobs using Azure Cache or the Azure Content Delivery Network (CDN). For upload requests, you can improve the throughput by using a larger block size. For queries to tables, it is also possible to implement client-side caching on clients that perform the same query operations and where the data doesn't change frequently.

High **AverageServerLatency** values can also be a symptom of poorly designed tables or queries that result in scan operations or that follow the append/prepend anti-pattern. See "[Metrics show an increase in PercentThrottlingError](#)" for more information.

NOTE

You can find a comprehensive checklist performance checklist here: [Microsoft Azure Storage Performance and Scalability Checklist](#).

You are experiencing unexpected delays in message delivery on a queue

If you are experiencing a delay between the time an application adds a message to a queue and the time it becomes available to read from the queue, then you should take the following steps to diagnose the issue:

- Verify the application is successfully adding the messages to the queue. Check that the application is not retrying the **AddMessage** method several times before succeeding. The Storage Client Library logs will show any repeated retries of storage operations.
- Verify there is no clock skew between the worker role that adds the message to the queue and the worker role that reads the message from the queue that makes it appear as if there is a delay in processing.
- Check if the worker role that reads the messages from the queue is failing. If a queue client calls the **GetMessage** method but fails to respond with an acknowledgement, the message will remain invisible on the queue until the **InvisibilityTimeout** period expires. At this point, the message becomes available for processing again.
- Check if the queue length is growing over time. This can occur if you do not have sufficient workers available to process all of the messages that other workers are placing on the queue. You should also check the metrics to see if delete requests are failing and the dequeue count on messages, which might indicate repeated failed attempts to delete the message.
- Examine the Storage Logging logs for any queue operations that have higher than expected **E2ELatency** and **ServerLatency** values over a longer period of time than usual.

Metrics show an increase in PercentThrottlingError

Throttling errors occur when you exceed the scalability targets of a storage service. The storage service does this to ensure that no single client or tenant can use the service at the expense of others. For more information, see [Azure Storage Scalability and Performance Targets](#) for details on scalability targets for storage accounts and performance targets for partitions within storage accounts.

If the **PercentThrottlingError** metric show an increase in the percentage of requests that are failing with a throttling error, you need to investigate one of two scenarios:

- [Transient increase in PercentThrottlingError](#)
- [Permanent increase in PercentThrottlingError error](#)

An increase in **PercentThrottlingError** often occurs at the same time as an increase in the number of storage requests, or when you are initially load testing your application. This may also manifest itself in the client as "503 Server Busy" or "500 Operation Timeout" HTTP status messages from storage operations.

Transient increase in PercentThrottlingError

If you are seeing spikes in the value of **PercentThrottlingError** that coincide with periods of high activity for the application, you should implement an exponential (not linear) back off strategy for retries in your client: this will reduce the immediate load on the partition and help your application to smooth out spikes in traffic. For more information about how to implement retry policies using the Storage Client Library, see [Microsoft.WindowsAzure.Storage.RetryPolicies Namespace](#) on MSDN.

NOTE

You may also see spikes in the value of **PercentThrottlingError** that do not coincide with periods of high activity for the application: the most likely cause here is the storage service moving partitions to improve load balancing.

Permanent increase in PercentThrottlingError error

If you are seeing a consistently high value for **PercentThrottlingError** following a permanent increase in your transaction volumes, or when you are performing your initial load tests on your application, then you need to evaluate how your application is using storage partitions and whether it is approaching the scalability targets for a storage account. For example, if you are seeing throttling errors on a queue (which counts as a single partition), then you should consider using additional queues to spread the transactions across multiple partitions. If you are seeing throttling errors on a table, you need to consider using a different partitioning scheme to spread your transactions across multiple partitions by using a wider range of partition key values. One common cause of this issue is the prepend/append anti-pattern where you select the date as the partition key and then all data on a particular day is written to one partition: under load, this can result in a write bottleneck. You should either consider a different partitioning design or evaluate whether using blob storage might be a better solution. You should also check if the throttling is occurring as a result of spikes in your traffic and investigate ways of smoothing your pattern of requests.

If you distribute your transactions across multiple partitions, you must still be aware of the scalability limits set for the storage account. For example, if you used ten queues each processing the maximum of 2,000 1KB messages per second, you will be at the overall limit of 20,000 messages per second for the storage account. If you need to process more than 20,000 entities per second, you should consider using multiple storage accounts. You should also bear in mind that the size of your requests and entities has an impact on when the storage service throttles your clients: if you have larger requests and entities, you may be throttled sooner.

Inefficient query design can also cause you to hit the scalability limits for table partitions. For example, a query with a filter that only selects one percent of the entities in a partition but that scans all the entities in a partition will need to access each entity. Every entity read will count towards the total number of transactions in that partition; therefore, you can easily reach the scalability targets.

NOTE

Your performance testing should reveal any inefficient query designs in your application.

Metrics show an increase in PercentTimeoutError

Your metrics show an increase in **PercentTimeoutError** for one of your storage services. At the same time, the client receives a high volume of "500 Operation Timeout" HTTP status messages from storage operations.

NOTE

You may see timeout errors temporarily as the storage service load balances requests by moving a partition to a new server.

The **PercentTimeoutError** metric is an aggregation of the following metrics: **ClientTimeoutError**, **AnonymousClientTimeoutError**, **SASClientTimeoutError**, **ServerTimeoutError**, **AnonymousServerTimeoutError**, and **SASServerTimeoutError**.

The server timeouts are caused by an error on the server. The client timeouts happen because an operation on the server has exceeded the timeout specified by the client; for example, a client using the Storage Client Library can set a timeout for an operation by using the **ServerTimeout** property of the **QueueRequestOptions** class.

Server timeouts indicate a problem with the storage service that requires further investigation. You can use metrics to see if you are hitting the scalability limits for the service and to identify any spikes in traffic that might be causing this problem. If the problem is intermittent, it may be due to load-balancing activity in the service. If the problem is persistent and is not caused by your application hitting the scalability limits of the service, you should raise a support issue. For client timeouts, you must decide if the timeout is set to an appropriate value in the client and either change the timeout value set in the client or investigate how you can improve the performance of the operations in the storage service, for example by optimizing your table queries or reducing the size of your messages.

Metrics show an increase in PercentNetworkError

Your metrics show an increase in **PercentNetworkError** for one of your storage services. The **PercentNetworkError** metric is an aggregation of the following metrics: **NetworkError**, **AnonymousNetworkError**, and **SASNetworkError**. These occur when the storage service detects a network error when the client makes a storage request.

The most common cause of this error is a client disconnecting before a timeout expires in the storage service. You should investigate the code in your client to understand why and when the client disconnects from the storage service. You can also use Wireshark, Microsoft Message Analyzer, or Tcping to investigate network connectivity issues from the client. These tools are described in the [Appendices](#).

The client is receiving HTTP 403 (Forbidden) messages

If your client application is throwing HTTP 403 (Forbidden) errors, a likely cause is that the client is using an expired Shared Access Signature (SAS) when it sends a storage request (although other possible causes include clock skew, invalid keys, and empty headers). If an expired SAS key is the cause, you will not see any entries in the server-side Storage Logging log data. The following table shows a sample from the client-side log generated by the Storage Client Library that illustrates this issue occurring:

SOURCE	VERBOSITY	VERBOSITY	CLIENT REQUEST ID	OPERATION TEXT
Microsoft.WindowsAzure.Storage	Information	3	85d077ab...	Starting operation with location Primary per location mode PrimaryOnly.
Microsoft.WindowsAzure.Storage	Information	3	85d077ab -...	Starting synchronous request to https://domemaildist.blob.core.windows.net/azureimblobcontainer/blobCreatedViaSAS.txt?sv=2014-02-14&sr=c&si=mypolicy&sig=OFnd4Rd7z01flvh%2BmcR6zbudIH2F5Ik%2FyhNYZEmJNQ%3D&api-version=2014-02-14.
Microsoft.WindowsAzure.Storage	Information	3	85d077ab -...	Waiting for response.

Source	Verbosity	Verbosity	Client Request ID	Operation Text
Microsoft.WindowsAzure.Storage	Warning	2	85d077ab -...	Exception thrown while waiting for response: The remote server returned an error: (403) Forbidden..
Microsoft.WindowsAzure.Storage	Information	3	85d077ab -...	Response received. Status code = 403, Request ID = 9d67c64a-64ed-4b0d-9515-3b14bbcd63d, Content-MD5 = , ETag = .
Microsoft.WindowsAzure.Storage	Warning	2	85d077ab -...	Exception thrown during the operation: The remote server returned an error: (403) Forbidden..
Microsoft.WindowsAzure.Storage	Information	3	85d077ab -...	Checking if the operation should be retried. Retry count = 0, HTTP status code = 403, Exception = The remote server returned an error: (403) Forbidden..
Microsoft.WindowsAzure.Storage	Information	3	85d077ab -...	The next location has been set to Primary, based on the location mode.
Microsoft.WindowsAzure.Storage	Error	1	85d077ab -...	Retry policy did not allow for a retry. Failing with The remote server returned an error: (403) Forbidden.

In this scenario, you should investigate why the SAS token is expiring before the client sends the token to the server:

- Typically, you should not set a start time when you create a SAS for a client to use immediately. If there are small clock differences between the host generating the SAS using the current time and the storage service, then it is possible for the storage service to receive a SAS that is not yet valid.
- You should not set a very short expiry time on a SAS. Again, small clock differences between the host generating the SAS and the storage service can lead to a SAS apparently expiring earlier than anticipated.
- Does the version parameter in the SAS key (for example **sv=2012-02-12**) match the version of the Storage Client Library you are using. You should always use the latest version of the Storage Client Library. For more information about SAS token versioning, see [What's new for Microsoft Azure Storage](#).
- o If you regenerate your storage access keys (click **Manage Access Keys** on any page in your storage account in the Azure Classic Portal) this can invalidate any existing SAS tokens. This may be an issue if

you generate SAS tokens with a long expiry time for client applications to cache.

If you are using the Storage Client Library to generate SAS tokens, then it is easy to build a valid token. However, if you are using the Storage REST API and constructing the SAS tokens by hand you should carefully read the topic [Delegating Access with a Shared Access Signature](#) on MSDN.

The client is receiving HTTP 404 (Not found) messages

If the client application receives an HTTP 404 (Not found) message from the server, this implies that the object the client was attempting to use (such as an entity, table, blob, container, or queue) does not exist in the storage service. There are a number of possible reasons for this, such as:

- [The client or another process previously deleted the object](#)
- [A Shared Access Signature \(SAS\) authorization issue](#)
- [Client-side JavaScript code does not have permission to access the object](#)
- [Network failure](#)

The client or another process previously deleted the object

In scenarios where the client is attempting to read, update, or delete data in a storage service it is usually easy to identify in the server-side logs a previous operation that deleted the object in question from the storage service. Very often, the log data shows that another user or process deleted the object. In the server-side Storage Logging log, the operation-type and requested-object-key columns show when a client deleted an object.

In the scenario where a client is attempting to insert an object, it may not be immediately obvious why this results in an HTTP 404 (Not found) response given that the client is creating a new object. However, if the client is creating a blob it must be able to find the blob container, if the client is creating a message it must be able to find a queue, and if the client is adding a row it must be able to find the table.

You can use the client-side log from the Storage Client Library to gain a more detailed understanding of when the client sends specific requests to the storage service.

The following client-side log generated by the Storage Client library illustrates the problem when the client cannot find the container for the blob it is creating. This log includes details of the following storage operations:

REQUEST ID	OPERATION
07b26a5d...	DeleteIfExists method to delete the blob container. Note that this operation includes a HEAD request to check for the existence of the container.
e2d06d78...	CreateIfNotExist method to create the blob container. Note that this operation includes a HEAD request that checks for the existence of the container. The HEAD returns a 404 message but continues.
de8b1c3c...	UploadFromStream method to create the blob. The PUT request fails with a 404 message

Log entries:

REQUEST ID	OPERATION TEXT
07b26a5d...	Starting synchronous request to https://domemaildist.blob.core.windows.net/azuremmblobcontainer .

REQUEST ID	OPERATION TEXT
07b26a5d-...	StringToSign = HEAD.....x-ms-client-request-id:07b26a5d-....x-ms-date:Tue, 03 Jun 2014 10:33:11 GMT.x-ms-version:2014-02-14./domemaildist/azuremmblobcontainer.restype:container.
07b26a5d-...	Waiting for response.
07b26a5d-...	Response received. Status code = 200, Request ID = eeead849-...Content-MD5 = , ETag = "0x8D14D2DC63D059B".
07b26a5d-...	Response headers were processed successfully, proceeding with the rest of the operation.
07b26a5d-...	Downloading response body.
07b26a5d-...	Operation completed successfully.
07b26a5d-...	Starting synchronous request to https://domemaildist.blob.core.windows.net/azuremmblobcontainer .
07b26a5d-...	StringToSign = DELETE.....x-ms-client-request-id:07b26a5d-....x-ms-date:Tue, 03 Jun 2014 10:33:12 GMT.x-ms-version:2014-02-14./domemaildist/azuremmblobcontainer.restype:container.
07b26a5d-...	Waiting for response.
07b26a5d-...	Response received. Status code = 202, Request ID = 6ab2a4cf-..., Content-MD5 = , ETag = .
07b26a5d-...	Response headers were processed successfully, proceeding with the rest of the operation.
07b26a5d-...	Downloading response body.
07b26a5d-...	Operation completed successfully.
e2d06d78-...	Starting asynchronous request to https://domemaildist.blob.core.windows.net/azuremmblobcontainer .
e2d06d78-...	StringToSign = HEAD.....x-ms-client-request-id:e2d06d78-....x-ms-date:Tue, 03 Jun 2014 10:33:12 GMT.x-ms-version:2014-02-14./domemaildist/azuremmblobcontainer.restype:container.
e2d06d78-...	Waiting for response.
de8b1c3c-...	Starting synchronous request to https://domemaildist.blob.core.windows.net/azuremmblobcontainer/blobCreated.txt .

REQUEST ID	OPERATION TEXT
de8b1c3c-...	StringToSign = PUT...64.qCmF+TQLPhq/YYK50mP9ZQ==x-ms-blob-type:BlockBlob.x-ms-client-request-id:de8b1c3c-....x-ms-date:Tue, 03 Jun 2014 10:33:12 GMT.x-ms-version:2014-02-14./domemaildist/azuremmblobcontainer/blobCreated.txt.
de8b1c3c-...	Preparing to write request data.
e2d06d78-...	Exception thrown while waiting for response: The remote server returned an error: (404) Not Found..
e2d06d78-...	Response received. Status code = 404, Request ID = 353ae3bc-..., Content-MD5 = , ETag = .
e2d06d78-...	Response headers were processed successfully, proceeding with the rest of the operation.
e2d06d78-...	Downloading response body.
e2d06d78-...	Operation completed successfully.
e2d06d78-...	Starting asynchronous request to https://domemaildist.blob.core.windows.net/azuremmblobcontainer .
e2d06d78-...	StringToSign = PUT...0.....x-ms-client-request-id:e2d06d78-....x-ms-date:Tue, 03 Jun 2014 10:33:12 GMT.x-ms-version:2014-02-14./domemaildist/azuremmblobcontainer.restype:container.
e2d06d78-...	Waiting for response.
de8b1c3c-...	Writing request data.
de8b1c3c-...	Waiting for response.
e2d06d78-...	Exception thrown while waiting for response: The remote server returned an error: (409) Conflict..
e2d06d78-...	Response received. Status code = 409, Request ID = c27da20e-..., Content-MD5 = , ETag = .
e2d06d78-...	Downloading error response body.
de8b1c3c-...	Exception thrown while waiting for response: The remote server returned an error: (404) Not Found..
de8b1c3c-...	Response received. Status code = 404, Request ID = 0eaeb3e-..., Content-MD5 = , ETag = .
de8b1c3c-...	Exception thrown during the operation: The remote server returned an error: (404) Not Found..

REQUEST ID	OPERATION TEXT
de8b1c3c...	Retry policy did not allow for a retry. Failing with The remote server returned an error: (404) Not Found..
e2d06d78...	Retry policy did not allow for a retry. Failing with The remote server returned an error: (409) Conflict..

In this example, the log shows that the client is interleaving requests from the **CreateIfNotExists** method (request id e2d06d78...) with the requests from the **UploadFromStream** method (de8b1c3c...); this is happening because the client application is invoking these methods asynchronously. You should modify the asynchronous code in the client to ensure that it creates the container before attempting to upload any data to a blob in that container. Ideally, you should create all your containers in advance.

A Shared Access Signature (SAS) authorization issue

If the client application attempts to use a SAS key that does not include the necessary permissions for the operation, the storage service returns an HTTP 404 (Not found) message to the client. At the same time, you will also see a non-zero value for **SASAuthorizationError** in the metrics.

The following table shows a sample server-side log message from the Storage Logging log file:

NAME	VALUE
Request start time	2014-05-30T06:17:48.4473697Z
Operation type	GetBlobProperties
Request status	SASAuthorizationError
HTTP status code	404
Authentication type	Sas
Service type	Blob
Request URL	https://domemaildist.blob.core.windows.net/azureimblobcontainer/blobCreatedViaSAS.txt
	?sv=2014-02-14&sr=c&si=mypolicy&sig=XXXXX&api-version=2014-02-14
Request id header	a1f348d5-8032-4912-93ef-b393e5252a3b
Client request ID	2d064953-8436-4ee0-aa0c-65cb874f7929

You should investigate why your client application is attempting to perform an operation it has not been granted permissions for.

Client-side JavaScript code does not have permission to access the object

If you are using a JavaScript client and the storage service is returning HTTP 404 messages, you check for the following JavaScript errors in the browser:

```
SEC7120: Origin http://localhost:56309 not found in Access-Control-Allow-Origin header.
SCRIPT7002: XMLHttpRequest: Network Error 0x80070005, Access is denied.
```

NOTE

You can use the F12 Developer Tools in Internet Explorer to trace the messages exchanged between the browser and the storage service when you are troubleshooting client-side JavaScript issues.

These errors occur because the web browser implements the [same-origin policy](#) security restriction that prevents a web page from calling an API in a different domain from the domain the page comes from.

To work around the JavaScript issue, you can configure Cross Origin Resource Sharing (CORS) for the storage service the client is accessing. For more information, see [Cross-Origin Resource Sharing \(CORS\) Support for Azure Storage Services](#) on MSDN.

The following code sample shows how to configure your blob service to allow JavaScript running in the Contoso domain to access a blob in your blob storage service:

```
CloudBlobClient client = new CloudBlobClient(blobEndpoint, new StorageCredentials(accountName, accountKey));
// Set the service properties.
ServiceProperties sp = client.GetServiceProperties();
sp.DefaultServiceVersion = "2013-08-15";
CorsRule cr = new CorsRule();
cr.AllowedHeaders.Add("*");
cr.AllowedMethods = CorsHttpMethods.Get | CorsHttpMethods.Put;
cr.AllowedOrigins.Add("http://www.contoso.com");
cr.ExposedHeaders.Add("x-ms-*");
cr.MaxAgeInSeconds = 5;
sp.Cors.CorsRules.Clear();
sp.Cors.CorsRules.Add(cr);
client.SetServiceProperties(sp);
```

Network Failure

In some circumstances, lost network packets can lead to the storage service returning HTTP 404 messages to the client. For example, when your client application is deleting an entity from the table service you see the client throw a storage exception reporting an "HTTP 404 (Not Found)" status message from the table service. When you investigate the table in the table storage service, you see that the service did delete the entity as requested.

The exception details in the client include the request id (7e84f12d...) assigned by the table service for the request: you can use this information to locate the request details in the server-side storage logs by searching in the **request-id-header** column in the log file. You could also use the metrics to identify when failures such as this occur and then search the log files based on the time the metrics recorded this error. This log entry shows that the delete failed with an "HTTP (404) Client Other Error" status message. The same log entry also includes the request id generated by the client in the **client-request-id** column (813ea74f...).

The server-side log also includes another entry with the same **client-request-id** value (813ea74f...) for a successful delete operation for the same entity, and from the same client. This successful delete operation took place very shortly before the failed delete request.

The most likely cause of this scenario is that the client sent a delete request for the entity to the table service, which succeeded, but did not receive an acknowledgement from the server (perhaps due to a temporary network issue). The client then automatically retried the operation (using the same **client-request-id**), and this retry failed because the entity had already been deleted.

If this problem occurs frequently, you should investigate why the client is failing to receive acknowledgements from the table service. If the problem is intermittent, you should trap the "HTTP (404) Not Found" error and log it in the client, but allow the client to continue.

The client is receiving HTTP 409 (Conflict) messages

The following table shows an extract from the server-side log for two client operations: **DeleteIfExists** followed

immediately by **CreateIfNotExists** using the same blob container name. Note that each client operation results in two requests sent to the server, first a **GetContainerProperties** request to check if the container exists, followed by the **DeleteContainer** or **CreateContainer** request.

TIMESTAMP	OPERATION	RESULT	CONTAINER NAME	CLIENT REQUEST ID
05:10:13.7167225	GetContainerProperties	200	mmcont	c9f52c89...
05:10:13.8167325	DeleteContainer	202	mmcont	c9f52c89...
05:10:13.8987407	GetContainerProperties	404	mmcont	bc881924...
05:10:14.2147723	CreateContainer	409	mmcont	bc881924...

The code in the client application deletes and then immediately recreates a blob container using the same name: the **CreateIfNotExists** method (Client request ID bc881924...) eventually fails with the HTTP 409 (Conflict) error. When a client deletes blob containers, tables, or queues there is a brief period before the name becomes available again.

The client application should use unique container names whenever it creates new containers if the delete/recreate pattern is common.

Metrics show low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors

The **PercentSuccess** metric captures the percent of operations that were successful based on their HTTP Status Code. Operations with status codes of 2XX count as successful, whereas operations with status codes in 3XX, 4XX and 5XX ranges are counted as unsuccessful and lower the **PercentSuccess** metric value. In the server-side storage log files, these operations are recorded with a transaction status of **ClientOtherErrors**.

It is important to note that these operations have completed successfully and therefore do not affect other metrics such as availability. Some examples of operations that execute successfully but that can result in unsuccessful HTTP status codes include:

- **ResourceNotFound** (Not Found 404), for example from a GET request to a blob that does not exist.
- **ResouceAlreadyExists** (Conflict 409), for example from a **CreateIfNotExist** operation where the resource already exists.
- **ConditionNotMet** (Not Modified 304), for example from a conditional operation such as when a client sends an **ETag** value and an HTTP **If-None-Match** header to request an image only if it has been updated since the last operation.

You can find a list of common REST API error codes that the storage services return on the page [Common REST API Error Codes](#).

Capacity metrics show an unexpected increase in storage capacity usage

If you see sudden, unexpected changes in capacity usage in your storage account, you can investigate the reasons by first looking at your availability metrics; for example, an increase in the number of failed delete requests might lead to an increase in the amount of blob storage you are using as application specific cleanup operations you might have expected to be freeing up space may not be working as expected (for example, because the SAS tokens used for freeing up space have expired).

You are experiencing unexpected reboots of Azure Virtual Machines that have a large number of attached VHDs

If an Azure Virtual Machine (VM) has a large number of attached VHDs that are in the same storage account, you could exceed the scalability targets for an individual storage account causing the VM to fail. You should check the

minute metrics for the storage account (**TotalRequests**/**TotalIngress**/**TotalEgress**) for spikes that exceed the scalability targets for a storage account. See the section "[Metrics show an increase in PercentThrottlingError](#)" for assistance in determining if throttling has occurred on your storage account.

In general, each individual input or output operation on a VHD from a Virtual Machine translates to [Get Page](#) or [Put Page](#) operations on the underlying page blob. Therefore, you can use the estimated IOPS for your environment to tune how many VHDs you can have in a single storage account based on the specific behavior of your application. We do not recommend having more than 40 disks in a single storage account. See [Azure Storage Scalability and Performance Targets](#) for details of the current scalability targets for storage accounts, in particular the total request rate and total bandwidth for the type of storage account you are using. If you are exceeding the scalability targets for your storage account, you should place your VHDs in multiple different storage accounts to reduce the activity in each individual account.

Your issue arises from using the storage emulator for development or test

You typically use the storage emulator during development and test to avoid the requirement for an Azure storage account. The common issues that can occur when you are using the storage emulator are:

- [Feature "X" is not working in the storage emulator](#)
- [Error "The value for one of the HTTP headers is not in the correct format" when using the storage emulator](#)
- [Running the storage emulator requires administrative privileges](#)

Feature "X" is not working in the storage emulator

The storage emulator does not support all of the features of the Azure storage services such as the file service. For more information, see [Differences Between the Storage Emulator and Azure Storage Services](#) on MSDN.

For those features that the storage emulator does not support, use the Azure storage service in the cloud.

Error "The value for one of the HTTP headers is not in the correct format" when using the storage emulator

You are testing your application that use the Storage Client Library against the local storage emulator and method calls such as **CreateIfNotExists** fail with the error message "The value for one of the HTTP headers is not in the correct format." This indicates that the version of the storage emulator you are using does not support the version of the storage client library you are using. The Storage Client Library adds the header **x-ms-version** to all the requests it makes. If the storage emulator does not recognize the value in the **x-ms-version** header, it rejects the request.

You can use the Storage Library Client logs to see the value of the **x-ms-version header** it is sending. You can also see the value of the **x-ms-version header** if you use Fiddler to trace the requests from your client application.

This scenario typically occurs if you install and use the latest version of the Storage Client Library without updating the storage emulator. You should either install the latest version of the storage emulator, or use cloud storage instead of the emulator for development and test.

Running the storage emulator requires administrative privileges

You are prompted for administrator credentials when you run the storage emulator. This only occurs when you are initializing the storage emulator for the first time. After you have initialized the storage emulator, you do not need administrative privileges to run it again.

For more information, see [Initialize the Storage Emulator by Using the Command-Line Tool](#) on MSDN (you can also initialize the storage emulator in Visual Studio, which will also require administrative privileges).

You are encountering problems installing the Azure SDK for .NET

When you try to install the SDK, it fails trying to install the storage emulator on your local machine. The installation log contains one of the following messages:

- CAQuietExec: Error: Unable to access SQL instance
- CAQuietExec: Error: Unable to create database

The cause is an issue with existing LocalDB installation. By default, the storage emulator uses LocalDB to persist data when it simulates the Azure storage services. You can reset your LocalDB instance by running the following commands in a command-prompt window before trying to install the SDK.

```
sqllocaldb stop v11.0
sqllocaldb delete v11.0
delete %USERPROFILE%\WAStorageEmulatorDb3*.*
sqllocaldb create v11.0
```

The **delete** command removes any old database files from previous installations of the storage emulator.

You have a different issue with a storage service

If the previous troubleshooting sections do not include the issue you are having with a storage service, you should adopt the following approach to diagnosing and troubleshooting your issue.

- Check your metrics to see if there is any change from your expected base-line behavior. From the metrics, you may be able to determine whether the issue is transient or permanent, and which storage operations the issue is affecting.
- You can use the metrics information to help you search your server-side log data for more detailed information about any errors that are occurring. This information may help you troubleshoot and resolve the issue.
- If the information in the server-side logs is not sufficient to troubleshoot the issue successfully, you can use the Storage Client Library client-side logs to investigate the behavior of your client application, and tools such as Fiddler, Wireshark, and Microsoft Message Analyzer to investigate your network.

For more information about using Fiddler, see "[Appendix 1: Using Fiddler to capture HTTP and HTTPS traffic](#)."

For more information about using Wireshark, see "[Appendix 2: Using Wireshark to capture network traffic](#)."

For more information about using Microsoft Message Analyzer, see "[Appendix 3: Using Microsoft Message Analyzer to capture network traffic](#)."

Appendices

The appendices describe several tools that you may find useful when you are diagnosing and troubleshooting issues with Azure Storage (and other services). These tools are not part of Azure Storage and some are third-party products. As such, the tools discussed in these appendices are not covered by any support agreement you may have with Microsoft Azure or Azure Storage, and therefore as part of your evaluation process you should examine the licensing and support options available from the providers of these tools.

Appendix 1: Using Fiddler to capture HTTP and HTTPS traffic

Fiddler is a useful tool for analyzing the HTTP and HTTPS traffic between your client application and the Azure storage service you are using. You can download Fiddler from <http://www.telerik.com/fiddler>.

NOTE

Fiddler can decode HTTPS traffic; you should read the Fiddler documentation carefully to understand how it does this, and to understand the security implications.

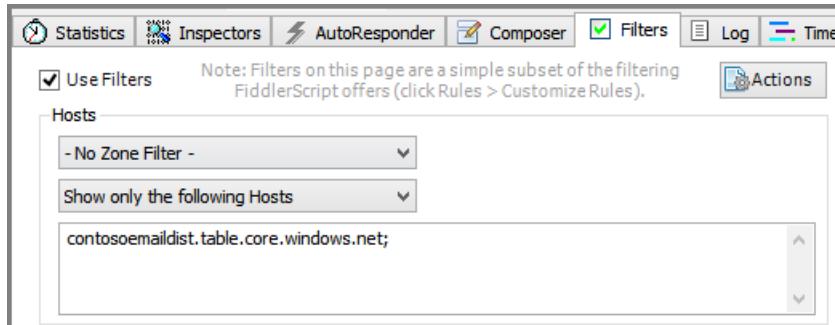
This appendix provides a brief walkthrough of how to configure Fiddler to capture traffic between the local machine where you have installed Fiddler and the Azure storage services.

After you have launched Fiddler, it will begin capturing HTTP and HTTPS traffic on your local machine. The following are some useful commands for controlling Fiddler:

- Stop and start capturing traffic. On the main menu, go to **File** and then click **Capture Traffic** to toggle capturing on and off.

- Save captured traffic data. On the main menu, go to **File**, click **Save**, and then click **All Sessions**: this enables you to save the traffic in a Session Archive file. You can reload a Session Archive later for analysis, or send it if requested to Microsoft support.

To limit the amount of traffic that Fiddler captures, you can use filters that you configure in the **Filters** tab. The following screenshot shows a filter that captures only traffic sent to the **contosoemaildist.table.core.windows.net** storage endpoint:

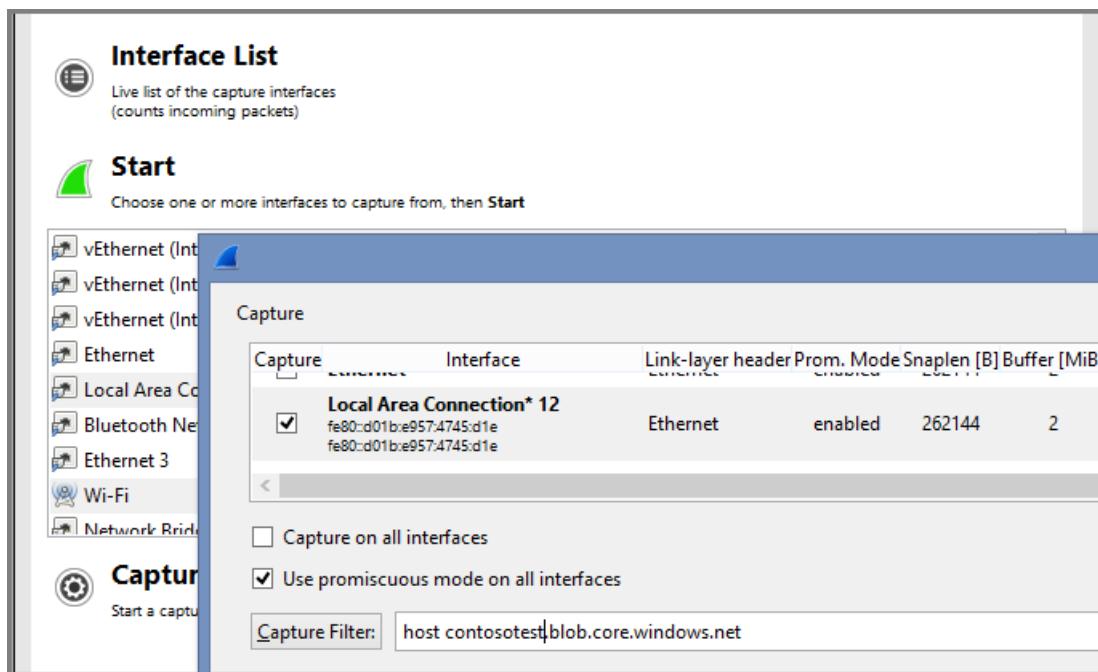


Appendix 2: Using Wireshark to capture network traffic

Wireshark is a network protocol analyzer that enables you to view detailed packet information for a wide range of network protocols. You can download Wireshark from <http://www.wireshark.org/>.

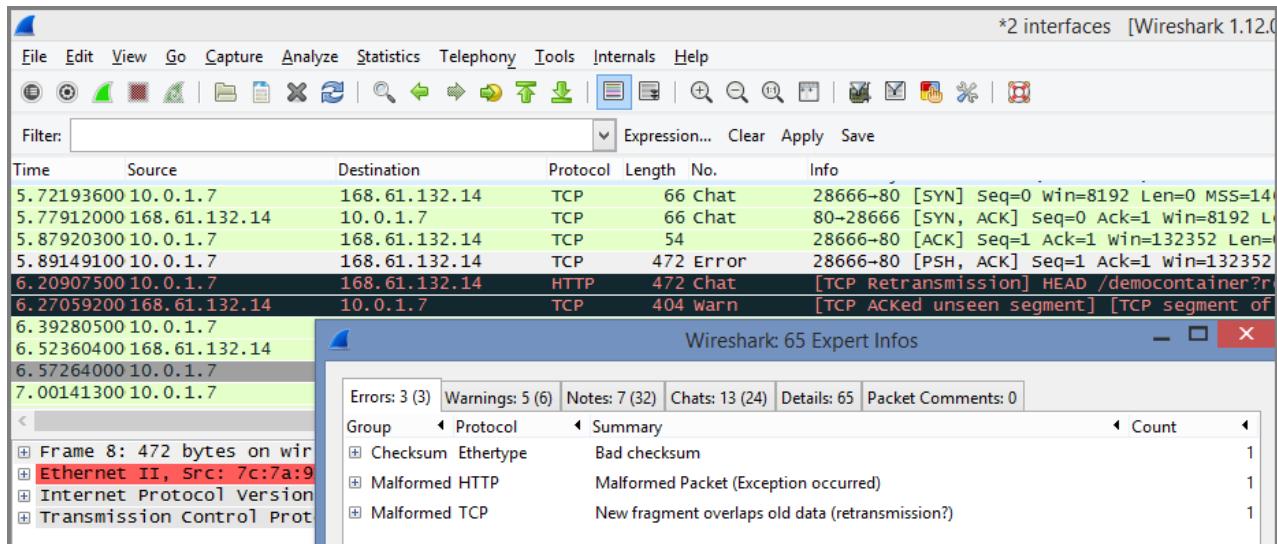
The following procedure shows you how to capture detailed packet information for traffic from the local machine where you installed Wireshark to the table service in your Azure storage account.

- Launch Wireshark on your local machine.
- In the **Start** section, select the local network interface or interfaces that are connected to the internet.
- Click **Capture Options**.
- Add a filter to the **Capture Filter** textbox. For example, **host contosoemaildist.table.core.windows.net** will configure Wireshark to capture only packets sent to or from the table service endpoint in the **contosoemaildist** storage account. For a complete list of Capture Filters see <http://wiki.wireshark.org/CaptureFilters>.

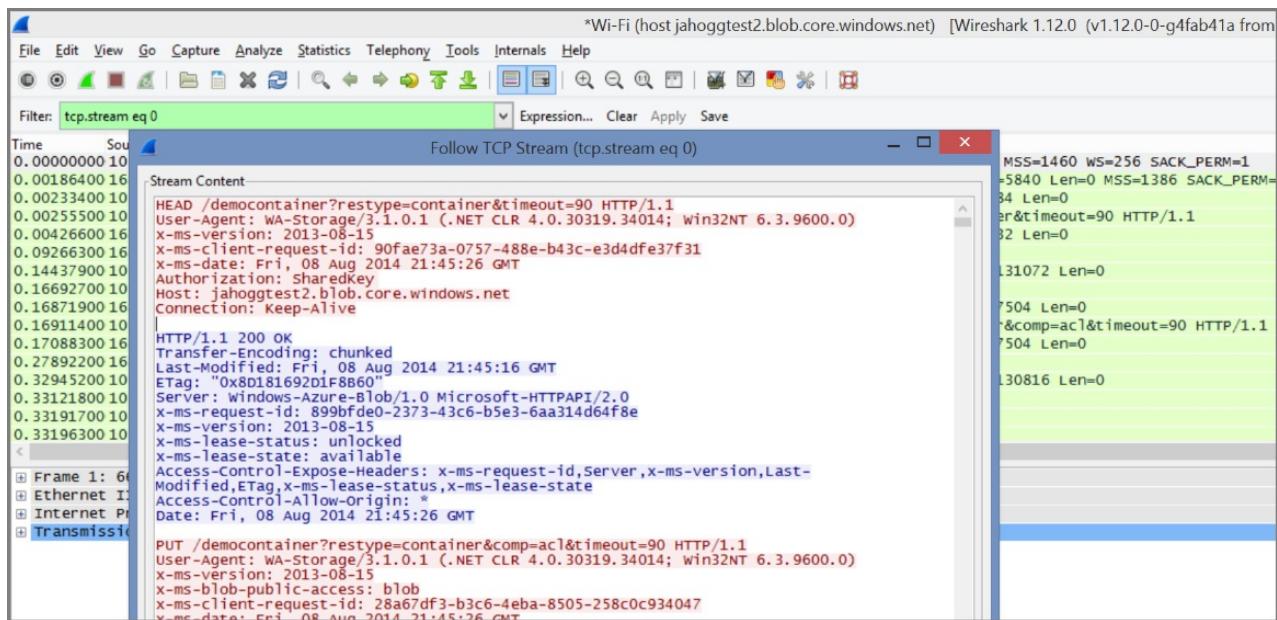


- Click **Start**. Wireshark will now capture all the packets sent to or from the table service endpoint as you use your client application on your local machine.
- When you have finished, on the main menu click **Capture** and then **Stop**.
- To save the captured data in a Wireshark Capture File, on the main menu click **File** and then **Save**.

WireShark will highlight any errors that exist in the **packetlist** window. You can also use the **Expert Info** window (click **Analyze**, then **Expert Info**) to view a summary of errors and warnings.



You can also choose to view the TCP data as the application layer sees it by right-clicking on the TCP data and selecting **Follow TCP Stream**. This is particularly useful if you captured your dump without a capture filter. See [here](#) for more information.



NOTE

For more information about using Wireshark, see the [Wireshark Users Guide](#).

Appendix 3: Using Microsoft Message Analyzer to capture network traffic

You can use Microsoft Message Analyzer to capture HTTP and HTTPS traffic in a similar way to Fiddler, and capture network traffic in a similar way to Wireshark.

Configure a web tracing session using Microsoft Message Analyzer

To configure a web tracing session for HTTP and HTTPS traffic using Microsoft Message Analyzer, run the Microsoft Message Analyzer application and then on the **File** menu, click **Capture/Trace**. In the list of available trace scenarios, select **Web Proxy**. Then in the **Trace Scenario Configuration** panel, in the **HostnameFilter** textbox, add the names of your storage endpoints (you can look up these names in the Azure Classic Portal). For example, if the name of your Azure storage account is **contosodata**, you should add the following to the **HostnameFilter**

textbox:

```
contosodata.blob.core.windows.net contosodata.table.core.windows.net contosodata.queue.core.windows.net
```

NOTE

A space character separates the hostnames.

When you are ready to start collecting trace data, click the **Start With** button.

For more information about the Microsoft Message Analyzer **Web Proxy** trace, see [PEF-WebProxy Provider](#) on TechNet.

The built-in **Web Proxy** trace in Microsoft Message Analyzer is based on Fiddler; it can capture client-side HTTPS traffic and display unencrypted HTTPS messages. The **Web Proxy** trace works by configuring a local proxy for all HTTP and HTTPS traffic that gives it access to unencrypted messages.

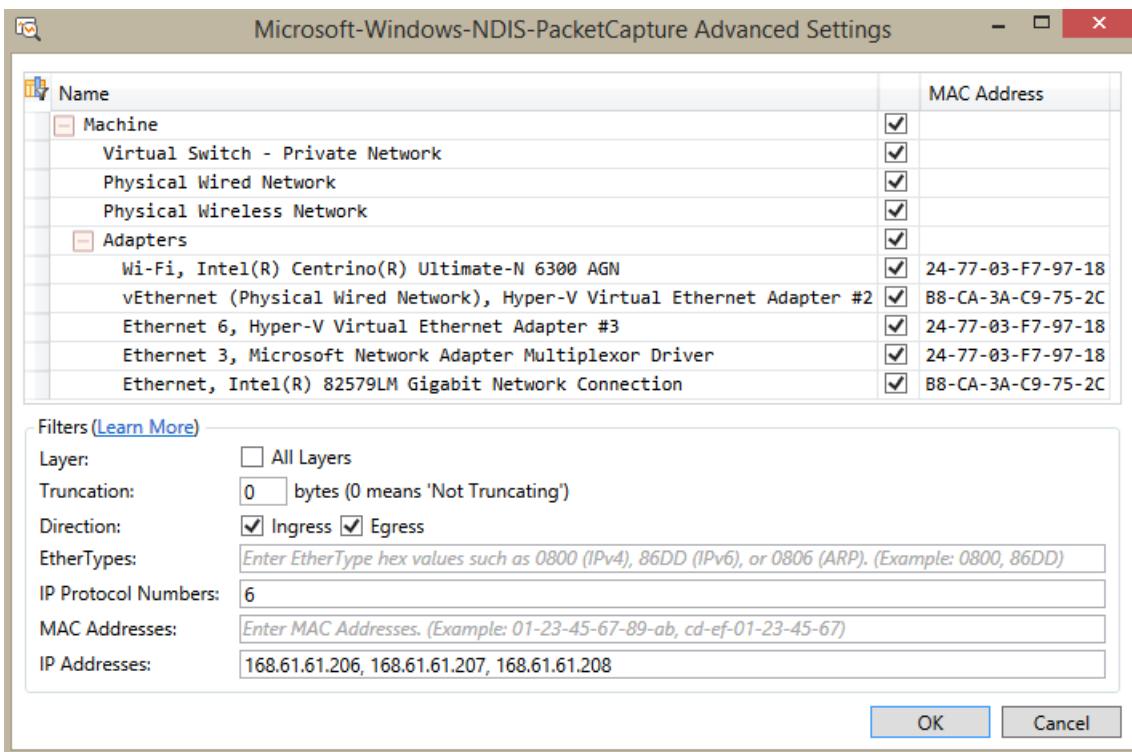
Diagnosing network issues using Microsoft Message Analyzer

In addition to using the Microsoft Message Analyzer **Web Proxy** trace to capture details of the HTTP/HTTPs traffic between the client application and the storage service, you can also use the built-in **Local Link Layer** trace to capture network packet information. This enables you to capture data similar to that which you can capture with Wireshark, and diagnose network issues such as dropped packets.

The following screenshot shows an example **Local Link Layer** trace with some **informational** messages in the **DiagnosisTypes** column. Clicking on an icon in the **DiagnosisTypes** column shows the details of the message. In this example, the server retransmitted message #305 because it did not receive an acknowledgement from the client:

MessageNumber	Type	Level	Message
302	i	Application	TCP: Retransmitted, original message is #305.
303		Warning	
304	i	Application	
305		Warning	
306	i	Application	
307	i	Application	

When you create the trace session in Microsoft Message Analyzer, you can specify filters to reduce the amount of noise in the trace. On the **Capture / Trace** page where you define the trace, click on the **Configure** link next to **Microsoft-Windows-NDIS-PacketCapture**. The following screenshot shows a configuration that filters TCP traffic for the IP addresses of three storage services:



For more information about the Microsoft Message Analyzer Local Link Layer trace, see [PEF-NDIS-PacketCapture Provider](#) on TechNet.

Appendix 4: Using Excel to view metrics and log data

Many tools enable you to download the Storage Metrics data from Azure table storage in a delimited format that makes it easy to load the data into Excel for viewing and analysis. Storage Logging data from Azure blob storage is already in a delimited format that you can load into Excel. However, you will need to add appropriate column headings based in the information at [Storage Analytics Log Format](#) and [Storage Analytics Metrics Table Schema](#).

To import your Storage Logging data into Excel after you download it from blob storage:

- On the **Data** menu, click **From Text**.
- Browse to the log file you want to view and click **Import**.
- On step 1 of the **Text Import Wizard**, select **Delimited**.

On step 1 of the **Text Import Wizard**, select **Semicolon** as the only delimiter and choose double-quote as the **Text qualifier**. Then click **Finish** and choose where to place the data in your workbook.

Appendix 5: Monitoring with Application Insights for Visual Studio Team Services

You can also use the Application Insights feature for Visual Studio Team Services as part of your performance and availability monitoring. This tool can:

- Make sure your web service is available and responsive. Whether your app is a web site or a device app that uses a web service, it can test your URL every few minutes from locations around the world, and let you know if there's a problem.
- Quickly diagnose any performance issues or exceptions in your web service. Find out if CPU or other resources are being stretched, get stack traces from exceptions, and easily search through log traces. If the app's performance drops below acceptable limits, we can send you an email. You can monitor both .NET and Java web services.

At the time of writing Application Insights is in preview. You can find more information at [Application Insights for Visual Studio Team Services on MSDN](#).

End-to-End Troubleshooting using Azure Storage Metrics and Logging, AzCopy, and Message Analyzer

1/17/2017 • 25 min to read • [Edit on GitHub](#)

Overview

Diagnosing and troubleshooting is a key skill for building and supporting client applications with Microsoft Azure Storage. Due to the distributed nature of an Azure application, diagnosing and troubleshooting errors and performance issues may be more complex than in traditional environments.

In this tutorial, we will demonstrate how to identify client certain errors that may affect performance, and troubleshoot those errors from end-to-end using tools provided by Microsoft and Azure Storage, in order to optimize the client application.

This tutorial provides a hands-on exploration of an end-to-end troubleshooting scenario. For an in-depth conceptual guide to troubleshooting Azure storage applications, see [Monitor, diagnose, and troubleshoot Microsoft Azure Storage](#).

Tools for troubleshooting Azure Storage applications

To troubleshoot client applications using Microsoft Azure Storage, you can use a combination of tools to determine when an issue has occurred and what the cause of the problem may be. These tools include:

- **Azure Storage Analytics.** [Azure Storage Analytics](#) provides metrics and logging for Azure Storage.
 - **Storage metrics** tracks transaction metrics and capacity metrics for your storage account. Using metrics, you can determine how your application is performing according to a variety of different measures. See [Storage Analytics Metrics Table Schema](#) for more information about the types of metrics tracked by Storage Analytics.
 - **Storage logging** logs each request to the Azure Storage services to a server-side log. The log tracks detailed data for each request, including the operation performed, the status of the operation, and latency information. See [Storage Analytics Log Format](#) for more information about the request and response data that is written to the logs by Storage Analytics.

NOTE

Storage accounts with a replication type of Zone-Redundant Storage (ZRS) do not have the metrics or logging capability enabled at this time.

- **Azure Classic Portal.** You can configure metrics and logging for your storage account in the [Azure Classic Portal](#). You can also view charts and graphs that show how your application is performing over time, and configure alerts to notify you if your application performs differently than expected for a specified metric.

See [Monitor a storage account in the Azure Portal](#) for information about configuring monitoring in the Azure Classic Portal.

- **AzCopy.** Server logs for Azure Storage are stored as blobs, so you can use AzCopy to copy the log blobs to a local directory for analysis using Microsoft Message Analyzer. See [Transfer data with the AzCopy Command-Line Utility](#) for more information about AzCopy.
- **Microsoft Message Analyzer.** Message Analyzer is a tool that consumes log files and displays log data in a

visual format that makes it easy to filter, search, and group log data into useful sets that you can use to analyze errors and performance issues. See [Microsoft Message Analyzer Operating Guide](#) for more information about Message Analyzer.

About the sample scenario

For this tutorial, we'll examine a scenario where Azure Storage metrics indicates a low percent success rate for an application that calls Azure storage. The low percent success rate metric (shown as **PercentSuccess** in the Azure Classic Portal and in the metrics tables) tracks operations that succeed, but that return an HTTP status code that is greater than 299. In the server-side storage log files, these operations are recorded with a transaction status of **ClientOtherErrors**. For more details about the low percent success metric, see [Metrics show low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors](#).

Azure Storage operations may return HTTP status codes greater than 299 as part of their normal functionality. But these errors in some cases indicate that you may be able to optimize your client application for improved performance.

In this scenario, we'll consider a low percent success rate to be anything below 100%. You can choose a different metric level, however, according to your needs. We recommend that during testing of your application, you establish a baseline tolerance for your key performance metrics. For example, you might determine, based on testing, that your application should have a consistent percent success rate of 90%, or 85%. If your metrics data shows that the application is deviating from that number, then you can investigate what may be causing the increase.

For our sample scenario, once we've established that the percent success rate metric is below 100%, we will examine the logs to find the errors that correlate to the metrics, and use them to figure out what is causing the lower percent success rate. We'll look specifically at errors in the 400 range. Then we'll more closely investigate 404 (Not Found) errors.

Some causes of 400-range errors

The examples below shows a sampling of some 400-range errors for requests against Azure Blob Storage, and their possible causes. Any of these errors, as well as errors in the 300 range and the 500 range, can contribute to a low percent success rate.

Note that the lists below are far from complete. See [Status and Error Codes](#) for details about general Azure Storage errors and about errors specific to each of the storage services.

Status Code 404 (Not Found) Examples

Occurs when a read operation against a container or blob fails because the blob or container is not found.

- Occurs if a container or blob has been deleted by another client before this request.
- Occurs if you are using an API call that creates the container or blob after checking whether it exists. The `CreateIfNotExists` APIs make a HEAD call first to check for the existence of the container or blob; if it does not exist, a 404 error is returned, and then a second PUT call is made to write the container or blob.

Status Code 409 (Conflict) Examples

- Occurs if you use a Create API to create a new container or blob, without checking for existence first, and a container or blob with that name already exists.
- Occurs if a container is being deleted, and you attempt to create a new container with the same name before the deletion operation is complete.
- Occurs if you specify a lease on a container or blob, and there is already a lease present.

Status Code 412 (Precondition Failed) Examples

- Occurs when the condition specified by a conditional header has not been met.

- Occurs when the lease ID specified does not match the lease ID on the container or blob.

Generate log files for analysis

In this tutorial, we'll use Message Analyzer to work with three different types of log files, although you could choose to work with any one of these:

- The **server log**, which is created when you enable Azure Storage logging. The server log contains data about each operation called against one of the Azure Storage services - blob, queue, table, and file. The server log indicates which operation was called and what status code was returned, as well as other details about the request and response.
- The **.NET client log**, which is created when you enable client-side logging from within your .NET application. The client log includes detailed information about how the client prepares the request and receives and processes the response.
- The **HTTP network trace log**, which collects data on HTTP/HTTPS request and response data, including for operations against Azure Storage. In this tutorial, we'll generate the network trace via Message Analyzer.

Configure server-side logging and metrics

First, we'll need to configure Azure Storage logging and metrics, so that we have data from the client application to analyze. You can configure logging and metrics in a variety of ways - via the [Azure Classic Portal](#), by using PowerShell, or programmatically. See [Enabling Storage Metrics and Viewing Metrics Data](#) and [Enabling Storage Logging and Accessing Log Data](#) for details about configuring logging and metrics.

Via the Azure Classic Portal

To configure logging and metrics for your storage account using the portal, follow the instructions at [Monitor a storage account in the Azure Portal](#).

NOTE

It's not possible to set minute metrics using the Azure Classic Portal. However, we recommend that you do set them for the purposes of this tutorial, and for investigating performance issues with your application. You can set minute metrics using PowerShell as shown below, or programmatically, or via the Azure Classic Portal.

Note that the Azure Classic Portal cannot display minute metrics, only hourly metrics.

Via PowerShell

To get started with PowerShell for Azure, see [How to install and configure Azure PowerShell](#).

1. Use the [Add-AzureAccount](#) cmdlet to add your Azure user account to the PowerShell window:

```
Add-AzureAccount
```

2. In the **Sign in to Microsoft Azure** window, type the email address and password associated with your account. Azure authenticates and saves the credential information, and then closes the window.
3. Set the default storage account to the storage account you are using for the tutorial by executing these commands in the PowerShell window:

```
$SubscriptionName = 'Your subscription name'  
$StorageAccountName = 'yourstorageaccount'  
Set-AzureSubscription -CurrentStorageAccountName $StorageAccountName -SubscriptionName $SubscriptionName
```

4. Enable storage logging for the Blob service:

```
Set-AzureStorageServiceLoggingProperty -ServiceType Blob -LoggingOperations Read,Write,Delete -PassThru -RetentionDays 7 -Version 1.0
```

5. Enable storage metrics for the Blob service, making sure to set **-MetricsType** to **Minute**:

```
Set-AzureStorageServiceMetricsProperty -ServiceType Blob -MetricsType Minute -MetricsLevel ServiceAndApi -PassThru -RetentionDays 7 -Version 1.0
```

Configure .NET client-side logging

To configure client-side logging for a .NET application, enable .NET diagnostics in the application's configuration file (web.config or app.config). See [Client-side Logging with the .NET Storage Client Library](#) and [Client-side Logging with the Microsoft Azure Storage SDK for Java](#) for details.

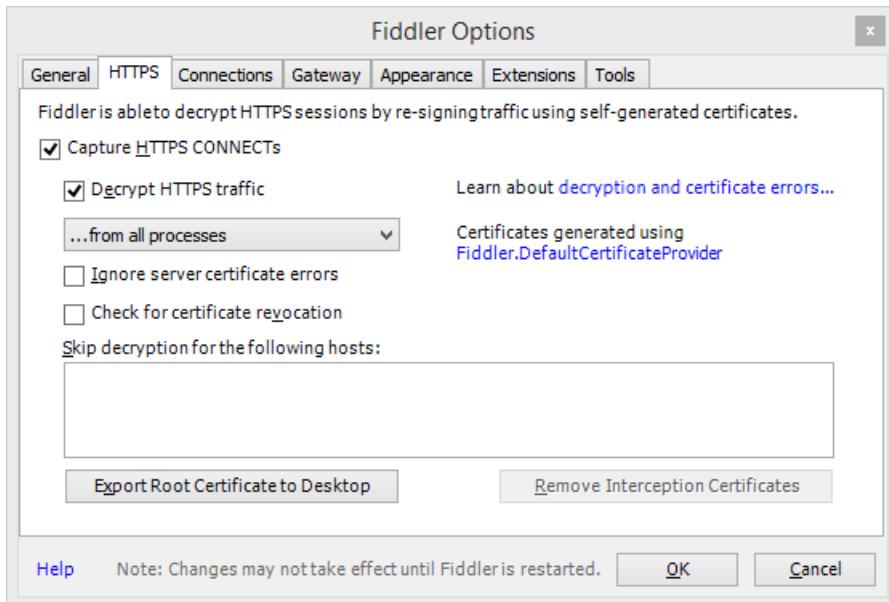
The client-side log includes detailed information about how the client prepares the request and receives and processes the response.

The Storage Client Library stores client-side log data in the location specified in the application's configuration file (web.config or app.config).

Collect a network trace

You can use Message Analyzer to collect an HTTP/HTTPS network trace while your client application is running. Message Analyzer uses [Fiddler](#) on the back end. Before you collect the network trace, we recommend that you configure Fiddler to record unencrypted HTTPS traffic:

1. Install [Fiddler](#).
2. Launch Fiddler.
3. Select **Tools | Fiddler Options**.
4. In the Options dialog, ensure that **Capture HTTPS CONNECTs** and **Decrypt HTTPS Traffic** are both selected, as shown below.



For the tutorial, collect and save a network trace first in Message Analyzer, then create an analysis session to analyze the trace and the logs. To collect a network trace in Message Analyzer:

1. In Message Analyzer, select **File | Quick Trace | Unencrypted HTTPS**.
2. The trace will begin immediately. Select **Stop** to stop the trace so that we can configure it to trace storage traffic only.
3. Select **Edit** to edit the tracing session.

4. Select the **Configure** link to the right of the **Microsoft-Pef-WebProxy** ETW provider.
5. In the **Advanced Settings** dialog, click the **Provider** tab.
6. In the **Hostname Filter** field, specify your storage endpoints, separated by spaces. For example, you can specify your endpoints as follows; change `storagesample` to the name of your storage account:

```
storagesample.blob.core.windows.net storagesample.queue.core.windows.net  
storagesample.table.core.windows.net
```

7. Exit the dialog, and click **Restart** to begin collecting the trace with the hostname filter in place, so that only Azure Storage network traffic is included in the trace.

NOTE

After you have finished collecting your network trace, we strongly recommend that you revert the settings that you may have changed in Fiddler to decrypt HTTPS traffic. In the Fiddler Options dialog, deselect the **Capture HTTPS CONNECTs** and **Decrypt HTTPS Traffic** checkboxes.

See [Using the Network Tracing Features](#) on Technet for more details.

Review metrics data in the Azure Classic Portal

Once your application has been running for a period of time, you can review the metrics charts that appear in the Azure Classic Portal to observe how your service has been performing. First, we'll add the **Success Percentage** metric to the Monitoring page:

1. Navigate to the Dashboard for your storage account in the [Azure Classic Portal](#), then select **Monitor** to view the monitoring page.
2. Click **Add Metrics** to display the **Choose Metrics** dialog.
3. Scroll down to find the **Success Percentage** group, expand it, then select **Aggregate**, as shown in the picture below. This metric aggregates success percentage data from all Blob operations.

CHOOSE METRICS

Select Metrics to Monitor

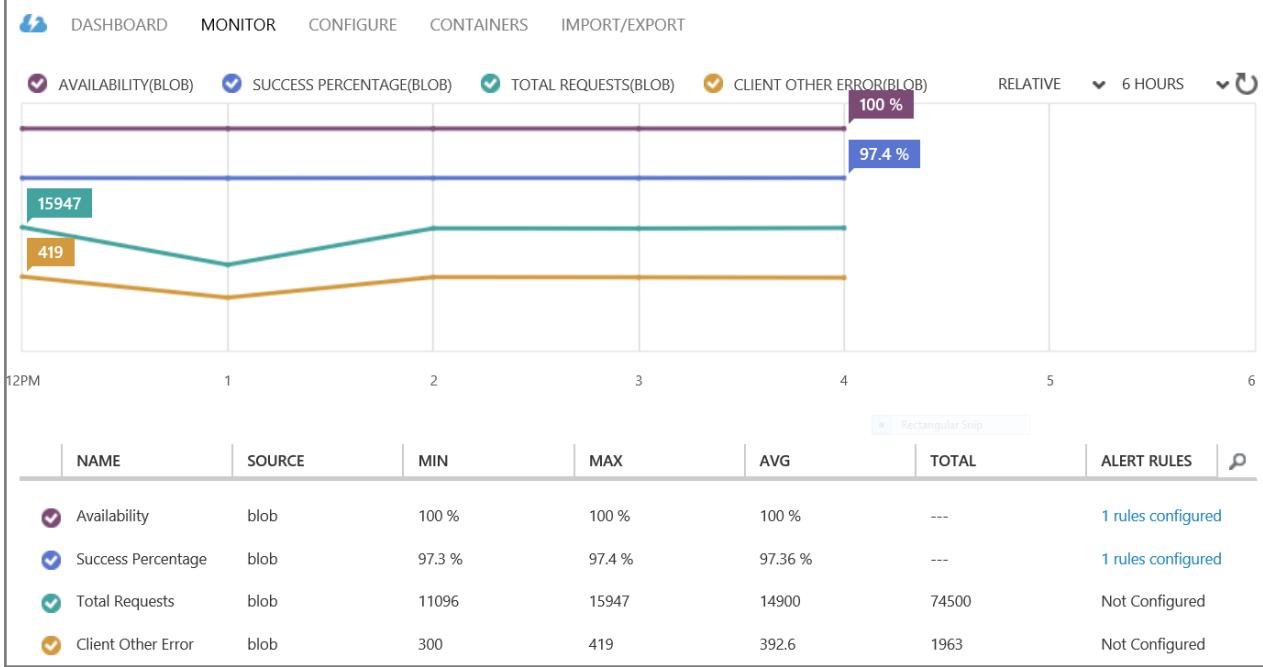
BLOBS TABLES QUEUES

NAME	UNIT
▶ SERVER TIMEOUT ERROR	
▶ SUCCESS	
◀ SUCCESS PERCENTAGE	
<input type="checkbox"/> AcquireLease	%
<input checked="" type="checkbox"/> Aggregate	%
<input type="checkbox"/> BreakLease	%
<input type="checkbox"/> ClearPage	%
<input type="checkbox"/> CopyBlob	%
<input type="checkbox"/> CreateContainer	%
<input type="checkbox"/> DeleteBlob	%
<input type="checkbox"/> DeleteContainer	%
<input type="checkbox"/> GetBlob	%

(✓)

In the Azure Classic Portal, you'll now see **Success Percentage** in the monitoring chart, along with any other metrics you may have added (up to six can be displayed on the chart at once). In the picture below, you can see that the percent success rate is somewhat below 100%, which is the scenario we'll investigate next by analyzing the logs in Message Analyzer:

storagesample



For more details on adding metrics to the Monitoring page, see [How to: Add metrics to the metrics table](#).

NOTE

It may take some time for your metrics data to appear in the Azure Classic Portal after you enable storage metrics. This is because hourly metrics for the previous hour are not displayed in the Azure Classic Portal until the current hour has elapsed. Also, minute metrics are not displayed in the Azure Classic Portal. So depending on when you enable metrics, it may take up to two hours to see metrics data.

Use AzCopy to copy server logs to a local directory

Azure Storage writes server log data to blobs, while metrics are written to tables. Log blobs are available in the well-known `$logs` container for your storage account. Log blobs are named hierarchically by year, month, day, and hour, so that you can easily locate the range of time you wish to investigate. For example, in the `storagesample` account, the container for the log blobs for 01/02/2015, from 8-9 am, is

`https://storagesample.blob.core.windows.net/$logs/blob/2015/01/08/0800`. The individual blobs in this container are named sequentially, beginning with `000000.log`.

You can use the AzCopy command-line tool to download these server-side log files to a location of your choice on your local machine. For example, you can use the following command to download the log files for blob operations that took place on January 2, 2015 to the folder `C:\Temp\Logs\Server`; replace `<storageaccountname>` with the name of your storage account, and `<storageaccountkey>` with your account access key:

```
AzCopy.exe /Source:http://<storageaccountname>.blob.core.windows.net/$logs /Dest:C:\Temp\Logs\Server  
/Pattern:"blob/2015/01/02" /SourceKey:<storageaccountkey> /S /V
```

AzCopy is available for download on the [Azure Downloads](#) page. For details about using AzCopy, see [Transfer data with the AzCopy Command-Line Utility](#).

For additional information about downloading server-side logs, see [Downloading Storage Logging log data](#).

Use Microsoft Message Analyzer to analyze log data

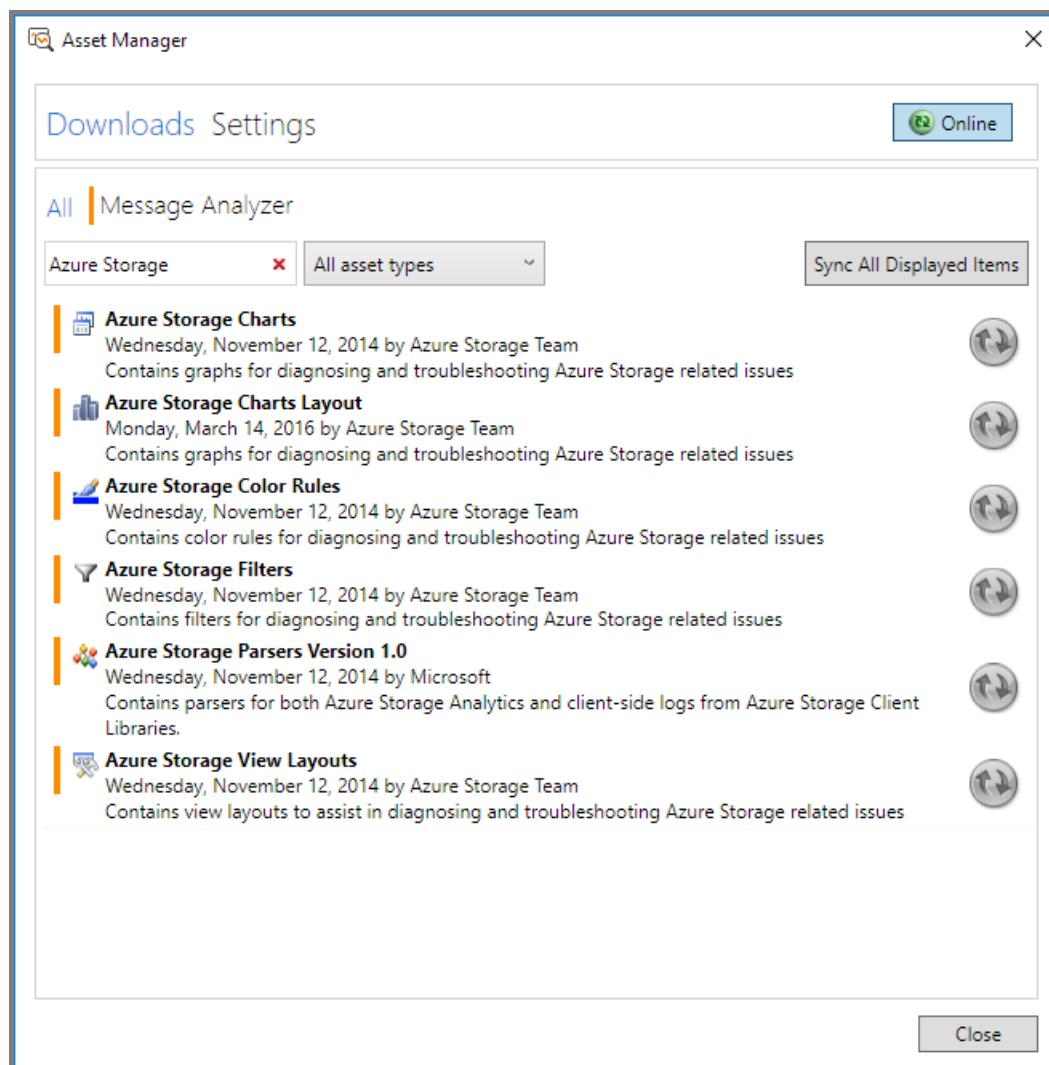
Microsoft Message Analyzer is a tool for capturing, displaying, and analyzing protocol messaging traffic, events,

and other system or application messages in troubleshooting and diagnostic scenarios. Message Analyzer also enables you to load, aggregate, and analyze data from log and saved trace files. For more information about Message Analyzer, see [Microsoft Message Analyzer Operating Guide](#).

Message Analyzer includes assets for Azure Storage that help you to analyze server, client, and network logs. In this section, we'll discuss how to use those tools to address the issue of low percent success in the storage logs.

Download and install Message Analyzer and the Azure Storage Assets

1. Download [Message Analyzer](#) from the Microsoft Download Center, and run the installer.
2. Launch Message Analyzer.
3. From the **Tools** menu, select **Asset Manager**. In the **Asset Manager** dialog, select **Downloads**, then filter on **Azure Storage**. You will see the Azure Storage Assets, as shown in the picture below.
4. Click **Sync All Displayed Items** to install the Azure Storage Assets. The available assets include:
 - **Azure Storage Color Rules:** Azure Storage color rules enable you to define special filters that use color, text, and font styles to highlight messages that contain specific information in a trace.
 - **Azure Storage Charts:** Azure Storage charts are predefined charts that graph server log data. Note that to use Azure Storage charts at this time, you may only load the server log into the Analysis Grid.
 - **Azure Storage Parsers:** The Azure Storage parsers parse the Azure Storage client, server, and HTTP logs in order to display them in the Analysis Grid.
 - **Azure Storage Filters:** Azure Storage filters are predefined criteria that you can use to query your data in the Analysis Grid.
 - **Azure Storage View Layouts:** Azure Storage view layouts are predefined column layouts and groupings in the Analysis Grid.
5. Restart Message Analyzer after you've installed the assets.



NOTE

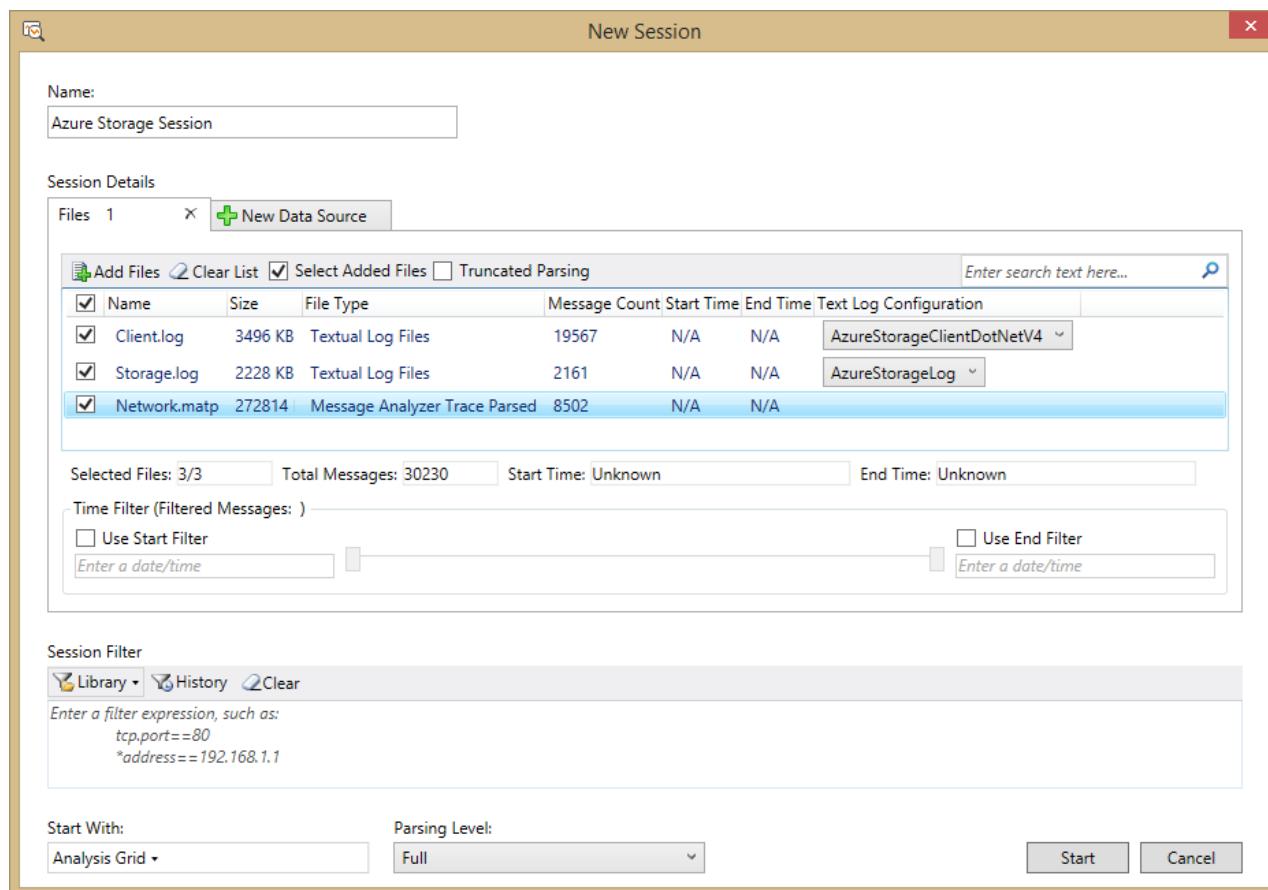
Install all of the Azure Storage assets shown for the purposes of this tutorial.

Import your log files into Message Analyzer

You can import all of your saved log files (server-side, client-side, and network) into a single session in Microsoft Message Analyzer for analysis.

1. On the **File** menu in Microsoft Message Analyzer, click **New Session**, and then click **Blank Session**. In the **New Session** dialog, enter a name for your analysis session. In the **Session Details** panel, click on the **Files** button.
2. To load the network trace data generated by Message Analyzer, click on **Add Files**, browse to the location where you saved your .matp file from your web tracing session, select the .matp file, and click **Open**.
3. To load the server-side log data, click on **Add Files**, browse to the location where you downloaded your server-side logs, select the log files for the time range you want to analyze, and click **Open**. Then, in the **Session Details** panel, set the **Text Log Configuration** drop-down for each server-side log file to **AzureStorageLog** to ensure that Microsoft Message Analyzer can parse the log file correctly.
4. To load the client-side log data, click on **Add Files**, browse to the location where you saved your client-side logs, select the log files you want to analyze, and click **Open**. Then, in the **Session Details** panel, set the **Text Log Configuration** drop-down for each client-side log file to **AzureStorageClientDotNetV4** to ensure that Microsoft Message Analyzer can parse the log file correctly.
5. Click **Start** in the **New Session** dialog to load and parse the log data. The log data displays in the Message Analyzer Analysis Grid.

The picture below shows an example session configured with server, client, and network trace log files.



Note that Message Analyzer loads log files into memory. If you have a large set of log data, you will want to filter it in order to get the best performance from Message Analyzer.

First, determine the time frame that you are interested in reviewing, and keep this time frame as small as possible.

In many cases you will want to review a period of minutes or hours at most. Import the smallest set of logs that can meet your needs.

If you still have a large amount of log data, then you may want to specify a session filter to filter your log data before you load it. In the **Session Filter** box, select the **Library** button to choose a predefined filter; for example, choose **Global Time Filter I** from the Azure Storage filters to filter on a time interval. You can then edit the filter criteria to specify the starting and ending timestamp for the interval you want to see. You can also filter on a particular status code; for example, you can choose to load only log entries where the status code is 404.

For more information about importing log data into Microsoft Message Analyzer, see [Retrieving Message Data](#) on TechNet.

Use the client request ID to correlate log file data

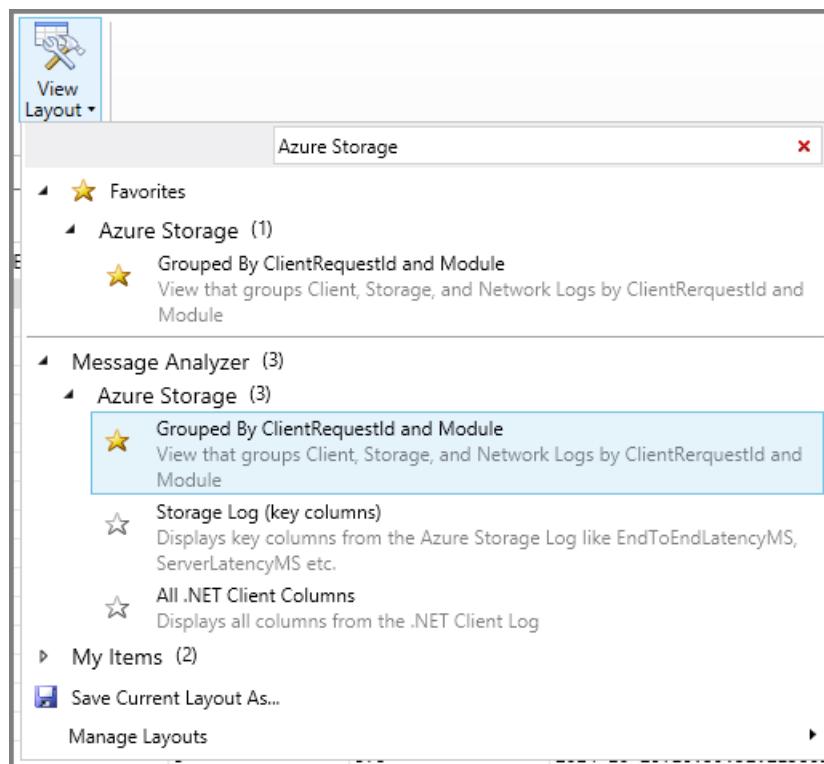
The Azure Storage Client Library automatically generates a unique client request ID for every request. This value is written to the client log, the server log, and the network trace, so you can use it to correlate data across all three logs within Message Analyzer. See [Client request ID](#) for additional information about the client request ID.

The sections below describe how to use pre-configured and custom layout views to correlate and group data based on the client request ID.

Select a view layout to display in the Analysis Grid

The Storage Assets for Message Analyzer include Azure Storage View Layouts, which are pre-configured views that you can use to display your data with useful groupings and columns for different scenarios. You can also create custom view layouts and save them for reuse.

The picture below shows the **View Layout** menu, available by selecting **View Layout** from the toolbar ribbon. The view layouts for Azure Storage are grouped under the **Azure Storage** node in the menu. You can search for **Azure Storage** in the search box to filter on Azure Storage view layouts only. You can also select the star next to a view layout to make it a favorite and display it at the top of the menu.



To begin with, select **Grouped by ClientRequestID and Module**. This view layout groups log data from all three logs first by client request ID, then by source log file (or **Module** in Message Analyzer). With this view, you can drill down into a particular client request ID, and see data from all three log files for that client request ID.

The picture below shows this layout view applied to the sample log data, with a subset of columns displayed. You

can see that for a particular client request ID, the Analysis Grid displays data from the client log, server log, and network trace.

ClientRequestId		Module				
MessageNumber	Timestamp	TimeElapse	Source	Destination	Module	
► ClientRequestId (3): 00d98e5a-cc8f-4a4b-8dc1-5a635d9d7265						
► Module (9): AzureStorageClientDotNetV4						
17271					AzureStorageClientDotNetV4	
17272					AzureStorageClientDotNetV4	
17273					AzureStorageClientDotNetV4	
17274					AzureStorageClientDotNetV4	
17275					AzureStorageClientDotNetV4	
17276					AzureStorageClientDotNetV4	
17277					AzureStorageClientDotNetV4	
17278					AzureStorageClientDotNetV4	
17279					AzureStorageClientDotNetV4	
► Module (1): AzureStorageLog						
1732	2014-10-20T16:39:43.3627051				AzureStorageLog	
► Module (1): HTTP						
7791	2014-10-20T16:39:42.5689042	0.0319180	Local	photouploadercs.blob.cor...	HTTP	
► ClientRequestId (3): 0103dde1-8391-468f-ae87-f95d30718b2b						
► ClientRequestId (3): 011652df-76b2-4ecd-8e92-819ac91087e6						

NOTE

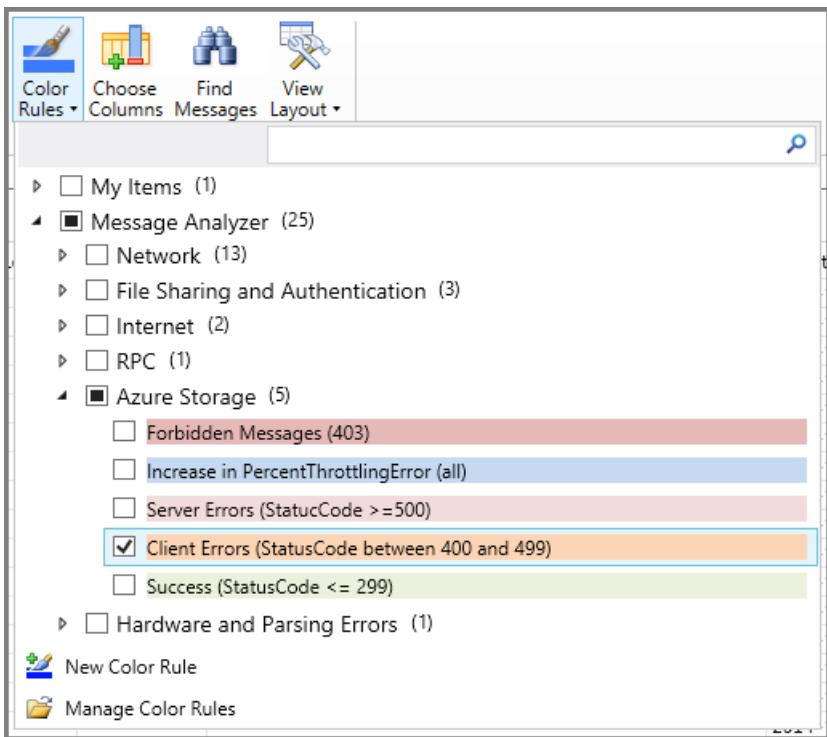
Different log files have different columns, so when data from multiple log files is displayed in the Analysis Grid, some columns may not contain any data for a given row. For example, in the picture above, the client log rows do not show any data for the **Timestamp**, **TimeElapsed**, **Source**, and **Destination** columns, because these columns do not exist in the client log, but do exist in the network trace. Similarly, the **Timestamp** column displays timestamp data from the server log, but no data is displayed for the **TimeElapsed**, **Source**, and **Destination** columns, which are not part of the server log.

In addition to using the Azure Storage view layouts, you can also define and save your own view layouts. You can select other desired fields for grouping data and save the grouping as part of your custom layout as well.

Apply color rules to the Analysis Grid

The Storage Assets also include color rules, which offer a visual means to identify different types of errors in the Analysis Grid. The predefined color rules apply to HTTP errors, so they appear only for the server log and network trace.

To apply color rules, select **Color Rules** from the toolbar ribbon. You'll see the Azure Storage color rules in the menu. For the tutorial, select **Client Errors (StatusCode between 400 and 499)**, as shown in the picture below.



In addition to using the Azure Storage color rules, you can also define and save your own color rules.

Group and filter log data to find 400-range errors

Next, we'll group and filter the log data to find all errors in the 400 range.

1. Locate the **StatusCode** column in the Analysis Grid, right-click the column heading, and select **Group**.
2. Next, group on the **ClientRequestId** column. You'll see that the data in the Analysis Grid is now organized by status code and by client request ID.
3. Display the View Filter tool window if it is not already displayed. On the toolbar ribbon, select **Tool Windows**, then **View Filter**.
4. To filter the log data to display only 400-range errors, add the following filter criteria to the **View Filter** window, and click **Apply**:

```
(AzureStorageLog.StatusCode >= 400 && AzureStorageLog.StatusCode <=499) || (HTTP.StatusCode >= 400 && HTTP.StatusCode <= 499)
```

The picture below shows the results of this grouping and filter. Expanding the **ClientRequestId** field beneath the grouping for status code 409, for example, shows an operation that resulted in that status code.

After applying this filter, you'll see that rows from the client log are excluded, as the client log does not include a **StatusCode** column. To begin with, we'll review the server and network trace logs to locate 404 errors, and then

we'll return to the client log to examine the client operations that led to them.

NOTE

You can filter on the **StatusCode** column and still display data from all three logs, including the client log, if you add an expression to the filter that includes log entries where the status code is null. To construct this filter expression, use:

```
*StatusCode >= 400 or !*StatusCode
```

This filter returns all rows from the client log and only rows from the server log and HTTP log where the status code is greater than 400. If you apply it to the view layout grouped by client request ID and module, you can search or scroll through the log entries to find ones where all three logs are represented.

Filter log data to find 404 errors

The Storage Assets include predefined filters that you can use to narrow log data to find the errors or trends you are looking for. Next, we'll apply two predefined filters: one that filters the server and network trace logs for 404 errors, and one that filters the data on a specified time range.

1. Display the View Filter tool window if it is not already displayed. On the toolbar ribbon, select **Tool Windows**, then **View Filter**.
2. In the View Filter window, select **Library**, and search on **Azure Storage** to find the Azure Storage filters. Select the filter for **404 (Not Found) messages in all logs**.
3. Display the **Library** menu again, and locate and select the **Global Time Filter**.
4. Edit the timestamps shown in the filter to the range you wish to view. This will help to narrow the range of data to analyze.
5. Your filter should appear similar to the example below. Click **Apply** to apply the filter to the Analysis Grid.

```
((AzureStorageLog.StatusCode == 404 || HTTP.StatusCode == 404)) And  
(#Timestamp >= 2014-10-20T16:36:38 and #Timestamp <= 2014-10-20T16:36:39)
```

The screenshot shows the 'View Filter' tool window with the title bar 'Start Page Azure Storage Se...'. A red box highlights the 'ClientRequestId' filter field. The main area displays a table with columns: 'MessageNumber', 'Timestamp', and 'Summary'. The table is grouped by 'ClientRequestId'. Several rows are visible, each containing two entries. For example, under 'ClientRequestId (2): 0b3d195b-aa64-42f3-ad58-c69632080f36', there are two rows: one for '3193' (Timestamp: 2014-10-20T16:36:38.8480664, Operation: GetBlob, Status: The specified blob does not exist. (404), GET http://) and one for '481' (Timestamp: 2014-10-20T16:36:38.9245975, Operation: GetBlob, Status: Success). Other groups show similar patterns of requests and their outcomes.

Analyze your log data

Now that you have grouped and filtered your data, you can examine the details of individual requests that generated 404 errors. In the current view layout, the data is grouped by client request ID, then by log source. Since we are filtering on requests where the StatusCode field contains 404, we'll see only the server and network trace data, not the client log data.

The picture below shows a specific request where a Get Blob operation yielded a 404 because the blob did not exist. Note that some columns have been removed from the standard view in order to display the relevant data.

ClientRequestId		Module			
MessageNumber	Timestamp	Module	Summary	StatusCode	
ClientRequestId (2): 2db35a12-b9f8-42d8-853e-8b60ec18b22d					
ClientRequestId (2): 2de8f2e5-5b7c-4092-9b5b-6bb12522bb13					
ClientRequestId (2): 398bac41-7725-484b-8a69-2a9e48fc669a		Module (1): AzureStorageLog			
411	2014-10-20T16:36:36.1403190	AzureStorageLog	GetBlob	404	
Module (1): HTTP					
3045	2014-10-20T16:36:36.0630581	HTTP	Operation, Status: The specified blob does not exist. (404), GET http://...	404	
ClientRequestId (2): 3a89917e-24ff-4ed9-bd60-991670442cca					
ClientRequestId (2): 3e30de3b-abef-4237-928d-1384ceb02e26					

Next, we'll correlate this client request ID with the client log data to see what actions the client was taking when the error happened. You can display a new Analysis Grid view for this session to view the client log data, which opens in a second tab:

1. First, copy the value of the **ClientRequestId** field to the clipboard. You can do this by selecting either row, locating the **ClientRequestId** field, right-clicking on the data value, and choosing **Copy 'ClientRequestId'**.
2. On the toolbar ribbon, select **New Viewer**, then select **Analysis Grid** to open a new tab. The new tab shows all data in your log files, without grouping, filtering, or color rules.
3. On the toolbar ribbon, select **View Layout**, then select **All .NET Client Columns** under the **Azure Storage** section. This view layout shows data from the client log as well as the server and network trace logs. By default it is sorted on the **MessageNumber** column.
4. Next, search the client log for the client request ID. On the toolbar ribbon, select **Find Messages**, then specify a custom filter on the client request ID in the **Find** field. Use this syntax for the filter, specifying your own client request ID:

```
*ClientRequestId == "398bac41-7725-484b-8a69-2a9e48fc669a"
```

Message Analyzer locates and selects the first log entry where the search criteria matches the client request ID. In the client log, there are several entries for each client request ID, so you may want to group them on the **ClientRequestId** field to make it easier to see them all together. The picture below shows all of the messages in the client log for the specified client request ID.

ClientRequestId					
MessageNumber	ClientRequestId	Level	LevelString	Description	
ClientRequestId (11): 39616321-08f2-4ce0-b1a8-45ac1ff182b4					
ClientRequestId (13): 3986bd72-5a8c-4f0d-8730-d5bf1536112					
ClientRequestId (12): 398bac41-7725-484b-8a69-2a9e48fc669a					
3803	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	Starting operation with location Primary per location mode PrimaryOnly.	
3804	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	Starting synchronous request to http://photouploaders.blob.core.windows.net/testclde9e5dad9c54fc6b0...	
3805	398bac41-7725-484b-8a69-2a9e48fc669a	4	Verbose	StringToSign = GET.....x-ms-client-request-id:398bac41-7725-484b-8a69-2a9e48fc669a.x-ms-date:...	
3806	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	Waiting for response.	
3807	398bac41-7725-484b-8a69-2a9e48fc669a	2	Exception	Exception thrown while waiting for response: The remote server returned an error: (404) Not Found..	
3808	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	Response received. Status code = 404, Request ID = 793143c6-0001-0029-1c50-74e56000000, Content-MD5=...	
3809	398bac41-7725-484b-8a69-2a9e48fc669a	2	Warning	Exception thrown during the operation: The remote server returned an error: (404) Not Found..	
3810	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	Checking if the operation should be retried. Retry count = 0, HTTP status code = 404, Retryable exce...	
3811	398bac41-7725-484b-8a69-2a9e48fc669a	3	Information	The next location has been set to Primary, based on the location mode.	
3812	398bac41-7725-484b-8a69-2a9e48fc669a	1	Error	Retry policy did not allow for a retry. Failing with The remote server returned an error: (404) Not...	
3045	398bac41-7725-484b-8a69-2a9e48fc669a				
411	398bac41-7725-484b-8a69-2a9e48fc669a				
ClientRequestId (13): 399fec04-555e-4b81-900d-f178a4a8f352					
ClientRequestId (9): 39b83db1-8409-4b37-be31-bf38a3fcc86					
ClientRequestId (11): 39cf9f20-a8d4-43b2-b67d-9785c17e031					

Using the data shown in the view layouts in these two tabs, you can analyze the request data to determine what may have caused the error. You can also look at requests that preceded this one to see if a previous event may have led to the 404 error. For example, you can review the client log entries preceding this client request ID to determine whether the blob may have been deleted, or if the error was due to the client application calling a **CreateIfNotExists** API on a container or blob. In the client log, you can find the blob's address in the **Description** field; in the server and network trace logs, this information appears in the **Summary** field.

Once you know the address of the blob that yielded the 404 error, you can investigate further. If you search the log entries for other messages associated with operations on the same blob, you can check whether the client

previously deleted the entity.

Analyze other types of storage errors

Now that you are familiar with using Message Analyzer to analyze your log data, you can analyze other types of errors using view layouts, color rules, and searching/filtering. The tables below lists some issues you may encounter and the filter criteria you can use to locate them. For more information on constructing filters and the Message Analyzer filtering language, see [Filtering Message Data](#).

TO INVESTIGATE...	USE FILTER EXPRESSION...	EXPRESSION APPLIES TO LOG (CLIENT, SERVER, NETWORK, ALL)
Unexpected delays in message delivery on a queue	AzureStorageClientDotNetV4.Description contains "Retrying failed operation."	Client
HTTP Increase in PercentThrottlingError	HTTP.Response.StatusCode == 500 HTTP.Response.StatusCode == 503	Network
Increase in PercentTimeoutError	HTTP.Response.StatusCode == 500	Network
Increase in PercentTimeoutError (all)	*StatusCode == 500	All
Increase in PercentNetworkError	AzureStorageClientDotNetV4.EventLog.Entry.Level < 2	Client
HTTP 403 (Forbidden) messages	HTTP.Response.StatusCode == 403	Network
HTTP 404 (Not found) messages	HTTP.Response.StatusCode == 404	Network
404 (all)	*StatusCode == 404	All
Shared Access Signature (SAS) authorization issue	AzureStorageLog.RequestStatus == "SASAuthorizationError"	Network
HTTP 409 (Conflict) messages	HTTP.Response.StatusCode == 409	Network
409 (all)	*StatusCode == 409	All
Low PercentSuccess or analytics log entries have operations with transaction status of ClientOtherErrors	AzureStorageLog.RequestStatus == "ClientOtherError"	Server
Nagle Warning	((AzureStorageLog.EndToEndLatencyMS - AzureStorageLog.ServerLatencyMS) > (AzureStorageLog.ServerLatencyMS * 1.5)) and (AzureStorageLog.RequestPacketSize < 1460) and (AzureStorageLog.EndToEndLatencyMS - AzureStorageLog.ServerLatencyMS >= 200)	Server
Range of time in Server and Network logs	#Timestamp >= 2014-10-20T16:36:38 and #Timestamp <= 2014-10-20T16:36:39	Server, Network

TO INVESTIGATE...	USE FILTER EXPRESSION...	EXPRESSION APPLIES TO LOG (CLIENT, SERVER, NETWORK, ALL)
Range of time in Server logs	AzureStorageLog.Timestamp >= 2014-10-20T16:36:38 and AzureStorageLog.Timestamp <= 2014-10-20T16:36:39	Server

Next steps

For more information about troubleshooting end-to-end scenarios in Azure Storage, see these resources:

- [Monitor, diagnose, and troubleshoot Microsoft Azure Storage](#)
- [Storage Analytics](#)
- [Monitor a storage account in the Azure Portal](#)
- [Transfer data with the AzCopy command-line utility](#)
- [Microsoft Message Analyzer Operating Guide](#)

Azure Storage Client Tools

1/17/2017 • 1 min to read • [Edit on GitHub](#)

Users of Azure Storage frequently want to be able to view/interact with their data using an Azure Storage client tool. In the tables below, we list a number of tools that allow you to do this. We put an "X" in each block if it provides the ability to either enumerate and/or access the data abstraction. The table also shows if the tools is free or not. "Trial" indicates that there is a free trial, but the full product is not free. "Y/N" indicates that a version is available for free, while a different version is available for purchase.

We've only provided a snapshot of the available Azure Storage client tools. These tools may continue to evolve and grow in functionality. If there are corrections or updates, please leave a comment to let us know. The same is true if you know of tools that ought to be here - we'd be happy to add them.

Microsoft Azure Storage Client Tools

AZURE STORA GE CLIE NT TOOL	BLOCK BLOB	PAGE BLOB	APPEN D BLOB	TABLES	QUEUE S	FILE S	FREE	PLATFORM			
								Web	Windo ws	OSX	Linux
Micro soft Azure Portal	X	X	X	X	X	X	Y	X			
Micro soft Azure Stora ge Explor er	X	X	X	X	X	X	Y		X	X	X
Micro soft Visual Studio Server Explor er	X	X	X	X	X		Y		X		

Third-Party Azure Storage Client Tools

We have not verified the functionality or quality claimed by the following third-party tools and their listing does not imply an endorsement by Microsoft.

AZURE STORA GE CLIE NT TOOL	BLOCK BLOB	PAGE BLOB	APPEN D BLOB	TABLES	QUEUE S	FILE S	FREE	PLATFORM			
								Web	Windo ws	OSX	Linux
Cloud Porta m	X	X	X	X	X	X	Trial	X			

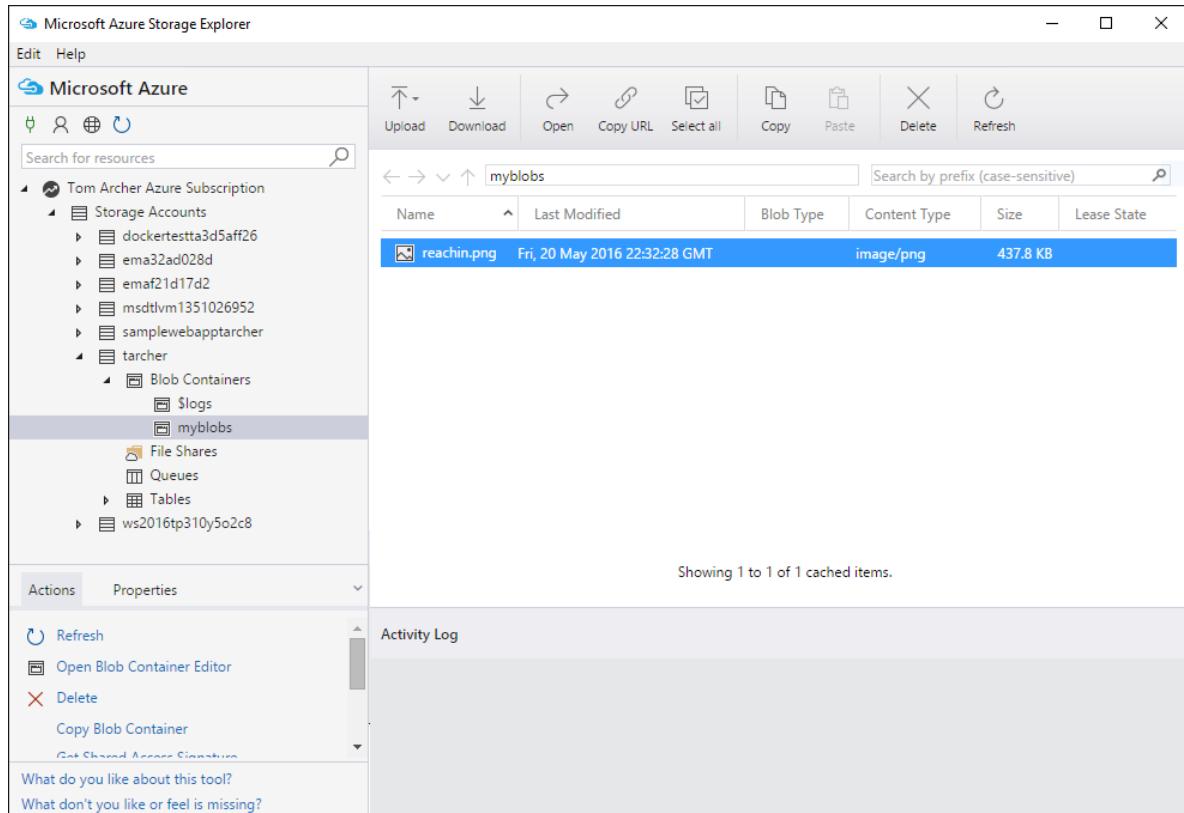
Cerab rata: Azure Mana geme nt Studio	X	X	X	X	X	X	Trial		X		
Cerab rata: Azure Explor er	X	X	X			X	Y		X		
Azure Stora ge Explor er	X	X		X	X		Y		X		
Cloud Berry Explor er	X	X				X	Y/N		X		
Cloud Comb ine	X	X		X	X		Trial		X		
Clums yLeaf: Azure Xplore r, Cloud Xplore r, Table Xplore r	X	X	X	X	X	X	Y		X		
Gladin et Cloud	X						Trial		X		
Azure Web Stora ge Explor er	X	X		X	X		Y	X			
Zudio	X	X		X	X	X	Trial	X			

Getting started with Storage Explorer (Preview)

1/17/2017 • 7 min to read • [Edit on GitHub](#)

Overview

Microsoft Azure Storage Explorer (Preview) is a standalone app that enables you to easily work with Azure Storage data on Windows, OS X, and Linux. In this article, you'll learn the various ways of connecting to and managing your Azure storage accounts.



Prerequisites

- [Download and install Storage Explorer \(preview\)](#)

Connect to a storage account or service

Storage Explorer (Preview) provides a myriad ways to connect to storage accounts. This includes connecting to storage accounts associated with your Azure subscriptions, connecting to storage accounts and services shared from other Azure subscriptions, and even connecting to and managing local storage using the Azure Storage Emulator. In addition, you can work with storage accounts in global and national Azure:

- [Connect to an Azure subscription](#) - Manage storage resources belonging to your Azure subscription.
- [Work with local development storage](#) - Manage local storage using the Azure Storage Emulator.
- [Attach to external storage](#) - Manage storage resources belonging to another Azure subscription or under national Azure clouds using the storage account's name, key and endpoints.
- [Attach storage account using SAS](#) - Manage storage resources belonging to another Azure subscription using a SAS.
- [Attach service using SAS](#) - Manage a specific storage service (blob container, queue, or table) belonging

to another Azure subscription using a SAS.

Connect to an Azure subscription

NOTE

If you don't have an Azure account, you can [sign up for a free trial](#) or [activate your Visual Studio subscriber benefits](#).

1. In Storage Explorer (Preview), select **Azure Account settings**.

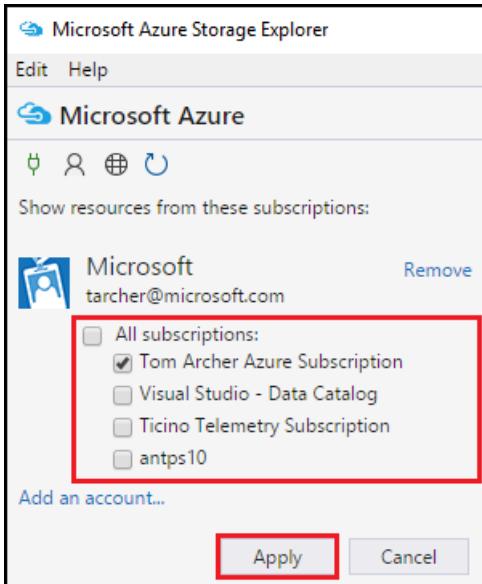


2. The left pane will now display all the Microsoft accounts you've logged into. To connect to another account, select **Add an account**, and follow the dialogs to sign in with a Microsoft account that is associated with at least one active Azure subscription.

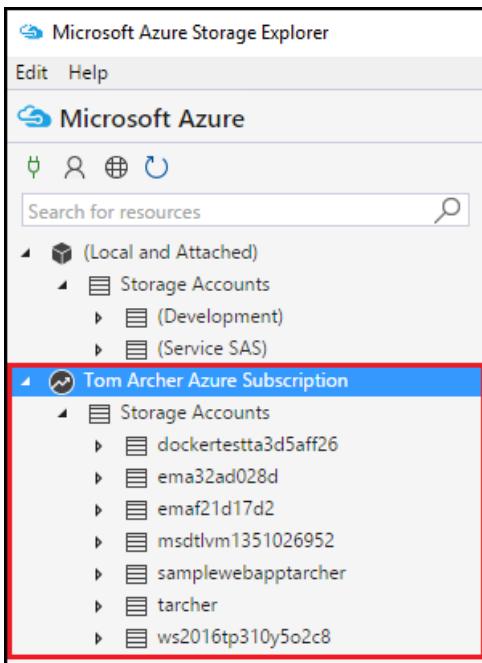
NOTE

Connecting to national Azure such as Black Forest Azure, Fairfax Azure and Mooncake Azure via sign-in is currently not supported. See **Attach or detach an external storage account** section for how to connect to national Azure storage accounts.

3. Once you successfully sign in with a Microsoft account, the left pane will populate with the Azure subscriptions associated with that account. Select the Azure subscriptions with which you want to work, and then select **Apply**. (Selecting **All subscriptions** toggles selecting all or none of the listed Azure subscriptions.)



4. The left pane will now display the storage accounts associated with the selected Azure subscriptions.



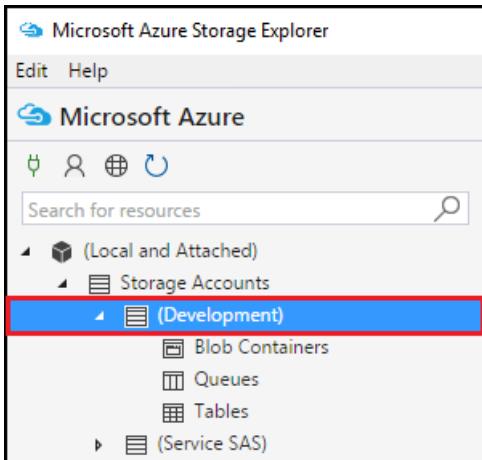
Work with local development storage

Storage Explorer (Preview) enables you to work against local storage using the Azure Storage Emulator. This allows you to write code against and test storage without necessarily having a storage account deployed on Azure (since the storage account is being emulated by the Azure Storage Emulator).

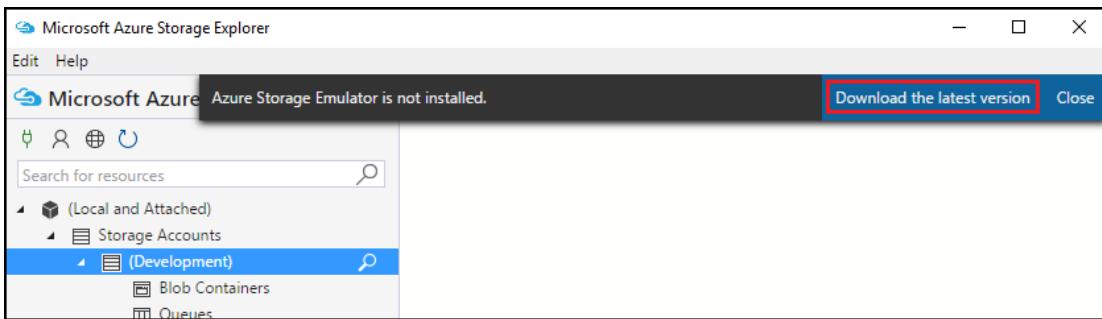
NOTE

The Azure Storage Emulator is currently supported only for Windows.

1. In the left pane of Storage Explorer (Preview), expand the **(Local and Attached > Storage Accounts > (Development))** node.



2. If you have not yet installed the Azure Storage Emulator, you'll be prompted to do so via an infobar. If the infobar is displayed, select **Download the latest version**, and install the emulator.



3. Once the emulator is installed, you'll have the ability to create and work with local blobs, queues, and tables. To learn how to work with each storage account type, select on the appropriate link below:

- [Manage Azure blob storage resources](#)
- Manage Azure file share storage resources - *Coming soon*
- Manage Azure queue storage resources - *Coming soon*
- Manage Azure table storage resources - *Coming soon*

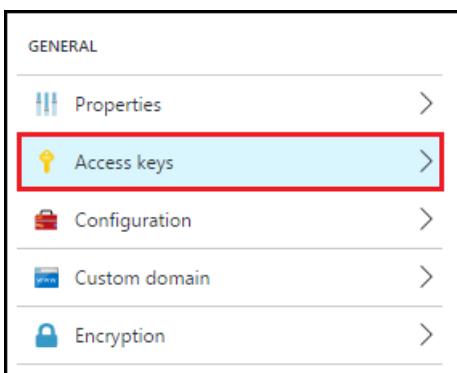
Attach or detach an external storage account

Storage Explorer (Preview) provides the ability to attach to external storage accounts so that storage accounts can be easily shared. This section explains how to attach to (and detach from) external storage accounts.

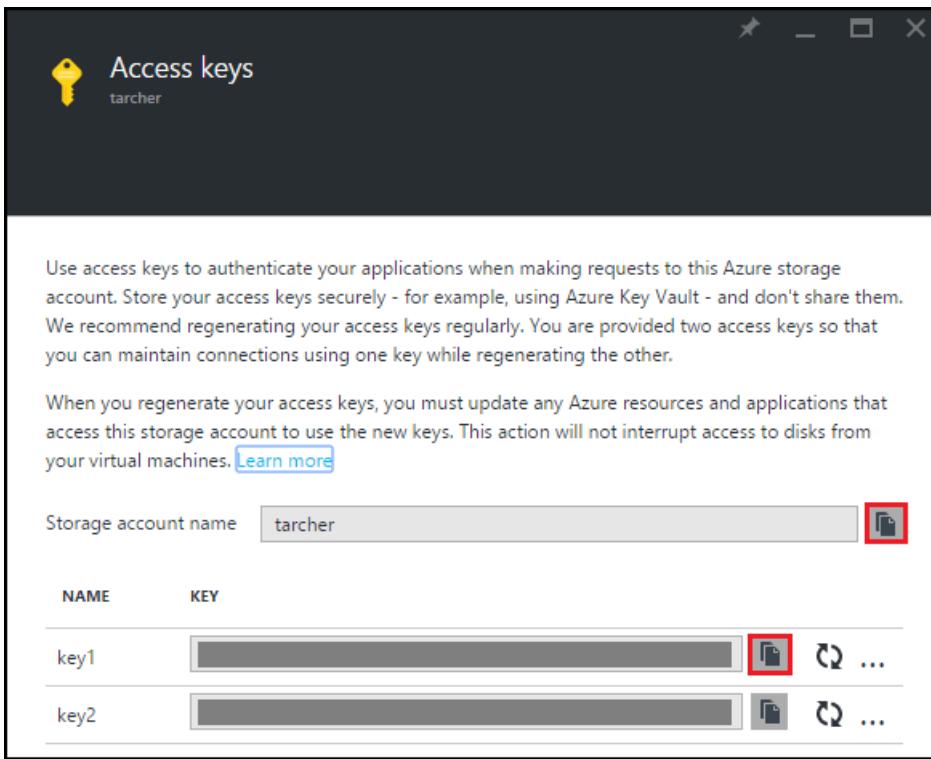
Get the storage account credentials

In order to share an external storage account, the owner of that account must first get the credentials - account name and key - for the account and then share that information with the person wanting to attach to that (external) account. Obtaining the storage account credentials can be done via the Azure portal by following these steps:

1. Sign in to the [Azure portal](#).
2. Select **Browse**.
3. Select **Storage Accounts**.
4. In the **Storage Accounts** blade, select the desired storage account.
5. In the **Settings** blade for the selected storage account, select **Access keys**.



6. In the **Access keys** blade, copy the **STORAGE ACCOUNT NAME** and **KEY 1** values for use when attaching to the storage account.



Attach to an external storage account

To attach to an external storage account, you'll need the account's name and key. The section *Get the storage account credentials* explains how to obtain these values from the Azure portal. However, note that in the portal, the account key is called "key 1" so where the Storage Explorer (Preview) asks for an account key, you'll enter (or paste) the "key 1" value.

1. In Storage Explorer (Preview), select **Connect to Azure storage**.



2. On the **Connect to Azure Storage** dialog, specify the account key ("key 1" value from the Azure portal), and then select **Next**.

NOTE

You can enter Storage Connection string from a storage account on national Azure. For example, enter connection strings similar to the following to connect to Azure Black Forest storage accounts:
DefaultEndpointsProtocol=https;AccountName=cawatest03;AccountKey=;EndpointSuffix=core.cloudapi.de;
You can get the connection string from Azure portal in the same way as described in the **Get the storage account credentials** section

Connect to Azure Storage

Enter a connection string, Shared Access Signature (SAS) URI, or an account key.

3. In the **Attach External Storage** dialog, enter the storage account name in the **Account name** box, specify any other desired settings, and select **Next** when done.

Attach External Storage

Enter information to connect to the Microsoft Azure storage account

Account name:

Account key:

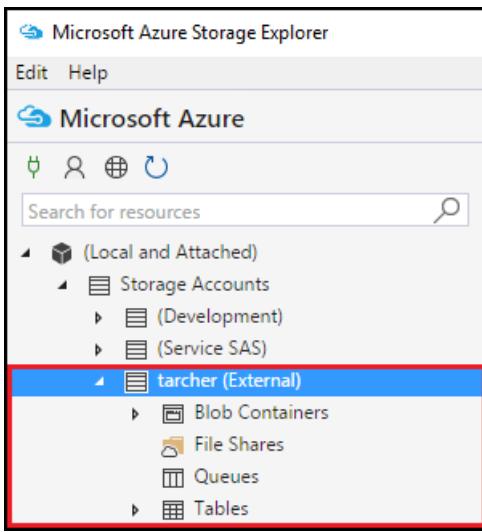
Storage endpoints domain:

- Microsoft Azure Default
- Microsoft Azure China
- Other (specify below)

Use HTTP (Not recommended)

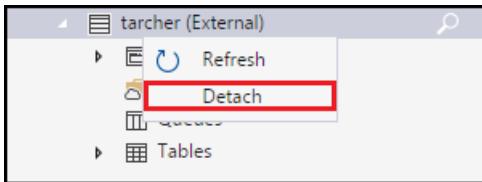
[Online privacy statement](#)

4. In the **Connection Summary** dialog, verify the information. If you want to change anything, select **Back** and re-enter the desired settings. Once finished, select **Connect**.
5. Once connected, the external storage account will be displayed with the text (**External**) appended to the storage account name.



Detach from an external storage account

1. Right-click the external storage account you want to detach, and - from the context menu - select **Detach**.



2. When the confirmation message box appears, select **Yes** to confirm the detachment from the external storage account.

Attach storage account using SAS

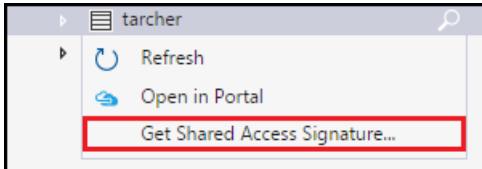
A [SAS \(Shared Access Signature\)](#) gives the admin of an Azure subscription the ability to grant access to a storage account on a temporary basis without having to provide their Azure subscription credentials.

To illustrate this, let's say UserA is an admin of an Azure subscription, and UserA wants to allow UserB to access a storage account for a limited time with certain permissions:

1. UserA generates a SAS (consisting of the connection string for the storage account) for a specific time period and with the desired permissions.
2. UserA shares the SAS with the person wanting access to the storage account - UserB, in our example.
3. UserB uses Storage Explorer (Preview) to attach to the account belonging to UserA using the supplied SAS.

Get a SAS for the account you want to share

1. In Storage Explorer (Preview), right-click the storage account you want share, and - from the context menu - select **Get Shared Access Signature**.



2. On the **Shared Access Signature** dialog, specify the time frame and permissions you want for the account, and select **Create**.

Shared Access Signature

Start time: 05/18/2016 04:23 PM

Expiry time: 05/19/2016 04:23 PM

Time zone:

Local
 UTC

Permissions:

Read
 Write
 Delete
 List

Services:

Blobs
 Queues
 Tables

Resource Types:

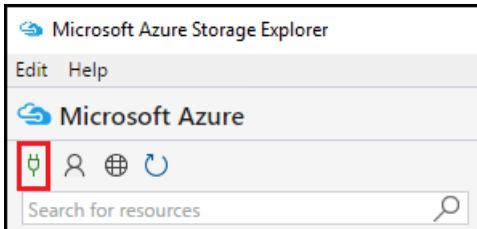
Service
 Container
 Object

Create **Cancel**

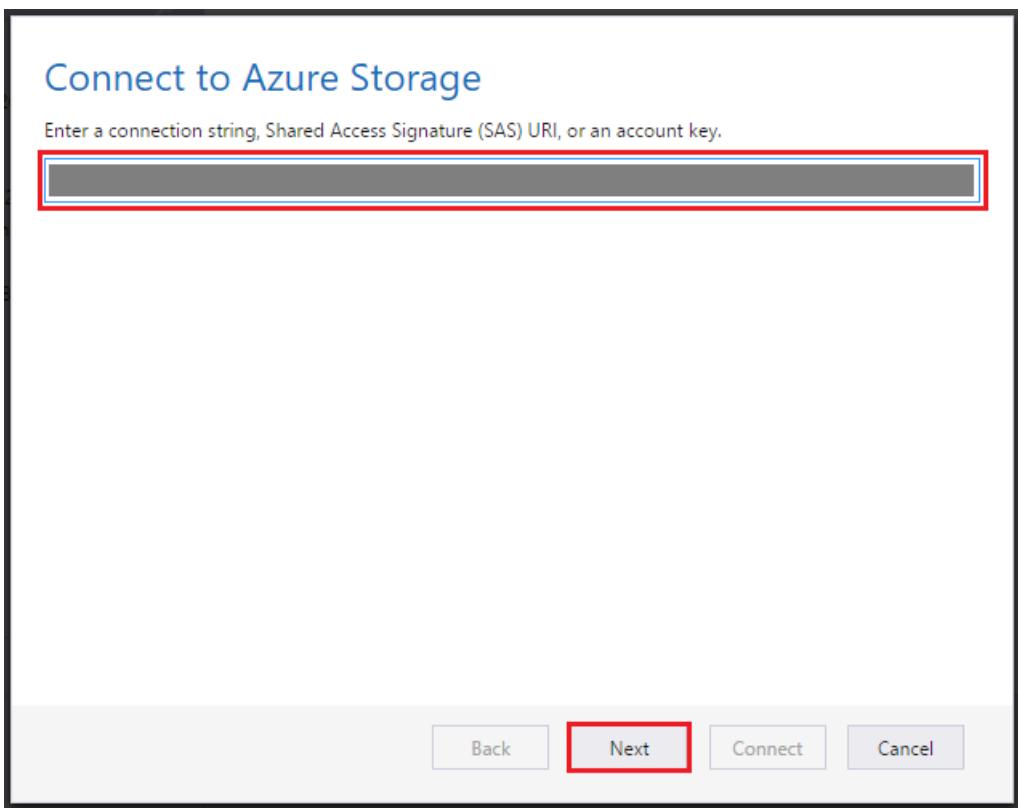
3. A second **Shared Access Signature** dialog will appear displaying the SAS. Select **Copy** next to the **Connection String** to copy it to the clipboard. Select **Close** to dismiss the dialog.

Attach to the shared account using the SAS

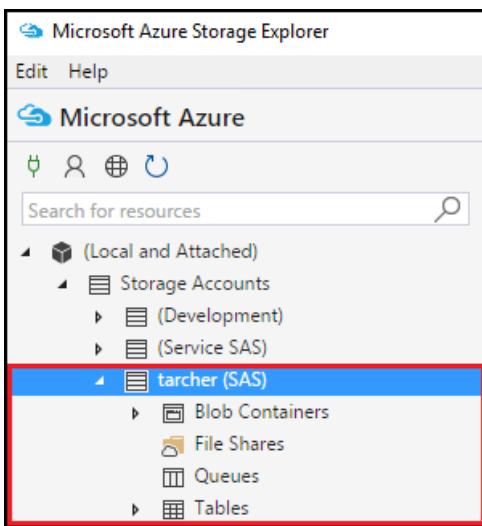
1. In Storage Explorer (Preview), select **Connect to Azure storage**.



2. On the **Connect to Azure Storage** dialog, specify the connection string, and then select **Next**.



3. In the **Connection Summary** dialog, verify the information. If you want to change anything, select **Back** and re-enter the desired settings. Once finished, select **Connect**.
4. Once attached, the storage account will be displayed with the text (SAS) appended to the account name you supplied.



Attach service using SAS

The section [Attach storage account using SAS](#) illustrates how an Azure subscription admin can grant temporary access to a storage account by generating (and sharing) a SAS for the storage account. Similarly, a SAS can be generated for a specific service (blob container, queue, or table) within a storage account.

Generate a SAS for the service you want to share

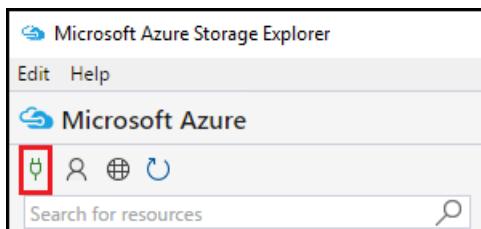
In this context, a service can be a blob container, queue, or table. The following sections explain how to generate the SAS for the listed service:

- [Get the SAS for a blob container](#)
- Get the SAS for a file share - *Coming soon*

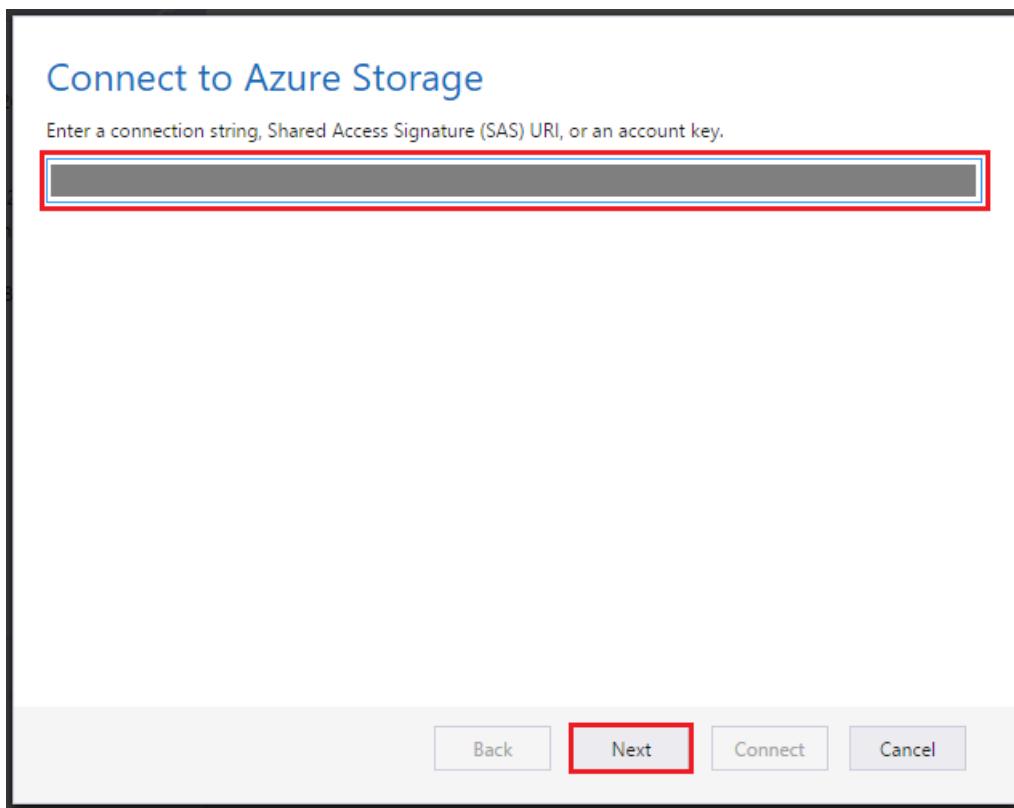
- Get the SAS for a queue - *Coming soon*
- Get the SAS for a table - *Coming soon*

Attach to the shared account service using the SAS

1. In Storage Explorer (Preview), select **Connect to Azure storage**.

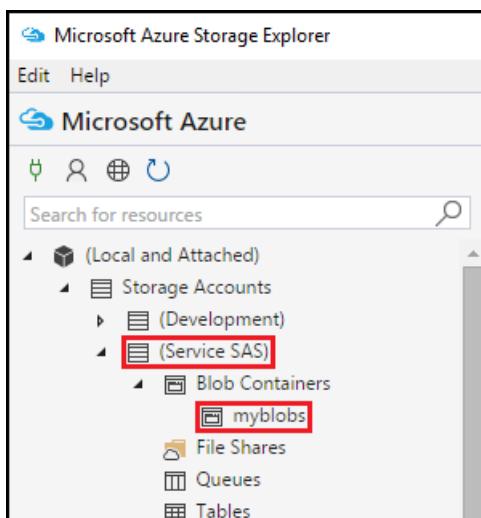


2. On the **Connect to Azure Storage** dialog, specify the SAS URI, and then select **Next**.



3. In the **Connection Summary** dialog, verify the information. If you want to change anything, select **Back** and re-enter the desired settings. Once finished, select **Connect**.

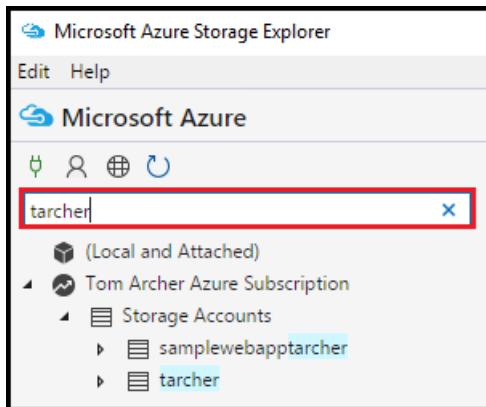
4. Once attached, the newly attached service will be displayed under the **(Service SAS)** node.



Search for storage accounts

If you have a long list of storage accounts, a quick way to locate a particular storage account is to use the search box at the top of the left pane.

As you are typing into the search box, the left pane will display only the storage accounts that match the search value you've entered up to that point. The following screen shot illustrates an example where I've searched for all storage accounts where the storage account name contains the text "tarcher".



To clear the search, select the **x** button in the search box.

Next steps

- [Manage Azure blob storage resources with Storage Explorer \(Preview\)](#)

Manage Azure Blob Storage resources with Storage Explorer (Preview)

1/17/2017 • 7 min to read • [Edit on GitHub](#)

Overview

[Azure Blob Storage](#) is a service for storing large amounts of unstructured data, such as text or binary data, that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately. In this article, you'll learn how to use Storage Explorer (Preview) to work with blob containers and blobs.

Prerequisites

To complete the steps in this article, you'll need the following:

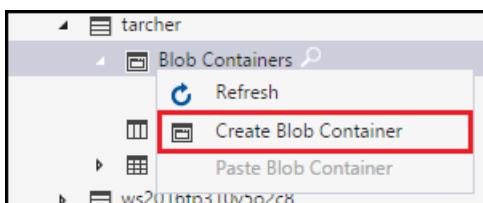
- [Download and install Storage Explorer \(preview\)](#)
- [Connect to a Azure storage account or service](#)

Create a blob container

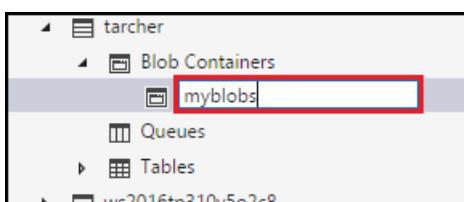
All blobs must reside in a blob container, which is simply a logical grouping of blobs. An account can contain an unlimited number of containers, and each container can store an unlimited number of blobs.

The following steps illustrate how to create a blob container within Storage Explorer (Preview).

1. Open Storage Explorer (Preview).
2. In the left pane, expand the storage account within which you wish to create the blob container.
3. Right-click **Blob Containers**, and - from the context menu - select **Create Blob Container**.



4. A text box will appear below the **Blob Containers** folder. Enter the name for your blob container. See the [Container naming rules](#) section for a list of rules and restrictions on naming blob containers.



5. Press **Enter** when done to create the blob container, or **Esc** to cancel. Once the blob container has been successfully created, it will be displayed under the **Blob Containers** folder for the selected storage account.

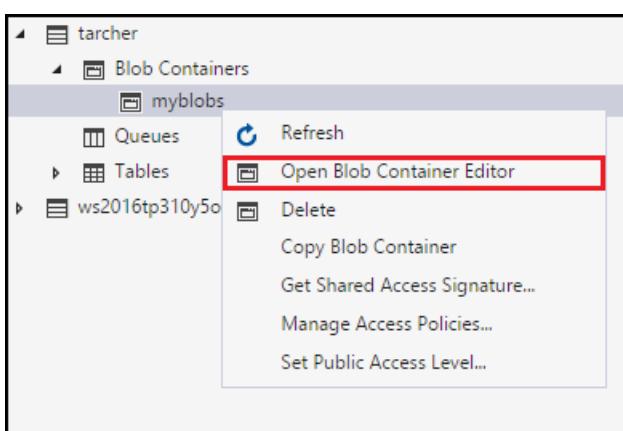


View a blob container's contents

Blob containers contain blobs and folders (that can also contain blobs).

The following steps illustrate how to view the contents of a blob container within Storage Explorer (Preview):

1. Open Storage Explorer (Preview).
2. In the left pane, expand the storage account containing the blob container you wish to view.
3. Expand the storage account's **Blob Containers**.
4. Right-click the blob container you wish to view, and - from the context menu - select **Open Blob Container Editor**. You can also double-click the blob container you wish to view.



5. The main pane will display the blob container's contents.

A screenshot of the main pane of Storage Explorer. At the top, there is a toolbar with icons for Upload, Download, Open, Copy URL, Select all, Copy, Paste, Delete, and Refresh. Below the toolbar, there is a search bar with the prefix 'myblobs' and a 'Search by prefix' button. A table below the search bar displays the contents of the 'myblobs' container. The table has columns for Name, Last Modified, Blob Type, Content Type, and Size. One item is listed: 'reachin.png' with a timestamp of 'Tue, 17 May 2016 20:21:44 GMT', type 'Block Blob', content type 'image/png', and size '437.8 KB'. At the bottom of the pane, a message says 'Showing 1 to 1 of 1 loaded items.'

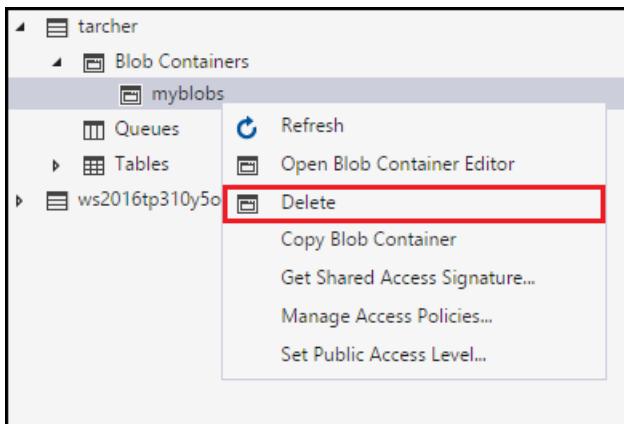
Delete a blob container

Blob containers can be easily created and deleted as needed. (To see how to delete individual blobs, refer to the section, [Managing blobs in a blob container](#).)

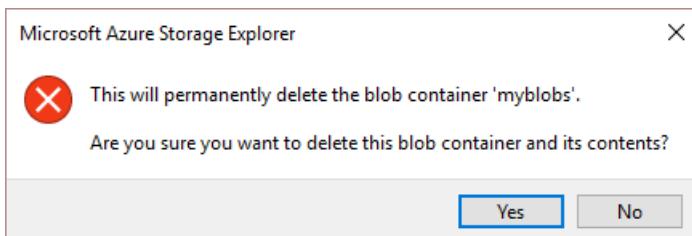
The following steps illustrate how to delete a blob container within Storage Explorer (Preview):

1. Open Storage Explorer (Preview).
2. In the left pane, expand the storage account containing the blob container you wish to view.
3. Expand the storage account's **Blob Containers**.
4. Right-click the blob container you wish to delete, and - from the context menu - select **Delete**. You can

also press **Delete** to delete the currently selected blob container.



5. Select **Yes** to the confirmation dialog.

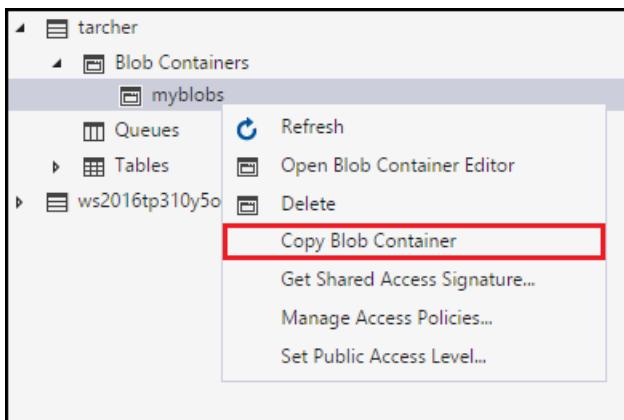


Copy a blob container

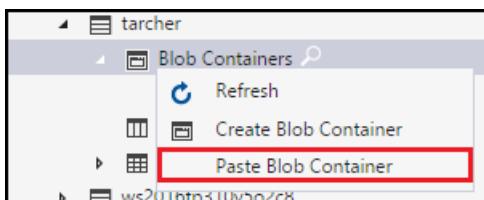
Storage Explorer (Preview) enables you to copy a blob container to the clipboard, and then paste that blob container into another storage account. (To see how to copy individual blobs, refer to the section, [Managing blobs in a blob container](#).)

The following steps illustrate how to copy a blob container from one storage account to another.

1. Open Storage Explorer (Preview).
2. In the left pane, expand the storage account containing the blob container you wish to copy.
3. Expand the storage account's **Blob Containers**.
4. Right-click the blob container you wish to copy, and - from the context menu - select **Copy Blob Container**.



5. Right-click the desired "target" storage account into which you want to paste the blob container, and - from the context menu - select **Paste Blob Container**.

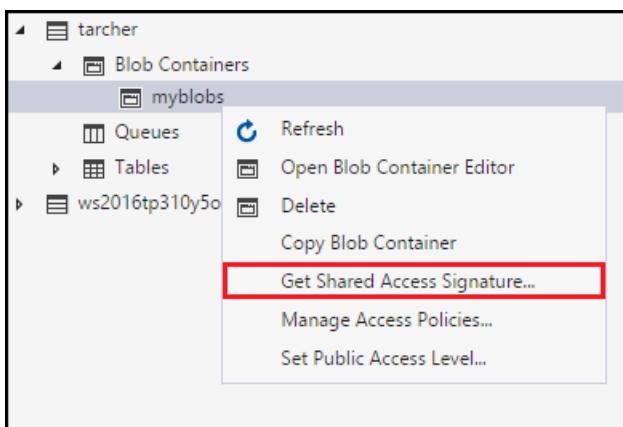


Get the SAS for a blob container

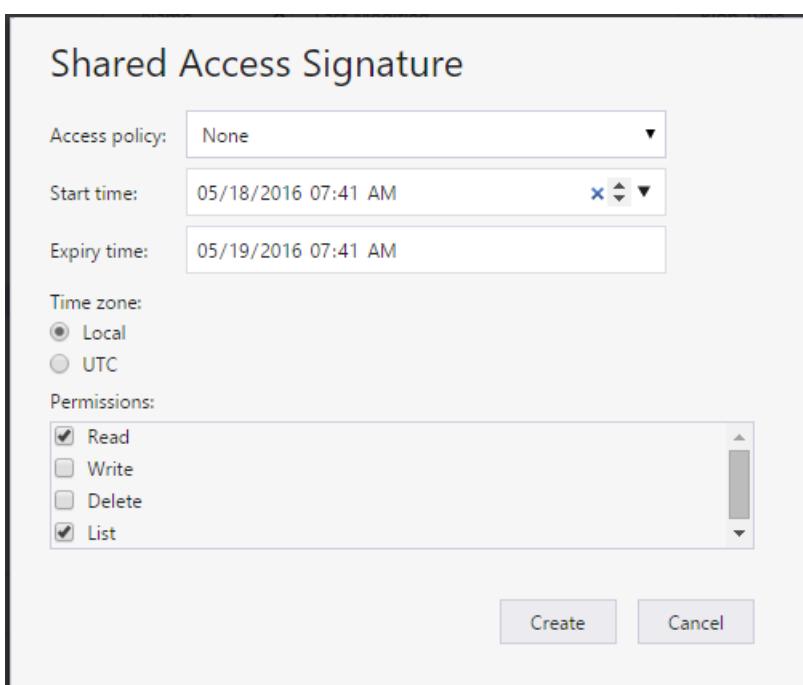
A [shared access signature \(SAS\)](#) provides delegated access to resources in your storage account. This means that you can grant a client limited permissions to objects in your storage account for a specified period of time and with a specified set of permissions, without having to share your account access keys.

The following steps illustrate how to create a SAS for a blob container:

1. Open Storage Explorer (Preview).
2. In the left pane, expand the storage account containing the blob container for which you wish to get a SAS.
3. Expand the storage account's **Blob Containers**.
4. Right-click the desired blob container, and - from the context menu - select **Get Shared Access Signature**.

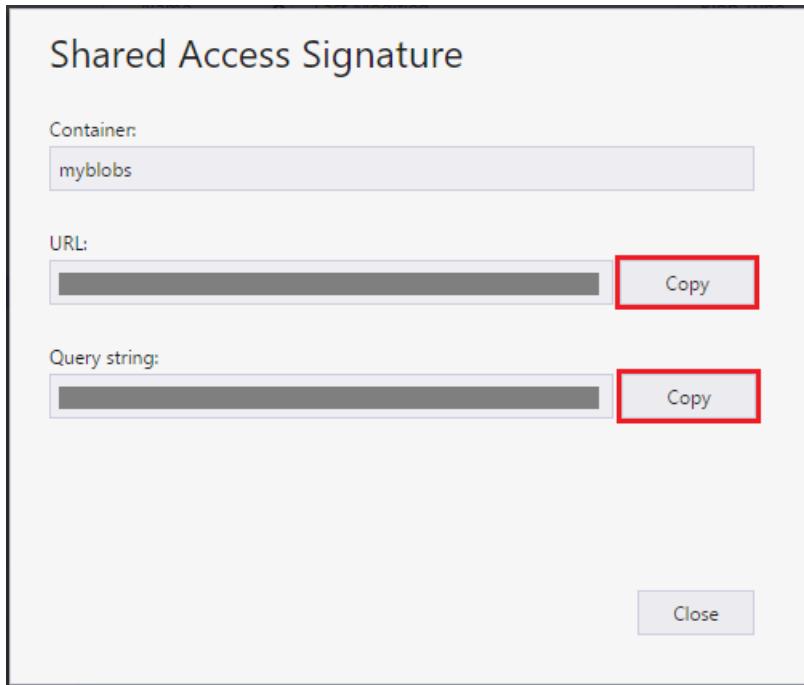


5. In the **Shared Access Signature** dialog, specify the policy, start and expiration dates, time zone, and access levels you want for the resource.



6. When you're finished specifying the SAS options, select **Create**.

7. A second **Shared Access Signature** dialog will then display that lists the blob container along with the URL and QueryStrings you can use to access the storage resource. Select **Copy** next to the URL you wish to copy to the clipboard.

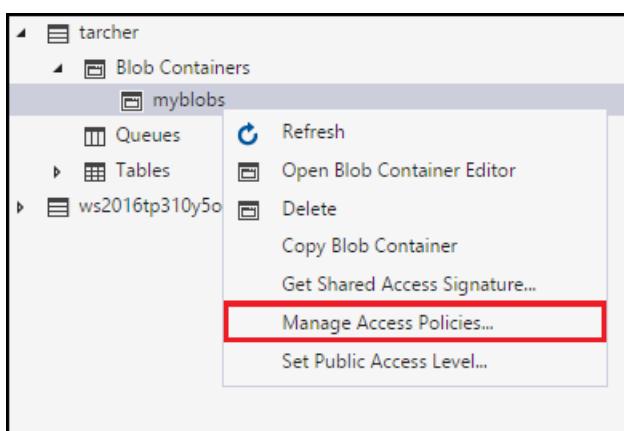


8. When done, select **Close**.

Manage Access Policies for a blob container

The following steps illustrate how to manage (add and remove) access policies for a blob container:

1. Open Storage Explorer (Preview).
2. In the left pane, expand the storage account containing the blob container whose access policies you wish to manage.
3. Expand the storage account's **Blob Containers**.
4. Select the desired blob container, and - from the context menu - select **Manage Access Policies**.



5. The **Access Policies** dialog will list any access policies already created for the selected blob container.

Access Policies

Container: myblobs

Access policies:

ID	Start time	Expiry time	Read	Write	Delete	List
myblobs-154C45ACC26	05/18/2016 07:52 AM	05/25/2016 07:52 AM	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Add Remove Save Cancel

6. Follow these steps depending on the access policy management task:

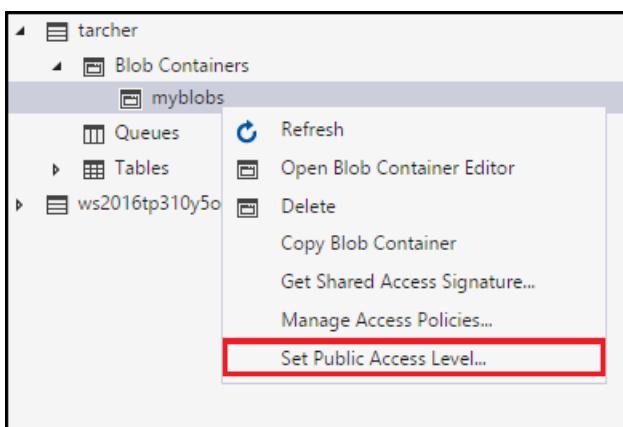
- **Add a new access policy** - Select **Add**. Once generated, the **Access Policies** dialog will display the newly added access policy (with default settings).
- **Edit an access policy** - Make any desired edits, and select **Save**.
- **Remove an access policy** - Select **Remove** next to the access policy you wish to remove.

Set the Public Access Level for a blob container

By default, every blob container is set to "No public access".

The following steps illustrate how to specify a public access level for a blob container.

1. Open Storage Explorer (Preview).
2. In the left pane, expand the storage account containing the blob container whose access policies you wish to manage.
3. Expand the storage account's **Blob Containers**.
4. Select the desired blob container, and - from the context menu - select **Set Public Access Level**.



5. In the **Set Container Public Access Level** dialog, specify the desired access level.

Set Container Public Access Level

Choose the access level:

- No public access
- Public read access for container and blobs
- Public read access for blobs only

[Apply](#)

[Cancel](#)

6. Select **Apply**.

Managing blobs in a blob container

Once you've created a blob container, you can upload a blob to that blob container, download a blob to your local computer, open a blob on your local computer, and much more.

The following steps illustrate how to manage the blobs (and folders) within a blob container.

1. Open Storage Explorer (Preview).
2. In the left pane, expand the storage account containing the blob container you wish to manage.
3. Expand the storage account's **Blob Containers**.
4. Double-click the blob container you wish to view.
5. The main pane will display the blob container's contents.

The screenshot shows the Storage Explorer interface. The toolbar at the top includes: Upload, Download, Open, Copy URL, Select all, Copy, Paste, Delete, and Refresh. Below the toolbar, a search bar contains 'myblobs'. A table lists the contents of the blob container:

Name	Last Modified	Blob Type	Content Type	Size
reachin.png	Tue, 17 May 2016 20:21:44 GMT	Block Blob	image/png	437.8 KB

At the bottom of the main pane, it says "Showing 1 to 1 of 1 loaded items."

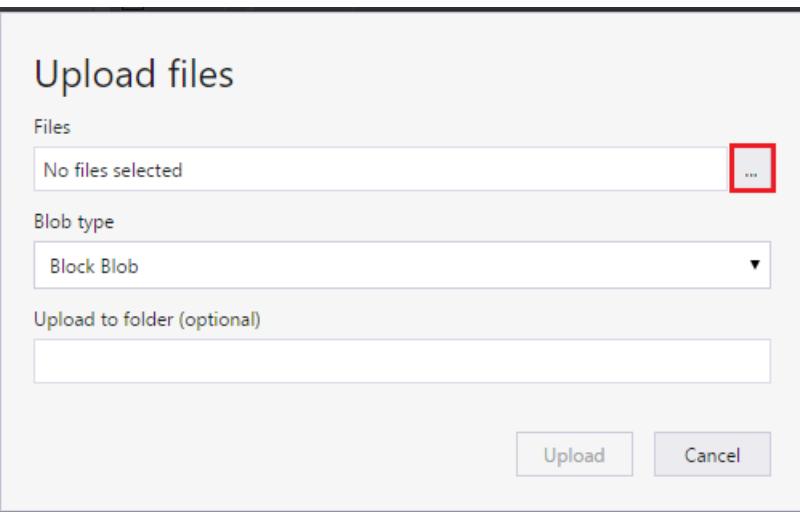
6. The main pane will display the blob container's contents.
7. Follow these steps depending on the task you wish to perform:

- **Upload files to a blob container**

- a. On the main pane's toolbar, select **Upload**, and then **Upload Files** from the drop-down menu.

The screenshot shows the Storage Explorer toolbar. The 'Upload' button is highlighted with a red box. The other buttons in the toolbar are: Download, Open, Copy URL, Select all, Copy, Paste, Delete, and Refresh. Below the toolbar, there are two options: 'Upload Files...' and 'Upload Folder...'. The 'Upload Files...' option is also highlighted with a red box.

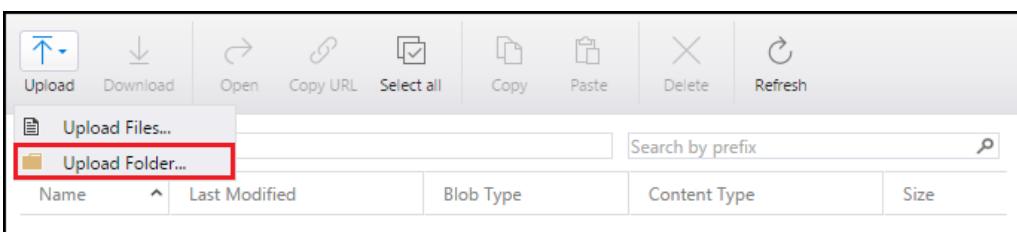
- b. In the **Upload files** dialog, select the ellipsis (...) button on the right side of the **Files** text box to select the file(s) you wish to upload.



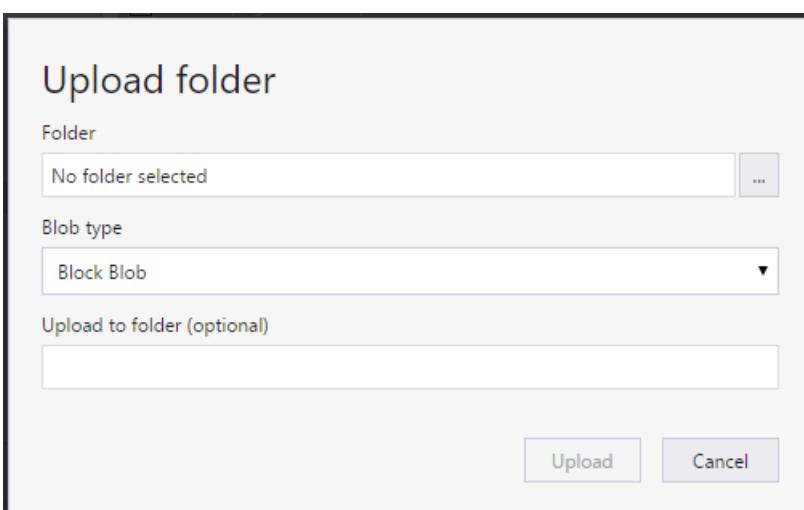
- c. Specify the type of **Blob type**. The article [Get started with Azure Blob storage using .NET](#) explains the differences between the various blob types.
- d. Optionally, specify a target folder into which the selected file(s) will be uploaded. If the target folder doesn't exist, it will be created.
- e. Select **Upload**.

- **Upload a folder to a blob container**

- a. On the main pane's toolbar, select **Upload**, and then **Upload Folder** from the drop-down menu.



- b. In the **Upload folder** dialog, select the ellipsis (...) button on the right side of the **Folder** text box to select the folder whose contents you wish to upload.



- c. Specify the type of **Blob type**. The article [Get started with Azure Blob storage using .NET](#) explains the differences between the various blob types.
- d. Optionally, specify a target folder into which the selected folder's contents will be uploaded. If the target folder doesn't exist, it will be created.
- e. Select **Upload**.

- **Download a blob to your local computer**

- a. Select the blob you wish to download.
- b. On the main pane's toolbar, select **Download**.
- c. In the **Specify where to save the downloaded blob** dialog, specify the location where you want the blob downloaded, and the name you wish to give it.
- d. Select **Save**.

- **Open a blob on your local computer**

- a. Select the blob you wish to open.
- b. On the main pane's toolbar, select **Open**.
- c. The blob will be downloaded and opened using the application associated with the blob's underlying file type.

- **Copy a blob to the clipboard**

- a. Select the blob you wish to copy.
- b. On the main pane's toolbar, select **Copy**.
- c. In the left pane, navigate to another blob container, and double-click it to view it in the main pane.
- d. On the main pane's toolbar, select **Paste** to create a copy of the blob.

- **Delete a blob**

- a. Select the blob you wish to delete.
- b. On the main pane's toolbar, select **Delete**.
- c. Select **Yes** to the confirmation dialog.

Next steps

- View the [latest Storage Explorer \(Preview\) release notes and videos](#).
- Learn how to [create applications using Azure blobs, tables, queues, and files](#).