II.2 Develop solutions that use blob storage

lundi 17 janvier 2022 14:15

When to use Blob storage

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

Type of Storage accounts

- **Standard:** This is the standard general-purpose v2 account and is recommended for most scenarios using Azure Storage.
- **Premium:** Premium accounts offer higher performance by using solid-state drives. If you create a premium account you can choose between three account types, block blobs, page blobs, or file shares.

The following table describes the types of storage accounts recommended by Microsoft for most scenarios using Blob storage.

Storage account type	Supported storage services	Usage
Standard general-purpose v2	Blob, Queue, and Table storage, Azure Files	Standard storage account type for blobs, file shares, queues, and tables. Recommended for most scenarios using Azure Storage. If you want support for NFS file shares in Azure Files, use the premium file shares account type.
Premium block blobs	Blob storage	Premium storage account type for block blobs and append blobs. Recommended for scenarios with high transactions rates, or scenarios that use smaller objects or require consistently low storage latency.
Premium page blobs	Page blobs only	Premium storage account type for page blobs only.
Premium file shares	Azure Files	Premium storage account type for file shares only.

NFS = network file system

+ change feed support

Access tiers for block blob data

- Hot : frequent access, default one.
- Cool : store large amount of data, infrequently accessed & stored for at least 30 days
- Archive : Only for individual block blobs. Tolerate several hours of retrieval latency, remain for at least 180 days. Minimum GPv2.

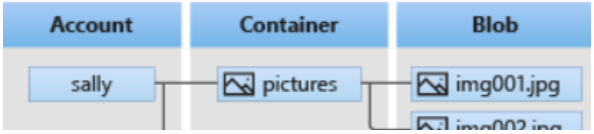
=> Can be changed at any time.

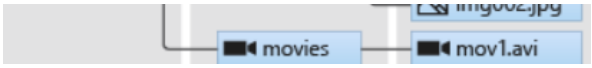
Type of resources

- **Storage account** : Allows you to formalize the requirements in terms of data storage in the cloud. Choice of **region, billing management, redundancy**. It groups together a set of azure storage services to which a set of parameters can be applied. A storage account is included in a resource group. Azure SQL and Cosmos DB are not included in a storage account. It provides a single namespace in Azure for data. Each piece of data has an address. For example, the default endpoint for a storage blob in the storage account mystorageaccount is http://mystorageaccount.blob.core.windows.net

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

- **Blobs** :
  - **Block blobs** store text and binary data. Block blobs are made up of blocks of data that can be managed individually. Block blobs can store up to about 190.7 TiB.
  - **Append blobs** are made up of blocks like block blobs, but are optimized for append operations. Append blobs are ideal for scenarios such as logging data from virtual machines.
  - **Page blobs** store random access files up to 8 TiB in size. Page blobs store virtual hard drive (VHD) files and serve as disks for Azure virtual machines.





Security

- All data are encrypted by default, whatever the performance tier (Standard or Premium)
- You can use Azure AD & RBAC on all operations on data
- Data can be secured in transit (HTTPS)
- OS & data disks are encrypted
- Delegated access to the data objects in Azure Storage can be granted using a shared access signature.

Redundancy

- Primary region
  - LRS : copy x3 in the same single physical location(=datacenter)
  - ZRS : copy x3 in the primary region (different datacenters in the same region)
- Secondary region
  - GRS : LRS in primary region + LRS in secondary region
  - GZRS : ZRS in primary region + LRS in secondary region

Create Storage account

```
az storage account create --resource-group az204-blob-rg --name \
<myStorageAcct> --location <myLocation> \
--kind BlockBlobStorage --sku Premium_LRS
```

Move items in Blob storage between storage accounts or containers

Connect to Azure blob storage

```
using Azure;
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;
using Azure.Storage.Sas;

...

// The variable sourceConnection is a string holding the connection string for the storage account
BlobServiceClient sourceClient = new BlobServiceClient(sourceConnection);
```

Create a container

```
//Create a unique name for the container
string containerName = "wtblob" + Guid.NewGuid().ToString();

// Create the container and return a container client object
BlobContainerClient containerClient = await blobServiceClient.CreateBlobContainerAsync(containerName);
Console.WriteLine("A container named '" + containerName + "' has been created. " +
    "\nTake a minute and verify in the portal." +
    "\nNext a file will be created and uploaded to the container.");
Console.WriteLine("Press 'Enter' to continue.");
Console.ReadLine();
```

Download a blob

```
// The variable sourceContainer is a string holding the container name
BlobContainerClient sourceBlobContainer = sourceClient.GetBlobContainerClient(sourceContainer);
sourceBlobContainer.CreateIfNotExists();

BlobClient sourceBlob = blobContainer.GetBlobClient("MyBlob.doc");
await sourceBlob.DownloadToAsync("MyFile.doc");

BlobProperties properties = (await sourceBlob.GetPropertiesAsync()).Value;
Console.WriteLine($"Last modified: {properties.LastModified}");
```

Upload a blob

```
// The variable destClient is a blob service client for the destination storage account
// The variable destContainer is a string holding the container name
BlobContainerClient destBlobContainer = destClient.GetBlobContainerClient(destContainer);
destBlobContainer.CreateIfNotExists();
```

```
destBlobContainer.CreateIfNotExists();

BlobClient destBlob = destContainer.GetBlobClient("NewBlob.doc");
await destBlob.UploadAsync("MyFile.doc");
```

Copy blobs between storage accounts

```
...

// The variable sourceBlob is a blob client representing the source blob
BlobClient destBlob = destContainer.GetBlobClient(sourceBlob.Name);
CopyFromUriOperation ops = await destBlob.StartCopyFromUriAsync(GetSharedAccessUri(sourceBlob.Name, sourceContainer));

...

// Create a SAS token for the source blob, to enable it to be read by the StartCopyFromUriAsync method
private static Uri GetSharedAccessUri(string blobName, BlobContainerClient container)
{
    DateTimeOffset expiredOn = DateTimeOffset.UtcNow.AddMinutes(60);

    BlobClient blob = container.GetBlobClient(blobName);
    Uri sasUri = blob.GenerateSasUri(BlobSasPermissions.Read, expiredOn);

    return sasUri;
}
```

```
// Display the status of the blob as it is copied
while(ops.HasCompleted == false)
{
    long copied = await ops.WaitForCompletionAsync();

    Console.WriteLine($"Blob: {destBlob.Name}, Copied: {copied} of {properties.ContentLength}");
    await Task.Delay(500);
}

Console.WriteLine($"Blob: {destBlob.Name} Complete");
```

Delete a blob

```
bool blobExisted = await sourceBlob.DeleteIfExistsAsync();
```

Iterate blobs in a container

```
// Iterate through the blobs in a container
List<BlobItem> segment = await blobContainer.GetBlobsAsync(prefix: "").ToListAsync();
foreach (BlobItem blobItem in segment)
{
    BlobClient blob = blobContainer.GetBlobClient(blobItem.Name);
    // Check the source file's metadata
    Response<BlobProperties> propertiesResponse = await blob.GetPropertiesAsync();
    BlobProperties properties = propertiesResponse.Value;

    // Check the last modified date and time
    // Add the blob to the list if has been modified since the specified date and time
    if (DateTimeOffset.Compare(properties.LastModified.ToUniversalTime(), transferBlobsModifiedSince.ToUniversalTime()) > 0)
    {
        blobList.Add(blob);
    }
}
```

Set and retrieve properties and metadata

Get properties on a blob

```
private static async Task GetBlobPropertiesAsync(BlobClient blob)
{
    try
    {
        // Get the blob properties
        BlobProperties properties = await blob.GetPropertiesAsync();

        // Display some of the blob's property values
        Console.WriteLine($" ContentLanguage: {properties.ContentLanguage}");
```

```
        Console.WriteLine($" ContentType: {properties.ContentType}");
        Console.WriteLine($" CreatedOn: {properties.CreatedOn}");
        Console.WriteLine($" LastModified: {properties.LastModified}");
    }
    catch (RequestFailedException e)
    {
        Console.WriteLine($"HTTP error code {e.Status}: {e.ErrorCode}");
        Console.WriteLine(e.Message);
        Console.ReadLine();
    }
}
```

Set properties on a blob

```
public static async Task SetBlobPropertiesAsync(BlobClient blob)
{
    Console.WriteLine("Setting blob properties...");

    try
    {
        // Get the existing properties
        BlobProperties properties = await blob.GetPropertiesAsync();

        BlobHttpHeaders headers = new BlobHttpHeaders
        {
            // Set the MIME ContentType every time the properties
            // are updated or the field will be cleared
            ContentType = "text/plain",
            ContentLanguage = "en-us",

            // Populate remaining headers with
            // the pre-existing properties
            CacheControl = properties.CacheControl,
            ContentDisposition = properties.ContentDisposition,
            ContentEncoding = properties.ContentEncoding,
            ContentHash = properties.ContentHash
        };

        // Set the blob's properties.
        await blob.SetHttpHeadersAsync(headers);
    }
    catch (RequestFailedException e)
    {
        Console.WriteLine($"HTTP error code {e.Status}: {e.ErrorCode}");
        Console.WriteLine(e.Message);
        Console.ReadLine();
    }
}
```

Read metadata on a blob

```
public static async Task ReadBlobMetadataAsync(BlobClient blob)
{
    try
    {
```

```
        // Get the blob's properties and metadata.
        BlobProperties properties = await blob.GetPropertiesAsync();

        Console.WriteLine("Blob metadata:");

        // Enumerate the blob's metadata.
        foreach (var metadataItem in properties.Metadata)
        {
            Console.WriteLine($"\tKey: {metadataItem.Key}");
            Console.WriteLine($"\tValue: {metadataItem.Value}");
        }
    }
    catch (RequestFailedException e)
    {
        Console.WriteLine($"HTTP error code {e.Status}: {e.ErrorCode}");
        Console.WriteLine(e.Message);
        Console.ReadLine();
    }
}
```

Set metadata on a blob

```
public static async Task AddBlobMetadataAsync(BlobClient blob)
{
    Console.WriteLine("Adding blob metadata...");

    try
    {
        IDictionary<string, string> metadata =
           new Dictionary<string, string>();

        // Add metadata to the dictionary by calling the Add method
        metadata.Add("docType", "textDocuments");

        // Add metadata to the dictionary by using key/value syntax
        metadata["category"] = "guidance";

        // Set the blob's metadata.
        await blob.SetMetadataAsync(metadata);
    }
    catch (RequestFailedException e)
    {
        Console.WriteLine($"HTTP error code {e.Status}: {e.ErrorCode}");
        Console.WriteLine(e.Message);
        Console.ReadLine();
    }
}
```

Implement storage policies, and data archiving and retention

Storage policies

```
async static Task CreateStoredAccessPolicyAsync(string containerName)
{
    string connectionString = "";

    // Use the connection string to authorize the operation to create the access policy.
    // Azure AD does not support the Set Container ACL operation that creates the policy.
    BlobContainerClient containerClient = new BlobContainerClient(connectionString, containerName);
```

```
    try
    {
        await containerClient.CreateIfNotExistsAsync();

        // Create one or more stored access policies.
        List<BlobSignedIdentifier> signedIdentifiers = new List<BlobSignedIdentifier>
        {
            new BlobSignedIdentifier
            {
                Id = "mysignedidentifier",
                AccessPolicy = new BlobAccessPolicy
                {
                    StartsOn = DateTimeOffset.UtcNow.AddHours(-1),
                    ExpiresOn = DateTimeOffset.UtcNow.AddDays(1),
                    Permissions = "rw"
                }
            }
        };
        // Set the container's access policy.
        await containerClient.SetAccessPolicyAsync(permissions: signedIdentifiers);
    }
    catch (RequestFailedException e)
    {
        Console.WriteLine(e.ErrorCode);
        Console.WriteLine(e.Message);
    }
    finally
    {
        await containerClient.DeleteAsync();
    }
}
```

Data archiving

The Archive tier is an offline tier for storing data that is rarely accessed. The Archive access tier has the **lowest storage cost, but higher data retrieval costs and latency** compared to the Hot and Cool tiers. Example usage scenarios for the Archive access tier include:

- Long-term backup, secondary backup, and archival datasets
- Original (raw) data that must be preserved, even after it has been processed into final usable form
- Compliance and archival data that needs to be stored for a long time and is hardly ever accessed

Data must remain in the Archive tier for at least 180 days or be subject to an early deletion charge. For example, if a blob is moved to the Archive tier and then deleted or moved to the Hot tier after 45 days, you'll be charged an early deletion fee equivalent to 135 (180 minus 45) days of storing that blob in the Archive tier.

While a blob is in the Archive tier, it can't be read or modified. To read or download a blob in the Archive tier, you must first rehydrate it to an online tier, either Hot or Cool. Data in the Archive tier can take up to 15 hours to rehydrate, depending on the priority you specify for the rehydration operation.

An archived blob's metadata remains available for read access, so that you can list the blob and its properties, metadata, and index tags. Metadata for a blob in the Archive tier is read-only, while blob index tags can be read or written. Snapshots are not supported for archived blobs.

The following operations are supported for blobs in the Archive tier:

- Copy Blob
- Delete Blob
- Find Blobs by Tags
- Get Blob Metadata
- Get Blob Properties
- Get Blob Tags
- List Blobs
- Set Blob Tags
- Set Blob Tier

Note

- The Archive tier is not supported for ZRS, GZRS, or RA-GZRS(Read access) accounts.
- Migrating from LRS to GRS is supported as long as no blobs were moved to the Archive tier while the account was set to LRS.
- An account can be moved back to GRS if the update is performed less than 30 days from the time the account became LRS, and no blobs were moved to the Archive tier while the account was set to LRS.

Retention

Azure Blob Storage lifecycle management offers a rich, rule-based policy which you can use to transition your data to the best access tier and to expire data at the end of its lifecycle.

Lifecycle management policy helps you:

- Transition blobs to a cooler storage tier such as hot to cool, hot to archive, or cool to archive in order to optimize for performance and cost

- Delete blobs at the end of their lifecycles
- Define up to 100 rules
- Run rules automatically once a day
- Apply rules to containers or specific subset of blobs, up to 10 prefixes per rule

Example

Consider a data set that is accessed frequently during the first month, is needed only occasionally for the next two months, is rarely accessed afterwards, and is required to be expired after seven years. In this scenario, hot storage is the best tier to use initially, cool storage is appropriate for occasional access, and archive storage is the best tier after several months and before it is deleted seven years later.

The following sample policy manages the lifecycle for such data. It applies to block blobs in container “foo”:

- Tier blobs to cool storage 30 days after last modification
- Tier blobs to archive storage 90 days after last modification
- Delete blobs 2,555 days (seven years) after last modification
- Delete blob snapshots 90 days after snapshot creation

```
{
   "rules": [
     {
       "name": "ruleFoo",
       "enabled": true,
       "type": "Lifecycle",
       "definition": {
         "filters": {
           "blobTypes": [ "blockBlob" ],
           "prefixMatch": [ "foo" ]
         },
         "actions": {
           "baseBlob": {
             "tierToCool": { "daysAfterModificationGreaterThan": 30 },
             "tierToArchive": { "daysAfterModificationGreaterThan": 90 },
             "delete": { "daysAfterModificationGreaterThan": 2555 }
           },
           "snapshot": {
             "delete": { "daysAfterCreationGreaterThan": 90 }
           }
         }
       }
     }
   ]
}
```

Add a lifecycle management policy with Azure CLI :

```
az storage account management-policy create \
    --account-name <storage-account> \
    --policy @policy.json \
    --resource-group <resource-group>
```

Change feed support

Goal : provide **ordered**, **guaranteed**, **durable**, **immutable**, **read-only** transaction logs of all the changes that occur to the blobs and the blob metadata in your storage account.

When to use : well-suited for scenarios that process data based on objects that have changed.

How it works :

- Stored in a special container in your storage account.
- Change events are appended to the change feed in Avro format.
- Any number of client app can independently read the change feed in parallel, and at their own pace.

Enable the change feed :

- 1 change feed for the blob service in each storage account. Stored in **$blobchangefeed** container.
- Change captures only at the **blob service** level.
- Client app can filter out event types as required.
- Can only be enabled on Standard General-purpose V2, premium block blob and blob storage accounts.