WINDOWS AZURE BLOB

Brad Calder, Tony Wang, Shane Mainali, and Jason Wu
May 2009

Table of Contents

1 Introduction

Windows Azure is the foundation of Microsoft’s Cloud Platform. It is the “Operating System for the Cloud” that provides essential building blocks for application developers to write scalable and highly available services. Windows Azure provides:

- Virtualized Computation
- Scalable Storage

- Automated Management
- Rich Developer SDK

Windows Azure Storage allows application developers to store their data in the cloud, so the application can access its data from anywhere at any time, store any amount of data and for any length of time, and be confident that the data is durable and will not be lost. Windows Azure Storage provides a rich set of data abstractions:

- Windows Azure Blob – provides storage for large data items.
- Windows Azure Table – provides structured storage for maintaining service state.
- Windows Azure Queue – provides asynchronous work dispatch to enable service communication.

To use Windows Azure Storage a user needs to create a storage account. This is done via the Windows Azure portal web interface. The user will receive a 256-bit secret key once the account is created. This secret key is then used to authenticate user requests to the storage system. Specifically, a HMAC SHA256 signature for the request is created using this secret key. The signature is passed with each request to authenticate the user requests by verifying the HMAC signature.

This document describes Windows Azure Blob, and how to use it. Windows Azure Blob enables applications to store large objects, up to 50GB each in the cloud. It supports a massively scalable blob system, where hot blobs will be served from many servers to scale out and meet the traffic needs of your application. Furthermore, the system is highly available and durable. You can always access your data from anywhere at any time, and the data is replicated at least 3 times for durability. In addition, strong consistency is provided to ensure that the object is immediately accessible once it is added or updated; a subsequent read will immediately see the changes made from a previously committed write.

2 Data Model

The figure below depicts the namespace of Windows Azure Blob.

- **Storage Account** – All access to Windows Azure Storage is done through a storage account.
    - This is the highest level of the namespace for accessing blobs
    - An account can have many Blob Containers

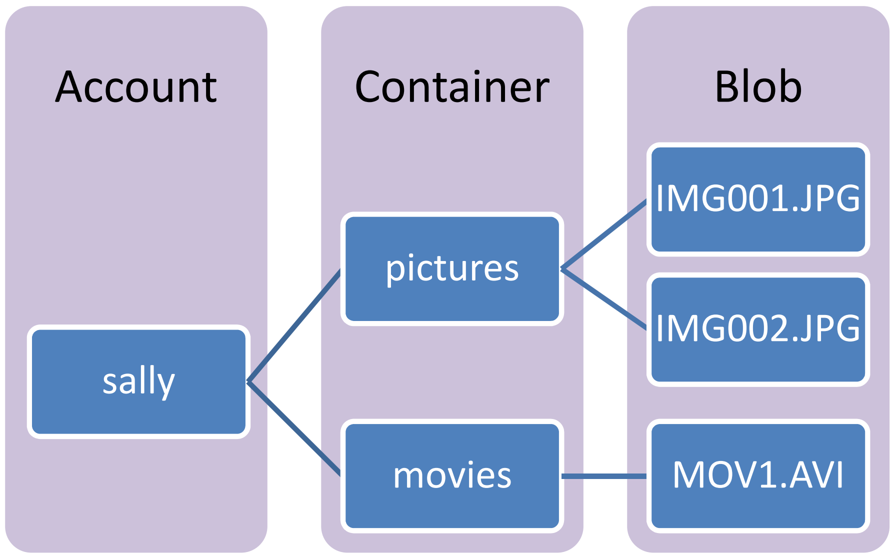


Figure 1 Blob Storage Concepts

- **Blob Container** – A container provides a grouping of a set of blobs. The container name is scoped by the account.
  - Sharing policies are set at the container level. Currently "Public READ" and "Private" are supported. When a container is "Public READ", all its contents can be read by anyone without requiring authentication. When a container is "Private", only the owner of the corresponding account can access the blobs in that container with authenticated access.
  - Containers can also have metadata associated with them. Metadata is in the form of <name, value> pairs, and they are up to 8KB in size per container.
  - The ability to list all of the blobs within the container is also provided.
- **Blob** – Blobs are stored in and scoped by Blob Containers. Each blob can be up to 50GB. A blob has a unique string name within the container. Blobs can have metadata associated with them, which are <name, value> pairs, and they are up to 8KB in size per blob. The blob metadata can be gotten and set separately from the blob data bits.

The above namespace is used to perform all access to Windows Azure Blob. The URI for a specific blob is structured as follows:

http://<account>.**blob**.core.windows.net/<container>/<blobname>

The storage account name is specified as the first part of the hostname followed by the keyword “blob”. This sends the request to the part of Windows Azure Storage that handles blob requests. The host name is followed by the container name, followed by “/”, and then the blob name. Accounts and containers have naming restrictions (see the SDK document for details), for example, the container name cannot contain a “/”.

A few notes on containers:

- Containers are scoped by accounts as described above. The storage system handles containers in a distributed manner, and there is no centralize resource bottleneck in terms of dealing with containers. The goal is to allow container operations to be on the high availability code paths of your application.
- There can be a delay when recreating a recently deleted container, especially when there were a large number of blobs in that container. The system needs to reclaim the blobs in that container before the same container name can be created again. While the server is deleting all of the blobs, recreating the container will fail with an error indicating the container is being deleted.
- When an application deletes a container or creates a brand new container, these commands are quickly committed on the server with acknowledgement back to the application, even though the delete can go on for awhile. Therefore, these can be on the high availability code paths for an application.

3 Blob REST Interface

All access to Windows Azure Blob is done through a standard HTTP REST PUT/GET/DELETE interface.

The HTTP/REST commands supported to implement the blob operations include:

- PUT Blob - Insert a new blob or overwrite an existing blob of the given name
- GET Blob - Get an entire blob, or get a range of bytes within the blob using the standard HTTP range GET operation.
- DELETE Blob - Delete an existing blob.
- Copy Blob - Copy a blob from a source blob to a destination blob within the same storage account. This copies the whole committed blob, including the blob metadata, properties, and committed blocklist. You can use CopyBlob along with DeleteBlob to rename a blob, to move a blob between containers, and to create backup copies of your existing blobs.
- Get Block List - Retrieve the list of blocks that have been uploaded as part of a blob. There are two block lists maintained for a blob, and this function allows retrieval of either of the two or both:
    - Committed Block List - This is the list of blocks that have been successfully committed as part of a PutBlockList for a given blob.
    - Uncommitted Block List - This is the list of blocks that have been uploaded for a blob since the last PutBlockList for the blob. These blocks represent the temporary/uncommitted blocks that have not yet been committed.

All of these operations can be done on a blob with the following URL:

http://<account>.**blob**.core.windows.net/<container>/<blobname>

You can upload a blob up to 64MB in size using a single PUT blob request up into the cloud. To go up to the 50GB blob size limit, one must use the block interface, which is described next.

See the SDK document for the complete definition of the REST APIs.

3.1 Versioning

For all of the Windows Azure Storage solutions, we have introduced a new HTTP header called "x-ms-version". All changes to the storage APIs will be versioned by this header. This allows prior versions of commands executed against the storage system to continue to work, as we extend the capabilities of the existing commands and introduce new commands.

The x-ms-version should be specified for all requests coming to Windows Azure Storage. If there is an anonymous request without a version, then the oldest support version of that command will be executed by the storage system.

By PDC 2009, we plan to require the x-ms-version to be specified by all non-anonymous commands. Until then, if there is not a version specified for a given request, we will assume that the version of the command the request wants to execute is the CTP version of the Windows Azure Storage APIs from PDC 2008. If a request comes in with an invalid x-ms-version, it will be rejected.

The current supported version is "x-ms-version: 2009-04-14". This can be used for all commands and requests sent to Windows Azure Storage. The new functionality we introduce for Windows Azure Blobs with this version is CopyBlob and the new version of the GetBlockList. This means that CopyBlob cannot be used unless the version header is provided. In addition, to use the new format and version of GetBlockList, the version 2009-04-14 must be specified. If that is not specified for GetBlockList, then the CTP 2008 version of GetBlockList will be used, which does not support returning the uncommitted block list.

4 A Blob as a List of Blocks

One of the target scenarios for Windows Azure Blob is to enable efficient upload of blobs that are many GBs in size. This is provided by Windows Azure Blob through the following steps:

- Break the Blob (e.g., Movie.avi) to be uploaded into contiguous blocks. For example, a 10GB movie can be broken up into 2500 blocks, each of size 4MB, where the first block represents bytes 1 through 4194304, the second block would be bytes 4194305 through 8388608, etc.
- Give each block a unique ID/name. This unique ID is scoped by the blob name being uploaded. For example, the first block could be called "Block 0001", the second block "Block 0002", etc.
- PUT each block into the cloud. This is done by doing a PUT specifying the URL above with the query specifying that this is a PUT block along with the block ID. In continuing our example, to put the first block, the blob name would be "Movie.avi", and the block ID is "Block 0001".
- After all of the blocks are stored in Windows Azure Storage, then we commit the list of uncommitted blocks uploaded to represent the blob name they were associated with. This is done with a PUT specifying the URL above with the query specifying that this is a blocklist command. Then the HTTP header contains the list of blocks to be committed for this blob. When this operation succeeds, the list of blocks, in the order in which they were listed, now

represents the readable version of the blob. The blob can then be read using the GET blob commands described above.

The following figure incorporates blocks into the Windows Azure Blob data concepts.

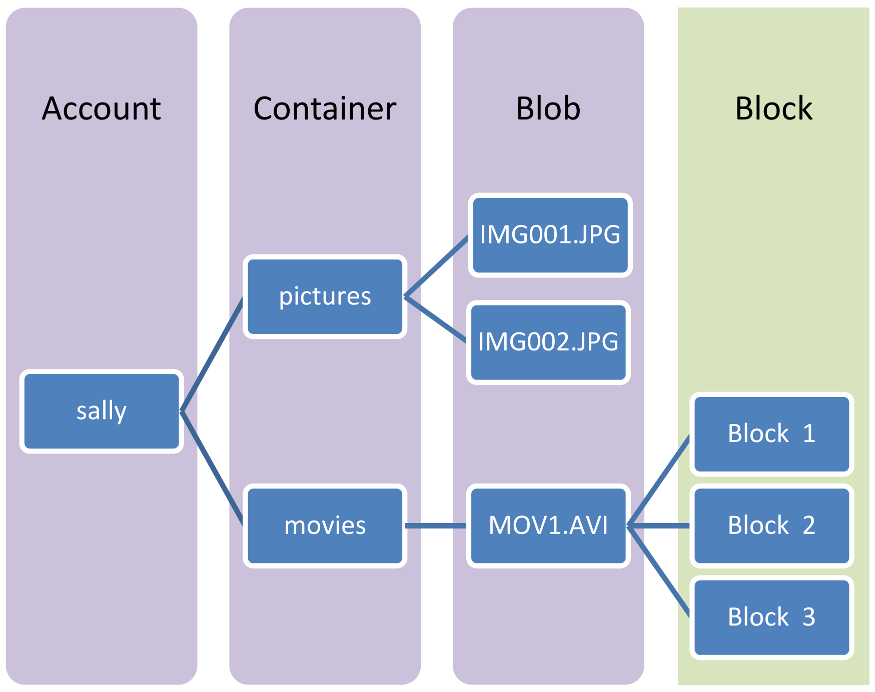


Figure 2 Blob Storage Concepts - Adding Blocks

As described earlier, blobs can be accessed via PUT and GET by using the following URL:

http://<account>.**blob**.core.windows.net/<container>/<blobname>

In the examples shown in Figure2, a single PUT can be used to put the images with the following URLs:

http://sally.**blob**.core.windows.net/pictures/IMG001.JPG

http://sally.**blob**.core.windows.net/pictures/IMG002.JPG

The same URLs can be used to get the blobs. In using a single PUT, blobs up to 64MB can be stored. To store blobs larger than 64MB and up to 50GB, one needs to first PUT all of the blocks, and then PUT the block list to comprise the readable version of the blob. In Figure 2 above, only after the blocks have been put and committed as part of the block list can the blob be read using the following URL:

http://sally.**blob**.core.windows.net/pictures/MOV1.AVI

GET operations always operate on the blob level, and do not involve specifying blocks.

4.1 Block Data Abstractions

Each block is identified by a Block ID, which is up to 64 bytes in size, and it is scoped by the blob name. So different blobs can have blocks with the same IDs. Blocks are immutable. Each block is up to 4MB in size, and the blocks within the same blob can be of different sizes. Windows Azure Blob provides the following Block level operations:

- PUT block - upload a block for a blob. Note that a block that has been successfully uploaded with a PUT block operation does not become part of the blob until it is committed as part of a block list with the PUT blocklist operation.
- PUT blocklist - commit a blob by specifying the list of block IDs that make up the blob. The blocks specified in this operation must have been successfully uploaded using PUT block calls. The order of the blocks in the PUT blocklist operation, will comprise the readable version of the blob.
- GET blocklist - retrieve the block list that has been previously committed for the blob using the PUT blocklist operation. The returned block list specifies the ID and size of each block. This function can also be used to retrieve the uncommitted block list, which consists of the blocks that have been uploaded for the blob via Put Block calls but have not been committed using a PUT block list.

Note that the Block ID can be viewed as a piece of metadata that you can track for each block. One example use of block ID is to have block IDs as the hash values of the data contents of each block. In this way, one can check the data integrity of each data block retrieved from the system.

4.2 REST Request Examples

All of the examples below refer to a blob named “MOV1.avi” contained in the “movies” container under the account “sally”.

4.2.1 REST PUT Block Examples

Below is an example of a REST request for a PUT block operation for a 4MB block. Note that the PUT HTTP verb is used, and the "?comp=block" query parameter indicates that this is a PUT block operation. Then BlockID is specified. Content-MD5 can be provided to guard against network transfer errors and insure integrity. The Content-MD5 in this case is the MD5 checksum of the block data in the request. The MD5 checksum is checked on the server, and if it does not match, an error is returned. The content length specifies the size of the block data contents. There is also an authorization header inside the HTTP request header as shown below.

```
PUT http://sally.blob.core.windows.net/movies/MOV1.avi
?comp=block &blockid=BlockId1 &timeout=60
HTTP/1.1 Content-Length: 4194304
Content-MD5: HUXZLQLMuI/KZ5KDcJPcOA==
Authorization: SharedKey sally:  F5a+dUDvef+PfMb4T8Rc2jHcwfK58KecSZY+l2naIao=
x-ms-date: Mon, 6 Apr 2009 17:00:25 GMT
x-ms-version: 2009-04-14
……… Block Data Contents ………
```

4.2.2 REST PUT BlockList Examples

Below is an example of a REST request for a PUT blocklist operation. Note that the PUT HTTP verb is used, and the "?comp=blocklist" query parameter indicates that this is a PUT blocklist operation. The block list is specified inside the HTTP request body in XML format, as shown in the example below. Note that the Content-Length field in the request header corresponds to the length of the request body, not the length of the blob to be created. There is also an authorization header inside the HTTP request header as shown below.

```
PUT http://sally.blob.core.windows.net/movies/MOV1.avi
?comp=blocklist &timeout=120
HTTP/1.1 Content-Length: 161213
Authorization: SharedKey  sally: QrmowAF72IsFEs0GaNCtRU143JpkflIgRTcOdKZaYxw=
x-ms-date: Mon, 6 Apr 2009 17:00:25 GMT
x-ms-version: 2009-04-14
<?xml version="1.0" encoding="utf-8"?>
<BlockList>
  <Block>BlockId1</Block>
  <Block>BlockId2</Block>
  ………………
</BlockList>
```

4.2.3 REST GET Blob Examples

Below is an example of a REST request for a GET blob operation. The HTTP verb GET is used in this case. The request below will get the entire contents of the given blob. If the container the blob belongs to ("movies" in this example) has sharing policy set to "Private", then authentication is required to get the blob. Otherwise, if the container has a "Public-Read" sharing policy, then authentication is not required, and the request header does not need an authorization header.

```
GET  http://sally.blob.core.windows.net/movies/MOV1.avi
HTTP/1.1
Authorization: SharedKey sally:  RGllHMtzKMi4y/nedSk5Vn74IU6/fRMwiPsL+uYSDjY=
x-ms-date: Mon, 6 Apr 2009 17:00:25 GMT
x-ms-version: 2009-04-14
```

Range GET is also supported as shown in the example below to retrieve a byte range within the given blob. Note that for GET blob requests, it is optional to provide version header. The example below does not provide a version header.

```
GET http://sally.blob.core.windows.net/movies/MOV1.avi
HTTP/1.1
Range: bytes=1024000-2048000
```

4.2.4 REST Get Block List Examples

Below is an example of a REST request for a Get Block List operation for retrieving the committed block list. Note that the GET HTTP verb is used, and the "?comp=blocklist" operation indicates that this is a Get Block List operation. Then "blocklisttype=committed" is specified, indicating that the committed block list is to be retrieved. Note that if this option is not specified, the committed block list will be returned by default.

```
GET http://sally.blob.core.windows.net/movies/MOV1.avi
? comp=blocklist&blocklisttype=committed
HTTP/1.1
x-ms-date: Wed, 08 Apr 2009 00:31:26 GMT
x-ms-version: 2009-04-14
Authorization: SharedKey sally:TOfGagLfokWXYXlGJ+R2neNr1+lIeKbURzlRfZR5fso=
```

The response header looks like the following:

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/xml
Server: Blob Service Version 1.0 Microsoft-HTTPAPI/2.0Date: Wed, 08 Apr 2009 00:33:19 GMT
```

The response body contains the committed block list:

```
<?xml version="1.0" encoding="utf-8"?>
<BlockList>
        <CommittedBlocks>
                <Block>
                        <Name>BlockId001</Name>
                        <Size>4194304</Size>
                </Block>
                <Block>
                        <Name>BlockId002</Name>
                        <Size>4194304</Size>
                </Block>
                ............
        </CommittedBlocks>
</BlockList>
```

Below is an example of a REST request to retrieve the uncommitted block list:

```
GET http://sally.blob.core.windows.net/movies/MOV1.avi
&comp=blocklist&blocklisttype=uncommitted
HTTP/1.1
x-ms-date: Wed, 08 Apr 2009 00:42:36 GMT
x-ms-version: 2009-04-14
```

```
Authorization: SharedKey sally:JoqnL0VayI4+wxIGeCVgr85Vm3diHD+Cfe37re3PQqo=
```

The Get Block List call can also return both the committed and the uncommitted block list. Below is an example of such a request:

```
GET http://sally.blob.core.windows.net/movies/MOV1.avi
&comp=blocklist&blocklisttype=all
HTTP/1.1
x-ms-date: Wed, 08 Apr 2009 00:42:36 GMT
x-ms-version: 2009-04-14
Authorization: SharedKey sally:JoqnL0VayI4+wxIGeCVgr85Vm3diHD+Cfe37re3PQqo=
```

This response includes both the committed and the uncommitted block list. An example is shown below. Note that in this example, both the committed block list and the uncommitted block list have blocks in there.

```
<?xml version="1.0" encoding="utf-8"?>
<BlockList>
    <CommittedBlocks>
        <Block>
            <Name>BlockId001</Name>
            <Size>4194304</Size>
        </Block>
        <Block>
            <Name>BlockId002</Name>
            <Size>4194304</Size>
        </Block>
        ...........
    </CommittedBlocks>
    <UncommittedBlocks>
        <Block>
            <Name>BlockId003</Name>
            <Size>4194304</Size>
        </Block>
        <Block>
            <Name>BlockId004</Name>
            <Size>1024000</Size>
        </Block>
        ...........
    </UncommittedBlocks>
</BlockList>
```

When getting the committed block list, the blocks are returned in the order in which they appear in the committed blob, including duplicates of the same block that may appear in the committed blob.

When getting the uncommitted block list, the blocks are returned in the order from the most recently uploaded uncommitted block to the oldest uploaded uncommitted block associated with the blob. In addition, duplicate block IDs that were uploaded are removed from the list. For a given block ID, only the most recently uploaded block ID and its size are shown in the block list.

4.2.5 REST Copy Blob Examples

Below is an example of a REST request for Copy Blob operation.

CopyBlob provides the ability to copy the contents of source blob A to a destination blob B. In doing this copy, the entire data bits are copied along with the blob properties, metadata and the committed block list. The uncommitted block list of the source is not copied nor touched. Upon completion of a successful copy, the destination blob's uncommitted block list is erased. There is also the option of creating a new set of user metadata for the destination blob when performing the copy.

The example below copies a blob in account sally's container "movies" to another container named "media". The destination blob name is specified in the URL provided in the PUT request. The source blob name is specified inside the "x-ms-copy-source" request header as shown in the example below.

```
PUT http://sally.blob.core.windows.net/media/MOV1.avi
HTTP/1.1
x-ms-date: Wed, 08 Apr 2009 00:36:19 GMT
x-ms-copy-source: /sally/movies/MOV1.avi
x-ms-version: 2009-04-14
Authorization: SharedKey sally:KLUKJBAn2WGPSbt8Hg61JjCRFnalLUgOw1vG9kq2/tA=
```

The Copy Blob operation can also specify conditions on the source blob and/or on the destination blob. The example below is a conditional copy operation, which only succeeds if the source has been modified since Wed, 08 Apr 2009 00:37:02 GMT and the destination has not been modified since that time.

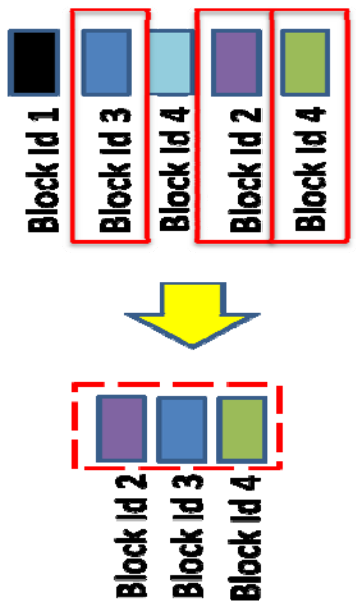
```
PUT http://sally.blob.core.windows.net/media/MOV1.avi
HTTP/1.1
x-ms-date: Wed, 08 Apr 2009 00:36:19 GMT
x-ms-copy-source: /sally/movies/MOV1.avi
x-ms-meta-comment: birthday party video
x-ms-meta-datetaken: April 5th 2009
x-ms-version: 2009-04-14
x-ms-source-if-modified-since: Wed, 08 Apr 2009 00:37:02 GMT
If-Unmodified-Since: Wed, 08 Apr 2009 00:37:02 GMT
Authorization: SharedKey sally:KLUKJBAn2WGPSbt8Hg61JjCRFnalLUgOw1vG9kq2/tA=
```

Note that, in the example above, "x-ms-meta-<xxx>" tags ("x-ms-meta-comment" and "x-ms-datetaken" in this example) reset the application metadata of the destination blob with the new application metadata. In this case, the application metadata of the destination blob will have name value pairs of <"comment", "birthday party video"> and <"datetaken", "April 5th 2009">.
If no "x-ms-meta-<xxx>" tags are specified, then the destination blob will copy the application metadata from the source blob in entirety.

4.3 Block Upload Scenarios

Uploading a blob as a list of blocks has the following benefits:

- Continuation – as each block is uploaded one can verify the success of that block and retry the block if there is a failure and continue from that point.
- Parallel Upload – one can upload the blocks in parallel to decrease the upload time of a very large blob.
- Out of Order Upload – one can even upload blocks out of order. What matters is the order of the list of blocks when doing a PUT blocklist. The list of blocks in the PUT blocklist operation specifies the order of the blocks that comprise the readable version of the blob.
- Associate BlockID as Metadata for Every Block – when you upload a blob via blocks, the committed block IDs along with their size are stored and can be retrieved later with GetBlocklist. The application can treat this as metadata that you can set and associate with every block in a stored blob. Therefore, you can store as part of the block ID your own MD5 of the block contents. Then if your application needs to check the MD5 checksum for given ranges of blob GETs, you can first GET the blocklist, and then get the blocks by getting the corresponding ranges for the blocks specified there, and then do an MD5 check on each block downloaded via the ranged GET blob.



```
PutBlock(BlockId1);
PutBlock(BlockId3);
PutBlock(BlockId4);
PutBlock(BlockId2);
PutBlock(BlockId4);
PutBlockList(BlockId2, BlockId3, BlockId4);
```

Figure 3 Block Upload Scenario

The example in Figure 4 will be used to explain the different scenarios one can encounter when using the block interface for uploading blobs. These are:

- Uploading Blocks with Same Block IDs - When blocks of the same block ID are uploaded for the same blob, the most recently uploaded block associated with that ID will be used when committing the blob with PUT blocklist. In the example above, two blocks with BlockId4 are uploaded and the latter one will be used in the final committed block list for the blob.

- Uploading Blocks Out of Order - Blocks can be uploaded in a different order as in the final committed block list in the blob. In the example above, the final committed block list has the blocks in the order of BlockId2, BlockId3, and BlockId4, but these blocks are uploaded in a different order. The readable blob data (via GET) is ordered with respect to the list specified in PUT blocklist. Therefore, if you did a get blob and read the blob from start to finish you would see the contents of Block Id 2, followed by Block ID 3 and then Block ID 4.

- Unused Blocks - Furthermore, some blocks may never get committed into the final block list in the blob. These blocks will be garbage collected by the system. BlockId1 and the first Block with ID BlockId4 in the example will be garbage collected. More specifically, once a new blob is created via PUT blocklist, all of the uncommitted blocks that have been uploaded so far, but not included in the block list specified in PUT blocklist, will be garbage collected.

A large blob may take a long time to be uploaded entirely. But in the mean time, the uploaded but uncommitted blocks do take up storage space, and someone may upload many blocks but never commit them using PUT blocklist. If there is an extended period of inactivity for that blob (currently set to a week), these uncommitted blocks will be garbage collected by the system.

An interesting scenario is when there are multiple concurrent writers to the same blob. There are two issues to discuss.

- Block IDs – If an application has multiple client writers to the same blob name via blocks, the block IDs have to be unique among the multiple writers to avoid conflicts, or the block IDs have to represent the contents of the block being written (so if multiple clients write the same block, the block will have the same ID, since it represents the same data). To provide correctness for multiple writers potentially writing the same Blob, it is recommended that the block ID be a hash (e.g., an MD5 hash) of the block contents. This way the block ID represents the contents of the block.
- First Commit Wins - When multiple clients try to upload blocks for the same blob concurrently, whoever commits the blob first via PUT blocklist (or a writer calling PUT blob) wins, and all of the other uncommitted blocks uploaded by the other writers for that blob name will be removed and garbage collected. Therefore, coordination across all the concurrent writers may be necessary to efficiently update the same blob in parallel.

4.4 Conditional PUT and GET and Optimistic Concurrency

Windows Azure Blob supports conditional PUT and GET, which can be used to efficiently handle concurrency and client caching.

Conditional PUT can be used where multiple users might update the same blob. For example, one can do a put blob conditionally on the last modification time of the blob to make sure the blob being modified is the same version that the client is modifying. This can be used to implement optimistic concurrency. For instance, two clients A and B are updating the same blob. They read the current version of the blob, make some changes to it, and then upload it back to the store. In this scenario, each of them will record the ETag of the blob they retrieved from the store. When they are ready to upload the updated version of the blob back to the store, they can use a conditional PUT blob based on the ETag they retrieved last time. The operation would specify that the PUT should be conditioned on "IF MATCH <ETag>". In this way, if the blob has been changed by a different client, the update operation will fail, and the client will be notified of this.

Conditional GET can be used to efficiently handle cache consistency issues. For example, a client has a local blob cache, which caches the hot blobs retrieved from the store. For each cached blob, its last modified time is recorded. When the client cache decides to refresh its blobs from the store, it can use conditional GET based on the modified time (on the condition being "if modified since time X"). This way, only the blobs that have been modified and are different to the cached copy need to be downloaded from the store.

4.5 Concurrent PUT and GET of Same Blob

When Get Blob and Put Blob overlap, the system provides "snapshot isolation" in the sense that the GET operation will only retrieve a single version of the blob. It will not mix the data from two different versions of the blob.

If the blob data is changed during a long GET, the GET operation can fail and return a "connection closed" error. In this case, the client should perform a conditional GET based on the ETag of the blob it tried to retrieve in order to resume the get. This allows the client to distinguish a real connection error from the case where the connection is closed due to blob data changes.

5 Blob Enumeration

The Blob system provides an interface to enumerate the blobs in a container. It supports hierarchical listing of blobs within a container. It also supports a continuation mechanism to allow enumeration of a large number of blobs.

5.1 Hierarchical Listing

The ListBlobs interface supports "prefix" and "delimiter" parameters, which are used to support hierarchical listing of blobs. For example, assume that in an account “sally”, we have a container "movies" that contains blobs with the following names:

```
Action/Rocky1.wmv
Action/Rocky2.wmv
Action/Rocky3.wmv
Action/Rocky4.wmv
Action/Rocky5.wmv
Drama/Crime/GodFather1.wmv
Drama/Crime/GodFather2.wmv
Drama/Memento.wmv
Horror/TheBlob.wmv
```

As we can see, "/" is used as a delimiter to have a directory-like hierarchy for the blob names. To list all the "directories", one can specify "delimiter=/" in the ListBlobs query, and the following will be the request and part of the response:

Request:

GET http://sally.blob.windows.net/movies?comp=list&delimiter=/

Response:

```
<BlobPrefix>Action</BlobPrefix>
<BlobPrefix>Drama</BlobPrefix>
<BlobPrefix>Horror</BlobPrefix>
```

Note that the "BlobPrefix" tag indicates that the corresponding entry is a blob name prefix, instead of a full blob name. Also note that same prefix is only returned once in the result.

As a step further, we can combine prefix with delimiter to list contents of a "sub-directory". For example, if we specify "prefix=Drama/" and "delimiter=/", we will list all the sub-directories and files inside the directory "Drama":

Request:

GET http://sally.blob.windows.net/movies?comp=list &prefix=Drama/ &delimiter=/

Response:

```
<BlobPrefix>Drama/Crime</BlobPrefix>
<Blob>Drama/Memento.wmv</Blob>
```

Note that "Drama/Memento.wmv" is a full blob name, therefore it is tagged as such.

5.2 Pagination

The ListBlobs interface allows one to specify "maxresults", which is the maximum number of results to be returned in that call. Furthermore, the system enforces an upper bound on the maximum number of results that can be returned in a single call (see the SDK document for details). When the smaller of the two limits is reached, the call returns with the corresponding results, along with an opaque "NextMarker". If this marker is not empty, then that indicates there are more results to be returned. One can use this "NextMarker" in a latter call to continue the listing onto the next page of results.

In the previous example, suppose we want to list all the blobs in the "Action" directory and every time return up to 3 results, then the first set of results would be:

Request:
    GET http://sally.blob.windows.net/movies?comp=list &prefix=Action &maxresults=3
Response:

```
<Blob>Action/Rocky1.wmv</Blob>
<Blob>Action/Rocky2.wmv</Blob>
<Blob>Action/Rocky3.wmv</Blob>
<NextMarker> OpaqueMarker1</NextMarker>
```

The first set of blobs will be returned along with an opaque marker. This opaque marker can be passed into a second ListBlobs call, which in this case will return the following results:

Request:
    GET http://sally.blob.windows.net/movies?comp=list &prefix=Action &maxresults=3 &marker=OpaqueMarker1

Response:

```
<Blob>Action/Rocky4.wmv</Blob>
<Blob>Action/Rocky5.wmv</Blob>
<NextMarker></NextMarker>
```

As shown above, the remaining blobs in the directory are returned, and "NextMarker" is set to empty, indicating that there are no more results.

6 Best Practices

When designing an application for use with Windows Azure Storage, it is important to handle errors appropriately. This section describes issues to consider when designing your application.

6.1 Retry Timeouts and “Connection closed by Host” errors

Requests that receive a Timeout or “Connection closed by Host” response might not have been processed by Windows Azure Storage. For example, if a PUT request returns a timeout, a subsequent GET might retrieve the old value or the updated value. If you see either of these responses, retry the request again with exponential back-off.

6.2 Tune Application for Repeated Timeout errors

Timeout errors can occur if there are network issues between your application and data center. Over the wide area network, it is recommended to break a single large transfer into a series of smaller calls, and design your application to handle timeouts/failures in this case so that it is able to resume after an error and continue to make progress. For example, instead of getting a 1GB blob at once, you can get a byte range of it every time and record the progress so that it can be resumed if the connection is broken or times out. For large GETs it is important to have logic in your application to continue the get starting at the byte range the GET left off if the application gets a timeout or closed connection.

Another example is that to do a listing of a large number of blobs, take advantage of the continuation support provided by Windows Azure Blob, and do a series of ListBlobs calls to get the blob lists in pages. Tune the size limit of each request based on the network link you have.

We designed the system to scale and be able to handle a large amount of traffic. However, extremely high rate of requests may lead to request timeouts. In that case, reducing your request rate may decrease or eliminate errors of this type. Generally speaking, most users will not experience these errors regularly; however, if you are experiencing high or unexpected Timeout errors, contact us via the MSDN Windows Azure forums to discuss how to optimize your use of Windows Azure Storage and prevent these types of errors in your application.

6.3 Error handling and reporting

The REST API is designed to look like a standard HTTP server and interact with existing HTTP clients (e.g., browsers, HTTP client libraries, proxies, caches, and so on). To ensure the HTTP clients handle errors properly, we map each Windows Azure Storage error to an HTTP status code.

HTTP status codes are less expressive than Windows Azure Storage error codes and contain less information about the error. Although the HTTP status codes contain less information about the error, clients that understand HTTP, but not the Windows Azure Storage errors, will usually handle the error correctly.

Therefore, when handling errors or reporting Windows Azure Storage errors to end users, use the Windows Azure Storage error code instead of the HTTP status code as it contains the most information about the error. Additionally, when debugging your application, you should also consult the human readable <ExceptionDetails> element of the XML error response.

6.4 Compressed content

Currently Windows Azure does not automatically compress data when you send it from your application to the storage in the cloud. However, your application can compress the data first and then store it in the cloud. This can give the application performance benefits in term of network bandwidth, especially when the data is highly compressible. Furthermore, remember to set the Content-Encoding header to "gzip" when uploading the blob compressed with gzip, so that when the blob is retrieved, the web clients know that the content is in a compressed form and know how to deal with it.

6.5 Use of Copy Blob

One can use CopyBlob operation to do rename. For example, to rename a blob A to a different name B within the same account, we can first copy A to B and then delete A. This can be used to efficiently rename large blobs within a storage account.

Another use of CopyBlob is to make a backupof an existing blob. One can use this operation to keep track of different "versions" of the blob by making a "copy" for each "version" of the blob before a new version of the blob is created.

The underlying data for a blob is lazily duplicated when a blob is copied, so your storage account will use more storage for each copy (backup) of a blob you make. This duplication is done lazily in the background in order to allow the CopyBlob request to execute quickly. Even though the duplication is done in the background, the source and the destination for a copy blob are treated as separate entities once the CopyBlob request succeeds.

6.6 Use of Uncommitted Get Block List

One use of GetBlockList operation is to retrieve the current uncommitted block list. It is useful in a few scenarios.

1. When uploading a large blob via PutBlock calls, if the client fails in the middle, when it comes back, it can query the current uncommitted block list and determine where to resume from.
2. A blob can be uploaded in parallel by having multiple clients putting different blocks of the blob in parallel. Using GetBlockList, one can monitor the progress of the upload by checking the current uncommitted block list.
3. If a Put Block List fails due to some blocks missing in the uncommitted block list for the blob, one can check the uncommitted block list against the expected uncommitted block list to find the missing blocks.

6.7 Being Resilient to Failures

Applications should build in the logic to resume on failures. In particular, it should track the progress of a lengthy transfer, and resume from where it left off upon failures. For example, for the upload scenarios, the system provides GetBlockList API to allow the application to retrieve the current progress in terms of the uncommitted block list, so that the application can resume. For the download scenarios, the application may keep track of how much and what parts of the data have been successfully downloaded so far, so that it can resume on failures.

6.8 Blob Container Best Practices

Blob containers provide the level of access control for user blob data for sharing purposes. For instance, one can set a container to be Public readable so that everyone can read all the contents of that container. One can also set a container to be Private, so that only the account owner can access it.

Another common use of blob containers is that some applications use a separate blob container for each of their notion of application users. In this case, the blob container abstraction provides a grouping mechanism for the applications, and allows the applications to manage each group independently.

The system has a size limit on a container, which will be in the low 10s of TBs for commercial release. The limit is not in terms of the number of blobs, but the number of bytes within a container. As a best practice, when an application has a large amount of data to store and the data can be grouped into different logical sets, we recommend applications to spread their data into different containers.