AZ-203T03

Develop for Azure storage

Contents

Module 0 Welcome to the course

Start Here

Welcome

Welcome to the **Develop for Azure storage** course. This course is part of a series of courses to help you prepare for the **AZ-203: Developing Solutions for Microsoft Azure**[1] certification exam.

Who should take this exam?

Candidates for this exam are Azure Developers who design and build cloud solutions such as applications and services. They participate in all phases of development, from solution design, to development and deployment, to testing and maintenance. They partner with cloud solution architects, cloud DBAs, cloud administrators, and clients to implement the solution.

Candidates should be proficient in developing apps and services by using Azure tools and technologies, including storage, security, compute, and communications.

Candidates must have at least one year of experience developing scalable solutions through all phases of software development and be skilled in at least one cloud-supported programming language.

Exam study areas

AZ-203 includes six study areas, as shown in the table. The percentages indicate the relative weight of each area on the exam. The higher the percentage, the more questions you are likely to see in that area.

AZ-203 Study Areas	Weight
Develop Azure Infrastructure as a Service compute solutions	10-15%
Develop Azure Platform as a Service compute solutions	20-25%
Develop for Azure storage	15-20%

1 https://www.microsoft.com/en-us/learning/exam-az-203.aspx

AZ-203 Study Areas	Weight
Implement Azure security	10-15%
Monitor, troubleshoot, and optimize Azure solutions	15-20%
Connect to and consume Azure, and third-party, services	20-25%

✓ This course will focus on preparing you for the **Develop for Azure storage** area of the AZ-203 certification exam.

Course description

In this course students will gain the knowledge and skills needed to leverage Azure storage services and features in their development solutions. It covers Azure Table storage, Azure Cosmos DB, Azure Blob, and developing against relational databases in Azure.

Level: Intermediate

Audience:

- Students in this course are interested in Azure development or in passing the Microsoft Azure Developer Associate certification exam.
- Students should have 1-2 years experience as a developer. This course assumes students know how to code and have a fundamental knowledge of Azure.
- It is recommended that students have some experience with PowerShell or Azure CLI, working in the Azure portal, and with at least one Azure-supported programming language. Most of the examples in this course are presented in C# .NET.

Course Syllabus

Module 1: Develop solutions that use Azure Table storage

- Azure Table storage overview
- Authorization in Table storage
- Table service REST API

Module 2: Develop solutions that use Azure Cosmos DB storage

- Azure Cosmos DB overview
- Managing containers and items
- Create and update documents by using code

Module 3: Develop solutions that use a relational database

- Azure SQL overview
- Create, read, update, and delete database tables by using code

Module 4: Develop solutions that use Microsoft Azure Blob storage

- Azure Blob storage overview
- Working with Azure Blob storage

Module 1 Develop solutions that use Azure Table storage

Azure Table storage overview

Introduction to Table storage in Azure

Azure Table storage is a service that stores structured NoSQL data in the cloud, providing 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 Table storage data is fast and cost-effective for many types of applications, and is typically lower in cost than traditional SQL for similar volumes of data.

You can use Table storage to store flexible datasets like user data for web applications, address books, device information, or other types of metadata 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.

Note: The content in this lesson applies to the original Azure Table storage. However, there is now a premium offering for table storage, the Azure Cosmos DB Table API that offers throughput-optimized tables, global distribution, and automatic secondary indexes.

What is Table storage

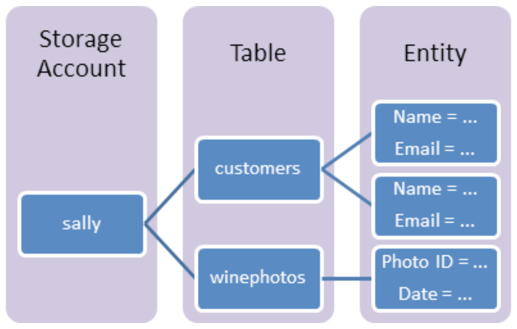
Azure Table storage stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of Table storage include:

- 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 Table storage to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table storage concepts

Table storage contains the following components:



- **URL format:** Azure Table Storage accounts use this format: `http://<storage account>.table.core.windows.net/<table>`
  - Azure Cosmos DB Table API accounts use this format: `http://<storage account>.table.cosmosdb.azure.com/<table>`
  - You can address Azure tables directly using this address with the OData protocol. For more information, see **OData.org**[1].
- **Accounts:** All access to Azure Storage is done through a storage account.
  - All access to Azure Cosmos DB is done through a Table API account.
- **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.
- **Entity:** An entity is a set of properties, similar to a database row. An entity in Azure Storage can be up to 1MB in size. An entity in Azure Cosmos DB can be up to 2MB 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 three 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.

Choosing Table storage or Cosmos DB Table API

Azure Cosmos DB Table API and Azure Table storage share the same table data model and expose the same create, delete, update, and query operations through their SDKs.

If you currently use Azure Table Storage, you gain the following benefits by moving to the Azure Cosmos DB Table API:

1 http://www.odata.org/

	Azure Table storage	Azure Cosmos DB Table API
Latency	Fast, but no upper bounds on latency.	Single-digit millisecond latency for reads and writes, backed with <10-ms latency reads and <15-ms latency writes at the 99th percentile, at any scale, anywhere in the world.
Throughput	Variable throughput model. Tables have a scalability limit of 20,000 operations/s.	Highly scalable with dedicated reserved throughput per table that's backed by SLAs. Accounts have no upper limit on throughput and support >10 million operations/s per table.
Global distribution	Single region with one optional readable secondary read region for high availability. You can't initiate failover.	Turnkey global distribution from one to 30+ regions. Support for automatic and manual failovers at any time, anywhere in the world.
Indexing	Only primary index on PartitionKey and RowKey. No secondary indexes.	Automatic and complete indexing on all properties, no index management.
Query	Query execution uses index for primary key, and scans otherwise.	Queries can take advantage of automatic indexing on properties for fast query times.
Consistency	Strong within primary region. Eventual within secondary region.	Five well-defined consistency levels to trade off availability, latency, throughput, and consistency based on your application needs.
Pricing	Storage-optimized.	Throughput-optimized.
SLAs	99.99% availability.	99.99% availability SLA for all single region accounts and all multi-region accounts with relaxed consistency, and 99.999% read availability on all multi-region database accounts Industry-leading comprehensive SLAs on general availability.

Developing with Azure Table storage

Azure Table storage has these SDKs available for development:

- WindowsAzure.Storage .NET SDK. This library enables you to work with the storage Table service.
- Python SDK. The Azure Cosmos DB Table SDK for Python also supports the storage Table service.
- Azure Storage SDK for Java. This Azure Storage SDK provides a client library in Java to consume Azure Table storage.
- Node.js SDK. This SDK provides a Node.js package and a browser-compatible JavaScript client library to consume the storage Table service.

- AzureRmStorageTable PowerShell module. This PowerShell module has cmdlets to work with storage Tables.
- Azure Storage Client Library for C++. This library enables you to build applications against Azure Storage.
- Azure Storage Table Client Library for Ruby. This project provides a Ruby package that makes it easy to access Azure storage Table services.
- Azure Storage Table PHP Client Library. This project provides a PHP client library that makes it easy to access Azure storage Table services.

Developing with the Azure Cosmos DB Table API

At this time, the Azure Cosmos DB Table API has four SDKs available for development:

- Microsoft.Azure.CosmosDB.Table .NET SDK. This library has the same classes and method signatures as the public Windows Azure Storage SDK, but also has the ability to connect to Azure Cosmos DB accounts using the Table API. Note that the `Microsoft.Azure.CosmosDB.Table` library is currently available for .NET Standard only, it's not yet available for .NET Core.
- Python SDK. The new Azure Cosmos DB Python SDK is the only SDK that supports Azure Table storage in Python. This SDK connects with both Azure Table storage and Azure Cosmos DB Table API.
- Java SDK. This Azure Storage SDK has the ability to connect to Azure Cosmos DB accounts using the Table API.
- Node.js SDK. This Azure Storage SDK has the ability to connect to Azure Cosmos DB accounts using the Table API.

Guidelines for table design

Designing tables for use with the Azure storage table service is very different from design considerations for a relational database. This lesson describes guidelines for designing your Table service solution to be read efficient and write efficient.

Design 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.

Design 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.

Design scalable and performant tables

To design scalable and performant tables, you must consider factors such as performance, scalability, and cost. If you have previously designed schemas for relational databases, these considerations are familiar, but while there are some similarities between the Azure Table service storage model and relational models, there are also important differences. These differences typically lead to different designs that may look counter-intuitive or wrong to someone familiar with relational databases, yet make sense if you are designing for a NoSQL key/value store such as the Azure Table service. Many of your design differences reflect the fact that the Table service is designed to support cloud-scale applications that can contain billions of entities (or rows in relational database terminology) of data or for datasets that must support high transaction volumes. Therefore, you must 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. Although the focus of this lesson is on the Table service, it includes discussion of the Azure Queue and Blob services, and how you might use them with the Table service.

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. The timestamp is applied 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 LMT. This document uses 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.

PartitionKey	RowKey	Timestamp	
Marketing	00001	2014-08-22T00:50:32Z	FirstName: Don LastName: Hall Age: 34 Email: donh@contoso.com
Marketing	00002	2014-08-22T00:50:34Z	FirstName: Jun LastName: Cao Age: 47 Email: junc@contoso.com
Marketing	Department	2014-08-22T00:50:30Z	DepartmentName: Marketing EmployeeCount: 153
Sales	00010	2014-08-22T00:50:44Z	FirstName: Ken LastName: Kwok Age: 23 Email: kenk@contoso.com

So far, this data appears 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. Also, 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.

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 to enable fast look-ups. However, the Table service does not create any secondary indexes, so **PartitionKey** and RowKey are the only indexed properties.

A table comprises one or more partitions, and many of the design decisions you make will be around choosing a suitable **PartitionKey** and **RowKey** to optimize your solution. A solution may consist of a single table that contains all your entities organized into partitions, but typically a solution has 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, and form the basis of how the table service scales.

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.

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 sometimes also referred to as batch transactions. EGTs can only operate on entities stored in the same partition (that is, share the same partition key in a given table). So anytime you require atomic transactional behavior across multiple entities, you must 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; otherwise, processing can be delayed.

EGTs also introduce a potential trade-off for you to evaluate in your design. That is, using more partitions increases the scalability of your application, because Azure has more opportunities for load balancing requests across nodes. But using more partitions 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.

Capacity considerations

The following table describes some of the key values to be aware of when you are designing a Table service solution:

	Capacity
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
Size of an individual entity	Up to 1 MB with a maximum of 255 properties (including the **PartitionKey**, **RowKey**, and **Times-tamp**)
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.

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 Table service solution. However, in many scenarios, storing denormalized or duplicate data in order to improve the performance or scalability of your solution is a valid approach.

Design for querying

Table service solutions may be read intensive, write intensive, or a mix of the two. This article 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.

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.

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

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.

- 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')`

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.

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.

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**.

Authorization in Azure Storage

Authorize with Shared Key

Every request made against a storage service must be authorized, unless the request is for a blob or container resource that has been made available for public or signed access. One option for authorizing a request is by using Shared Key.

Use the Shared Key authorization scheme to make requests against the Table service using the REST API. Shared Key authorization for the Table service in version 2009-09-19 and later uses the same signature string as in previous versions of the Table service.

An authorized request requires two headers: the `Date` or `x-ms-date` header and the `Authorization` header. The following sections describe how to construct these headers.

Specifying the Date Header

All authorized requests must include the Coordinated Universal Time (UTC) timestamp for the request. You can specify the timestamp either in the `x-ms-date` header, or in the standard HTTP/HTTPS `Date` header. If both headers are specified on the request, the value of `x-ms-date` is used as the request's time of creation.

The storage services ensure that a request is no older than 15 minutes by the time it reaches the service. This guards against certain security attacks, including replay attacks. When this check fails, the server returns response code 403 (Forbidden).

Note: The `x-ms-date` header is provided because some HTTP client libraries and proxies automatically set the `Date` header, and do not give the developer an opportunity to read its value in order to include it in the authorized request. If you set `x-ms-date`, construct the signature with an empty value for the `Date` header.

Specifying the Authorization Header

An authorized request must include the `Authorization` header. If this header is not included, the request is anonymous and may only succeed against a container or blob that is marked for public access, or against a container, blob, queue, or table for which a shared access signature has been provided for delegated access.

To authorize a request, you must sign the request with the key for the account that is making the request and pass that signature as part of the request.

The format for the `Authorization` header is as follows:

```
Authorization="[SharedKey|SharedKeyLite] <AccountName>:<Signature>"
```

Where `SharedKey` or `SharedKeyLite` is the name of the authorization scheme, `AccountName` is the name of the account requesting the resource, and `Signature` is a Hash-based Message Authentication Code (HMAC) constructed from the request and computed by using the SHA256 algorithm, and then encoded by using Base64 encoding.

Note: It is possible to request a resource that resides beneath a different account, if that resource is publicly accessible.

The following sections describe how to construct the `Authorization` header.

Constructing the Signature String

How you construct the signature string depends on which service and version you are authorizing against and which authorization scheme you are using. When constructing the signature string, keep in mind the following:

- The VERB portion of the string is the HTTP verb, such as GET or PUT, and must be uppercase.
- For Shared Key authorization for the Blob, Queue, and File services, each header included in the signature string may appear only once. If any header is duplicated, the service returns status code 400 (Bad Request).
- The values of all standard HTTP headers must be included in the string in the order shown in the signature format, without the header names. These headers may be empty if they are not being specified as part of the request; in that case, only the new-line character is required.
- If the `x-ms-date` header is specified, you may ignore the `Date` header, regardless of whether it is specified on the request, and simply specify an empty line for the `Date` portion of the signature string. It is acceptable to specify both `x-ms-date` and `Date`; in this case, the service uses the value of `x-ms-date`.
- If the `x-ms-date` header is not specified, specify the `Date` header in the signature string, without including the header name.
- All new-line characters (\n) shown are required within the signature string.
- The signature string includes canonicalized headers and canonicalized resource strings. Canonicalizing these strings puts them into a standard format that is recognized by Azure Storage. For detailed information on constructing the `CanonicalizedHeaders` and `CanonicalizedResource` strings that make up part of the signature string, see the appropriate sections later in this topic.

Table Service (Shared Key authorization)

The format of the signature string for Shared Key against the Table service is the same for all versions.

The Shared Key signature string for a request against the Table service differs slightly from that for a request against the Blob or Queue service, in that it does not include the `CanonicalizedHeaders` portion of the string. Additionally, the `Date` header in this case is never empty even if the request sets the `x-ms-date` header. If the request sets `x-ms-date`, that value is also used for the value of the `Date` header.

To encode the signature string for a request against the Table service made using the REST API, use the following format:

```
StringToSign = VERB + "\n" +
               Content-MD5 + "\n" +
               Content-Type + "\n" +
               Date + "\n" +
               CanonicalizedResource;
```

Note: Beginning with version 2009-09-19, the Table service requires that all REST calls include the `DataServiceVersion` and `MaxDataServiceVersion` headers.

Establishing a stored access policy

A stored access policy provides an additional level of control over service-level shared access signatures (SAS) on the server side. Establishing a stored access policy serves to group shared access signatures and to provide additional restrictions for signatures that are bound by the policy. You can use a stored access policy to change the start time, expiry time, or permissions for a signature, or to revoke it after it has been issued.

The following storage resources support stored access policies:

- Blob containers
- File shares
- Queues
- Tables

Note:
Note that a stored access policy on a container can be associated with a shared access signature granting permissions to the container itself or to the blobs it contains. Similarly, a stored access policy on a file share can be associated with a shared access signature granting permissions to the share itself or to the files it contains.

Stored access policies are currently not supported for account SAS.

Creating or Modifying a Stored Access Policy

To create or modify a stored access policy, call the Set ACL operation for the resource with a request body that specifies the terms of the access policy. The body of the request includes a unique signed identifier of your choosing, up to 64 characters in length, and the optional parameters of the access policy, as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<SignedIdentifiers>
  <SignedIdentifier>
    <Id>unique-64-char-value</Id>
    <AccessPolicy>
      <Start>start-time</Start>
      <Expiry>expiry-time</Expiry>
      <Permission>abbreviated-permission-list</Permission>
    </AccessPolicy>
  </SignedIdentifier>
</SignedIdentifiers>
```

Table entity range restrictions (`startpk`, `startrk`, `endpk`, and `endrk`) cannot be specified in a stored access policy.

A maximum of five access policies may be set on a container, table, or queue at any given time. Each `SignedIdentifier` field, with its unique `Id` field, corresponds to one access policy. Attempting to set more than five access policies at one time results in the service returning status code 400 (Bad Request).

Modifying or Revoking a Stored Access Policy

To modify the parameters of the stored access policy, you can call the access control list operation for the resource type to replace the existing policy, specifying a new start time, expiry time, or set of permissions. For example, if your existing policy grants read and write permissions to a resource, you can modify it to grant only read permissions for all future requests. In this case, the signed identifier of the new policy, as specified by the `ID` field, would be identical to the signed identifier of the policy you are replacing.

To revoke a stored access policy, you can either delete it, or rename it by changing the signed identifier. Changing the signed identifier breaks the associations between any existing signatures and the stored access policy. Deleting or renaming the stored access policy immediately effects all of the shared access signatures associated with it.

To remove a single access policy, call the resource's Set ACL operation, passing in the set of signed identifiers that you wish to maintain on the container. To remove all access policies from the resource, call the Set ACL operation with an empty request body.

Cross-Origin Resource Sharing (CORS) Support for the Azure Storage Services

Beginning with version 2013-08-15, the Azure storage services support Cross-Origin Resource Sharing (CORS) for the Blob, Table, and Queue services. The File service supports CORS beginning with version 2015-02-21.

CORS is an HTTP feature that enables a web application running under one domain to access resources in another domain. Web browsers implement a security restriction known as same-origin policy that prevents a web page from calling APIs in a different domain; CORS provides a secure way to allow one domain (the origin domain) to call APIs in another domain. See **the CORS specification**[2] for details on CORS.

You can set CORS rules individually for each of the storage services, by calling `Set Blob Service Properties`, `Set File Service Properties`, `Set Queue Service Properties`, and `Set Table Service Properties`. Once you set the CORS rules for the service, then a properly authenticated request made against the service from a different domain will be evaluated to determine whether it is allowed according to the rules you have specified.

Important: CORS is not an authentication mechanism. Any request made against a storage resource when CORS is enabled must either have a proper authentication signature, or must be made against a public resource. CORS is not supported for Premium Storage accounts.

2 http://www.w3.org/TR/cors/

Table service REST API

Table services resources

The Table service exposes the following resources via the REST API:

- **Account.** The storage account is a uniquely identified entity within the storage system. The storage account is the parent namespace for the Table service. All tables are associated with an account.
- **Tables.** The Tables resource represents the set of tables within a given storage account.
- **Entity.** An individual table stores data as a collection of entities.

Resource URI Syntax

The base URI for Table service resources is the storage account:

```
https://myaccount.table.core.windows.net
```

To list the tables in a given storage account, to create a new table, or to delete a table, refer to the set of tables in the specified storage account:

```
https://myaccount.table.core.windows.net/Tables
```

To return a single table, name that table within the Tables collection, as follows:

```
https://myaccount.table.core.windows.net/Tables('MyTable')
```

To query entities in a table, or to insert, update, or delete an entity, refer to that table directly within the storage account. This basic syntax refers to the set of all entities in the named table:

```
https://myaccount.table.core.windows.net/MyTable()
```

The format for addressing data resources for queries conforms to that specified by the OData Protocol Specification. You can use this syntax to filter entities based on criteria specified on the URI.

Note that all values for query parameters must be URL encoded before they are sent to the Azure storage services.

Supported HTTP Operations

Each resource supports operations based on the HTTP verbs GET, PUT, HEAD, and DELETE. The verb, syntax, and supported HTTP version(s) for each operation appears on the reference page for each operation. For a complete list of operation reference pages, see **Table Service REST API**[3].

Query timeout and pagination

The Table service supports the following two types of query operations:

- The `Query Tables` operation returns the list of tables within the specified storage account. The list of tables may be filtered according to criteria specified on the request.

3 https://docs.microsoft.com/en-us/rest/api/storageservices/table-service-rest-api

- The `Query Entities` operation returns a set of entities from the specified table. Query results may be filtered according to criteria specified on the request.
- A query against the Table service may return a maximum of 1,000 items at one time and may execute for a maximum of five seconds. If the result set contains more than 1,000 items, if the query did not complete within five seconds, or if the query crosses the partition boundary, the response includes headers which provide the developer with continuation tokens to use in order to resume the query at the next item in the result set. Continuation token headers may be returned for a Query Tables operation or a Query Entities operation.
- Note that the total time allotted to the request for scheduling and processing the query is 30 seconds, including the five seconds for query execution.
- It is possible for a query to return no results but to still return a continuation header.
- The continuation token headers are shown in the following table.

Continuation token header	Description
`x-ms-continuation-NextTableName`	This header is returned in the context of a `Query Tables` operation. If the list of tables returned is not complete, a hash of the name of the next table in the list is included in the continuation token header.
`x-ms-continuation-NextPartitionKey`	This header is returned in the context of a `Query Entities` operation. The header contains a hash of the next partition key to be returned in a subsequent query against the table.
`x-ms-continuation-NextRowKey`	This header is returned in the context of a `Query Entities` operation. The header contains a hash of the next row key to be returned in a subsequent query against the table. Note that in some instances, `x-ms-continuation-NextRowKey` may be `null`.

To retrieve the continuation tokens and execute a subsequent query to return the next page of results, first inspect the response headers for continuation tokens. If there are no headers or the header values are `null`, there are no additional entities to return.

Note: When making subsequent requests that include continuation tokens, be sure to pass the original URI on the request. For example, if you have specified a `$filter`, `$select`, or `$top` query option as part of the original request, you will want to include that option on subsequent requests. Otherwise your subsequent requests may return unexpected results. The `$top` query option in this case specifies the maximum number of results per page, not the maximum number of results in the whole response set.

If you are handling continuation tokens manually using the Microsoft .NET Client Library, first cast the result of the query operation to a `QueryOperationResponse` object. You can then access the continuation token headers in the `Headers` property of the `QueryOperationResponse` object.

After you have retrieved the continuation tokens, use their values to construct a query to return the next page of results. A query request URI may take these parameters, which correspond to the continuation token headers returned with the response:

- `NextTableName`
- `NextPartitionKey`
- `NextRowKey`

The total time allotted to the request for scheduling and processing a query is 30 seconds, including five seconds for query execution.

If the operation is an insert, update, or delete operation, the operation may have succeeded on the server despite an error being returned by the client. This may happen when the client timeout is set to less than 30 seconds, which is the maximum timeout for an insert, update, or delete operation.

Sample Response Headers and Subsequent Request

The following code example shows a set of sample response headers from an entity query against a table named Customers that returns continuation headers. Both `x-ms-continuation-NextPartitionKey` and `x-ms-continuation-NextRowKey` are returned.

```
Date: Mon, 27 Jun 2016 20:11:08 GMT
Content-Type: application/json;charset=utf-8
Server: Windows-Azure-Table/1.0 Microsoft-HTTPAPI/2.0
Cache-Control: no-cache
x-ms-request-id: f9b2cd09-4dec-4570-b06d-4fa30179a58e
x-ms-version: 2015-12-11
x-ms-continuation-NextPartitionKey: 1!8!U21pdGg-
x-ms-continuation-NextRowKey: 1!8!QmVuOTk5
Content-Length: 880298
```

The request for the next page of data can be constructed like the following URI:

```
http://myaccount.table.core.windows.net/Customers?NextPartition-
Key=1!8!U21pdGg-&NextRowKey=1!12!QmVuMTg5OA--
```

Querying Tables and Entities

Querying tables and entities in the Table service requires careful construction of the request URI. The following sections describe query options and demonstrate some common scenarios.

Basic Query Syntax

To return all of the tables in a given storage account, perform a `GET` operation on the Tables resource, as described in the `Query Tables` operation.

- The basic URI for addressing the Tables: `https://myaccount.table.core.windows.net/Tables`
- To return a single named table: `https://myaccount.table.core.windows.net/Tables('MyTable')`
- To return all entities in a table, specify the table name on the URI without the Tables resource: `https://myaccount.table.core.windows.net/MyTable()`

Query results are sorted by `PartitionKey`, then by `RowKey`. Ordering results in any other way is not currently supported.

You can specify additional options to limit the set of tables or entities returned, as described below.

Supported Query Options

The Table service supports the following query options, which conform to the OData Protocol Specification. You can use these options to limit the set of tables, entities, or entity properties returned by a query.

System query option	Description
`$filter`	Returns only tables or entities that satisfy the specified filter. Note that no more than 15 discrete comparisons are permitted within a `$filter` string.
`$top`	Returns only the top `n` tables or entities from the set.
`$select`	Returns the desired properties of an entity from the set. This query option is only supported for requests using version 2011-08-18 or newer.
	Additional query options defined by OData are not supported by the Table service.

Note: When making subsequent requests that include continuation tokens, be sure to pass the original URI on the request. For example, if you have specified a `$filter`, `$select`, or `$top` query option as part of the original request, you will want to include that option on subsequent requests. Otherwise your subsequent requests may return unexpected results. The $top query option in the case where results are paginated specifies the maximum number of results per page, not the maximum number of results in the whole response set.

Supported Comparison Operators

Within a `$filter` clause, you can use comparison operators to specify the criteria against which to filter the query results.

For all property types, the following comparison operators are supported:

Operator	URI expression
`Equal`	`eq`
`GreaterThan`	`gt`
`GreaterThanOrEqual`	`ge`
`LessThan`	`lt`
`LessThanOrEqual`	`le`
`NotEqual`	`ne`

Additionally, the following operators are supported for Boolean properties:

Operator	URI expression
`And`	`and`
`Not`	`not`
`Or`	`or`

Query String Encoding

The following characters must be encoded if they are to be used in a query string:

- Forward slash (/)

- Question mark (?)
- Colon (:)
- 'At' symbol (@)
- Ampersand (&)
- Equals sign (=)
- Plus sign (+)
- Comma (,)
- Dollar sign ($)

Single quotes in query strings must be represented as two consecutive single quotes (`''`). For example, "o'clock" would be: `o''clock`.

Sample Query Expressions

The following samples show how to construct the request URI for some typical entity queries using REST syntax.

Note that both the `$top` and `$filter` options can be used to filter on table names as well, using the syntax demonstrated for filtering on properties of type `String`.

Returning the Top n Entities

To return the top `n` entities for any query, specify the `$top` query option. The following example returns the top 10 entities from a table named Customers:

```
https://myaccount.table.core.windows.net/Customers()?$top=10
```

Filtering on the PartitionKey and RowKey Properties

Because the `PartitionKey` and `RowKey` properties form an entity's primary key, you can use a special syntax to identify the entity, as follows:

```
https://myaccount.table.core.windows.net/Customers(PartitionKey='MyParti-
tion',RowKey='MyRowKey1')
```

Alternatively, you can specify these properties as part of the `$filter` option, as shown in the following section.

Note that the key property names and constant values are case-sensitive. Both the `PartitionKey` and `RowKey` properties are of type `String`.

Constructing Filter Strings

When constructing a filter string, keep these rules in mind:

- Use the logical operators defined by the OData Protocol Specification to compare a property to a value. Note that it is not possible to compare a property to a dynamic value; one side of the expression must be a constant.

- The property name, operator, and constant value must be separated by URL-encoded spaces. A space is URL-encoded as `%20`.
- All parts of the filter string are case-sensitive.
- The constant value must be of the same data type as the property in order for the filter to return valid results.

Note: Be sure to check whether a property has been explicitly typed before assuming it is of a type other than string. If a property has been explicitly typed, the type is indicated within the response when the entity is returned. If the property has not been explicitly typed, it will be of type `String`, and the type will not be indicated within the response when the entity is returned.

Filtering on String Properties

When filtering on string properties, enclose the string constant in single quotes.

The following example filters on the `PartitionKey` and `RowKey` properties; additional non-key properties could also be added to the query string.

```
https://myaccount.table.core.windows.net/Customers()?$filter=PartitionKey%20
eq%20'MyPartitionKey'%20and%20RowKey%20eq%20'MyRowKey1'
```

The following example filters on a `FirstName` and `LastName` property:

```
https://myaccount.table.core.windows.net/Customers()?$filter=LastName%20
eq%20'Smith'%20and%20FirstName%20eq%20'John'
```

Note that the Table service does not support wildcard queries. However, you can perform prefix matching by using comparison operators on the desired prefix. The following example returns entities with a `LastName` property beginning with the letter 'A':

```
https://myaccount.table.core.windows.net/Customers()?$filter=LastName%20
ge%20'A'%20and%20LastName%20lt%20'B'
```

Filtering on Numeric Properties

To filter on an integer or floating-point number, specify the constant value on the URI without quotes.

This example returns all entities with an `Age` property whose value is greater than 30:

```
https://myaccount.table.core.windows.net/Customers()?$filter=Age%20gt%2030
```

Filtering on Boolean Properties

To filter on a Boolean value, specify `true` or `false` without quotes. The following example returns all entities where the IsActive property is set to `true`:

```
https://myaccount.table.core.windows.net/Customers()?$filter=IsActive%20
eq%20true
```

Filtering on DateTime Properties

To filter on a `DateTime` value, specify the `datetime` keyword on the URI, followed by the date/time constant in single quotes. The date/time constant must be in combined UTC format.

The following example returns entities where the `CustomerSince` property is equal to July 10, 2008:

```
https://myaccount.table.core.windows.net/Customers()?$filter=Custom-
erSince%20eq%20datetime'2008-07-10T00:00:00Z'
```

Filtering on GUID Properties

To filter on a GUID value, specify the `guid` keyword on the URI, followed by the guid constant in single quotes.

The following example returns entities where the `GuidValue` property is equal to :

```
https://myaccount.table.core.windows.net/Customers()?$filter=GuidValue%20
eq%20guid'a455c695-df98-5678-aaaa-81d3367e5a34'
```

Inserting and updating entities

To insert or update an entity, you include with the request an an OData ATOM or OData JSON entity that specifies the properties and data for the entity.

The `Insert Entity` operation inserts a new entity with a unique primary key, formed from the combination of the PartitionKey and the RowKey. The `Update Entity` operation replaces an existing entity with the same `PartitionKey` and `RowKey`. The `Merge Entity` operation updates the properties of an existing entity, but does not replace the entity. The `Insert Or Merge Entity` operation creates a new entity with a unique primary key or updates the properties of an existing entity, but does not replace the entity. The `Insert Or Replace Entity` operation creates a new entity with a unique primary key or replaces an existing entity.

Constructing the Atom Feed

The Atom feed for an insert or update operation defines the entity's properties by specifying their names and data types, and sets the values for those properties.

The `content` element contains the entity's property definitions, which are specified within the `m:properties` element. The property's type is specified by the `m:type` attribute.

Here is an example of an Atom feed for an Insert Entity operation:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://www.w3.org/2005/Atom">
  <title />
  <author>
    <name />
  </author>
  <id />
  <content type="application/xml">
    <m:properties>
```

```
      <d:Address>Mountain View</d:Address>
      <d:Age m:type="Edm.Int32">23</d:Age>
      <d:AmountDue m:type="Edm.Double">200.23</d:AmountDue>
      <d:BinaryData m:type="Edm.Binary" m:null="true" />
      <d:CustomerCode m:type="Edm.Guid">c9da6455-213d-42c9-9a79-
3e9149a57833</d:CustomerCode>
      <d:CustomerSince m:type="Edm.
DateTime">2008-07-10T00:00:00</d:CustomerSince>
      <d:IsActive m:type="Edm.Boolean">true</d:IsActive>
      <d:NumOfOrders m:type="Edm.Int64">255</d:NumOfOrders>
      <d:PartitionKey>mypartitionkey</d:PartitionKey>
      <d:RowKey>myrowkey1</d:RowKey>
    </m:properties>
  </content>
</entry>
```

Note: Atom payloads are supported only in versions prior to 2015-12-11. Beginning with version 2015-12-11, payloads must be in JSON.

Constructing the JSON Feed

To insert or update an entity using the OData JSON format, create a JSON object with property names as keys together with their property values. You may need to include the property type if it cannot be inferred through OData JSON type detection heuristics.

The JSON payload corresponding to the ATOM example above is as follows:

```
{
   "Address":"Mountain View",
   "Age":23,
   "AmountDue":200.23,
   "CustomerCode@odata.type":"Edm.Guid",
   "CustomerCode":"c9da6455-213d-42c9-9a79-3e9149a57833",
   "CustomerSince@odata.type":"Edm.DateTime",
   "CustomerSince":"2008-07-10T00:00:00",
   "IsActive":true,
   "NumOfOrders@odata.type":"Edm.Int64",
   "NumOfOrders":"255",
   "PartitionKey":"mypartitionkey",
   "RowKey":"myrowkey"
}
```

Review Questions

Module 1 review questions

Storage tables REST API

Storage tables offer a set of transactional functionality that mirrors the traditional Create, Read, Update, and Delete methods found in many other data sources. Can you name the basic operations that can be performed on entities by using the representational state transfer (REST) pattern.

> Click to see suggested answer

```
Base URL https://[account].table.core.windows.net/[table]
```

Method	Endpoint
GET	https://[account].table.core.windows.net/table
PUT	https://[account].table.core.windows.net/table
POST	https://[account].table.core.windows.net/[table]
DELETE	https://[account].table.core.windows.net/table
MERGE	https://[account].table.core.windows.net/table

Connecting to Azure Storage

Every request made against Microsoft Azure Storage must be authorized, unless the request is for a binary large object (blob) or container resource that has been made available for public or signed access. What are some of the options for authorizing a request?

> Click to see suggested answer

One option for authorizing a request is by using a shared key with the REST API. Another method of authorizing access to a storage account is by using a connection string. A connection string includes the authentication information required for your application to access data in an Azure storage account at run time. You can configure connection strings to:

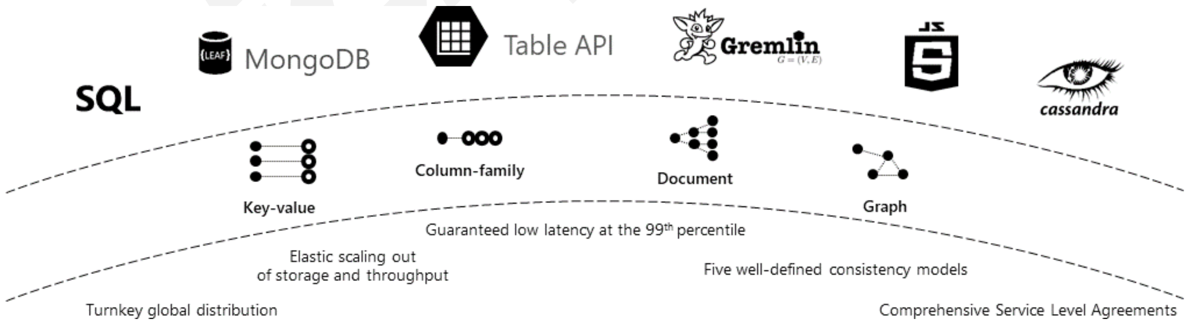
- Connect to the Azure Storage Emulator.
- Access an Azure storage account.
- Access specified resources in Azure via a Shared Access Signature (SAS).

Module 2 Develop solutions that use Azure Cosmos DB storage

Azure Cosmos DB overview

Azure Cosmos DB

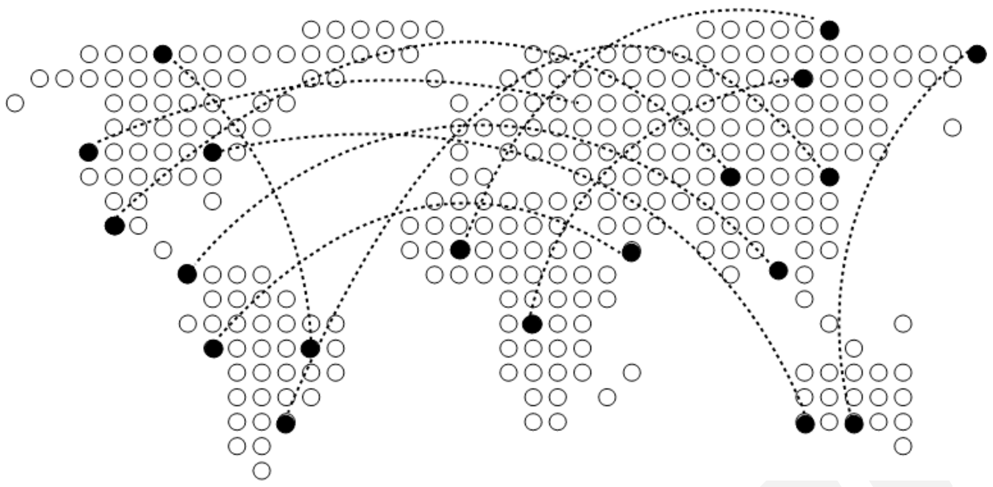
Microsoft Azure Cosmos DB is a database service native to Azure that focuses on providing a high-performance database regardless of your selected API or data model. Azure Cosmos DB offers multiple APIs and models that can be used interchangeably for various application scenarios.


SQL
MongoDB
Table API
Gremlin
cassandra
Key-value
Column-family
Document
Graph
Guaranteed low latency at the 99th percentile
Elastic scaling out of storage and throughput
Five well-defined consistency models
Turnkey global distribution
Comprehensive Service Level Agreements

Core functionality

Global replication

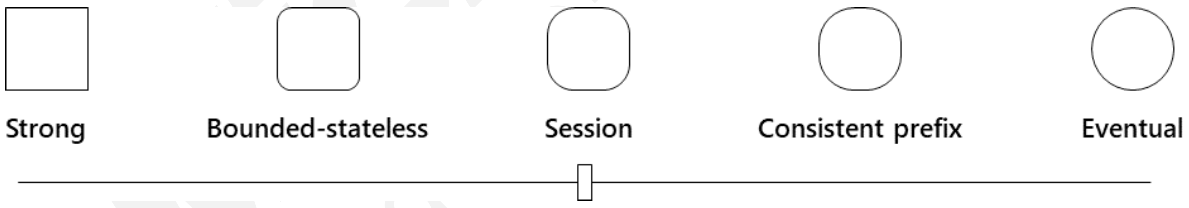
Azure Cosmos DB has a feature referred to as *turnkey global distribution* that automatically replicates data to other Azure datacenters across the globe without the need to manually write code or build a replication infrastructure.



Consistency levels

Commercial distributed databases fall into two categories: databases that do not offer well-defined, provable consistency choices at all and databases that offer two extreme programmability choices (strong versus eventual consistency). The former burdens application developers with the minutia of their replication protocols and expects them to make difficult tradeoffs among consistency, availability, latency, and throughput. The latter pressure them to choose one of the two extremes.

Azure Cosmos DB provides five consistency levels: strong, bounded-staleness, session, consistent prefix, and eventual. Bounded-staleness, session, consistent prefix, and eventual are referred to as *relaxed consistency models*, because they provide less consistency than strong, which is the most highly consistent model available.



The consistency levels range from very strong consistency—where reads are guaranteed to be visible across replicas before a write is fully committed across all replicas—to eventual consistency, where writes are readable immediately, and replicas are eventually consistent with the primary.

Consistency Level	Description
Strong	When a write operation is performed on your primary database, the write operation is replicated to the replica instances. The write operation is committed (and visible) on the primary only after it has been committed and confirmed by all replicas.

Consistency Level	Description
Bounded Stateless	This level is similar to the Strong level with the major difference that you can configure how stale documents can be within replicas. Staleness refers to the quantity of time (or the version count) a replica document can be behind the primary document.
Session	This level guarantees that all read and write operations are consistent within a user session. Within the user session, all reads and writes are monotonic and guaranteed to be consistent across primary and replica instances.
Consistent Prefix	This level has loose consistency but guarantees that when updates show up in replicas, they will show up in the correct order (that is, as prefixes of other updates) without any gaps.
Eventual	This level has the loosest consistency and essentially commits any write operation against the primary immediately. Replica transactions are asynchronously handled and will eventually (over time) be consistent with the primary. This tier has the best performance, because the primary database does not need to wait for replicas to commit to finalize it's transactions.

Choose the right consistency level for your application

Distributed databases relying on replication for high availability, low latency or both, make the fundamental tradeoff between the read consistency vs. availability, latency, and throughput. Most commercially available distributed databases ask developers to choose between the two extreme consistency models: strong consistency and eventual consistency. Azure Cosmos DB allows developers to choose among the five well-defined consistency models: strong, bounded staleness, session, consistent prefix, and eventual. Each of these consistency models is well-defined, intuitive and can be used for specific real-world scenarios. Each of the five consistency models provide availability and performance tradeoffs and are backed by comprehensive SLAs. The following simple considerations will help you make the right choice in many common scenarios.

SQL API and Table API

Consider the following points if your application is built by using Cosmos DB SQL API or Table API

- For many real-world scenarios, session consistency is optimal and it's the recommended option.
- If your application requires strong consistency, it is recommended that you use bounded staleness consistency level.
- If you need stricter consistency guarantees than the ones provided by session consistency and single-digit-millisecond latency for writes, it is recommended that you use bounded staleness consistency level.

- If your application requires eventual consistency, it is recommended that you use consistent prefix consistency level.
- If you need less strict consistency guarantees than the ones provided by session consistency, it is recommended that you use consistent prefix consistency level.
- If you need the highest availability and lowest latency, then use eventual consistency level.

Consistency guarantees in practice

You may get stronger consistency guarantees in practice. Consistency guarantees for a read operation correspond to the freshness and ordering of the database state that you request. Read-consistency is tied to the ordering and propagation of the write/update operations.

- When the consistency level is set to **bounded staleness**, Cosmos DB guarantees that the clients always read the value of a previous write, with a lag bounded by the staleness window.
- When the consistency level is set to **strong**, the staleness window is equivalent to zero, and the clients are guaranteed to read the latest committed value of the write operation.
- For the remaining three consistency levels, the staleness window is largely dependent on your workload. For example, if there are no write operations on the database, a read operation with **eventual**, **session**, or **consistent** prefix consistency levels is likely to yield the same results as a read operation with strong consistency level.

If your Cosmos DB account is configured with a consistency level other than the strong consistency, you can find out the probability that your clients may get strong and consistent reads for your workloads by looking at the Probabilistic Bounded Staleness (PBS) metric. This metric is exposed in the Azure portal.

Probabilistic bounded staleness shows how eventual is your eventual consistency. This metric provides an insight into how often you can get a stronger consistency than the consistency level that you have currently configured on your Cosmos DB account. In other words, you can see the probability (measured in milliseconds) of getting strongly consistent reads for a combination of write and read regions.

Consistency levels and Azure Cosmos DB APIs

Five consistency models offered by Azure Cosmos DB are natively supported by the Azure Cosmos DB SQL API. When you use Azure Cosmos DB, the SQL API is the default.

Azure Cosmos DB also provides native support for wire protocol-compatible APIs for popular databases. Databases include MongoDB, Apache Cassandra, Gremlin, and Azure Table storage. These databases don't offer precisely defined consistency models or SLA-backed guarantees for consistency levels. They typically provide only a subset of the five consistency models offered by Azure Cosmos DB. For the SQL API, Gremlin API, and Table API, the default consistency level configured on the Azure Cosmos DB account is used.

The following sections show the mapping between the data consistency requested by an OSS client driver for Apache Cassandra 4.x and MongoDB 3.4. This document also shows the corresponding Azure Cosmos DB consistency levels for Apache Cassandra and MongoDB.

Mapping between Apache Cassandra and Azure Cosmos DB consistency levels

This table shows the "read consistency" mapping between the Apache Cassandra 4.x client and the default consistency level in Azure Cosmos DB. The table shows multi-region and single-region deployments.

Apache Cassandra 4.x	Azure Cosmos DB (multi-region)	Azure Cosmos DB (single region)
ONE, TWO, THREE	Consistent prefix	Consistent prefix
LOCAL_ONE	Consistent prefix	Consistent prefix
QUORUM, ALL, SERIAL	Bounded staleness is the default. Strong is in private preview.	Strong
LOCAL_QUORUM	Bounded staleness	Strong
LOCAL_SERIAL	Bounded staleness	Strong

Mapping between MongoDB 3.4 and Azure Cosmos DB consistency levels

The following table shows the "read concerns" mapping between MongoDB 3.4 and the default consistency level in Azure Cosmos DB. The table shows multi-region and single-region deployments.

MongoDB 3.4	Azure Cosmos DB (multi-region)	Azure Cosmos DB (single region)
Linearizable	Strong	Strong
Majority	Bounded staleness	Strong
Local	Consistent prefix	Consistent prefix

Azure Cosmos DB supported APIs

Today, Azure Cosmos DB can be accessed by using five different APIs. The underlying data structure in Azure Cosmos DB is a data model based on atom record sequences that enabled Azure Cosmos DB to support multiple data models. Because of the flexible nature of atom record sequences, Azure Cosmos DB will be able to support many more models and APIs over time.

MongoDB API

The MongoDB API in Azure Cosmos DB acts as a massively scalable MongoDB service powered by the Azure Cosmos DB platform. It is compatible with existing MongoDB libraries, drivers, tools, and applications.

Table API

The Table API in Azure Cosmos DB is a key-value database service built to provide premium capabilities (for example, automatic indexing, guaranteed low latency, and global distribution) to existing Azure Table storage applications without making any app changes.

Gremlin API

The Gremlin API in Azure Cosmos DB is a fully managed, horizontally scalable graph database service that makes it easy to build and run applications that work with highly connected datasets supporting Open Graph APIs (based on the Apache TinkerPop specification, Apache Gremlin).

Apache Cassandra API

The Cassandra API in Azure Cosmos DB is a globally distributed Apache Cassandra service powered by the Azure Cosmos DB platform. Compatible with existing Apache Cassandra libraries, drivers, tools, and applications.

SQL API

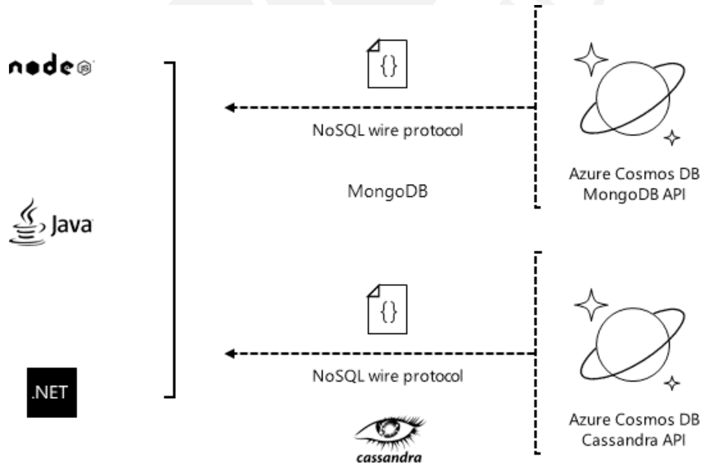
The SQL API in Azure Cosmos DB is a JavaScript and JavaScript Object Notation (JSON) native API based on the Azure Cosmos DB database engine. The SQL API also provides query capabilities rooted in the familiar SQL query language. Using SQL, you can query for documents based on their identifiers or make deeper queries based on properties of the document, complex objects, or even the existence of specific properties. The SQL API supports the execution of JavaScript logic within the database in the form of stored procedures, triggers, and user-defined functions.

Migrating from NoSQL

Many NoSQL database engines are simple to get started with, but they provide problems as you scale, including:

- Tedious set-up and maintenance requirements for a multiple-server database cluster
- Expensive and complex high-availability solutions
- Challenges in achieving end-to-end security, including encryption at rest and in flight
- Required resource overprovisioning and unpredictable costs to achieve scale

Azure Cosmos DB has a MongoDB API and a Cassandra API to provide a NoSQL service offering for two of the most popular NoSQL database platforms. Both APIs are protocol compatible with the Cassandra API supporting CQLv4 and the MongoDB API supporting MongoDB v5. Many applications can be "lifted and shifted" to Azure Cosmos DB without the need to rewrite code.



To achieve a successful migration, it is important to keep a few tips in mind:

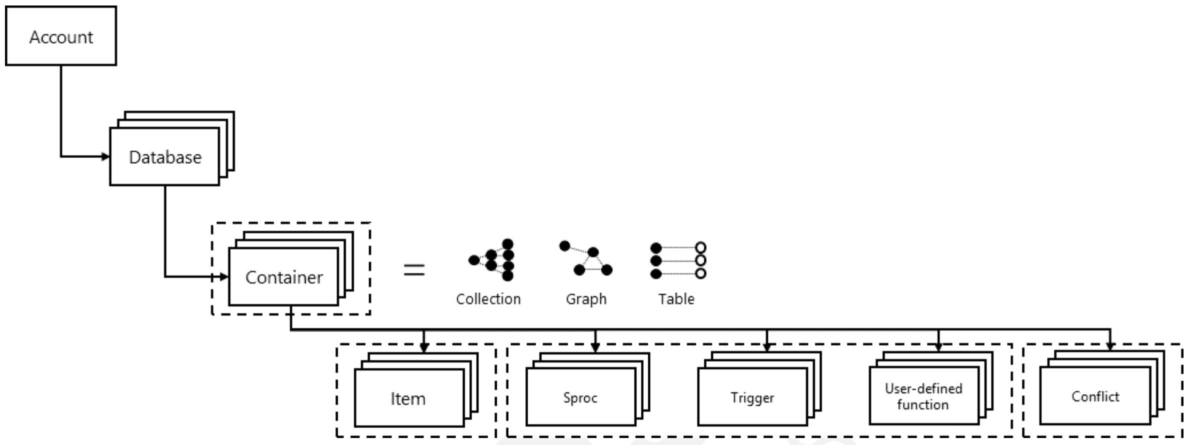
- Instead of writing custom code, you should use native tools, such as the Cassandra shell, mongodump, and mongoexport.
- Azure Cosmos DB containers should be allocated prior to the migration with the appropriate throughput levels set. Many of the tools will create containers for you with default settings that are not ideal.

- Prior to migrating, you should increase the container's throughput to at least 1,000 Request Units (RUs) per second so that the import tools are not throttled. The throughput can be reverted back to the typical values after the import is complete.

Managing containers and items

Resource hierarchy

The JSON documents stored in the Azure Cosmos DB SQL API are managed through a well-defined hierarchy of database resources. The Azure Cosmos DB hierarchical resource model consists of sets of resources under a database account, each addressable via a logical and stable URI. A set of resources is referred to as a feed.



Resource	Description
Account	A database account is associated with a set of databases and a fixed amount of large object (blob) storage for attachments. You can create one or more database accounts by using your Azure subscription. For more information, visit the pricing page.
Database	A database is a logical container of document storage partitioned across collections. It is also a users container.
Collection (container)	A collection is a container of JSON documents and the associated JavaScript application logic. Collections can span one or more partitions or servers and can scale to handle practically unlimit-ed volumes of storage or throughput.
Document (item)	User-defined (arbitrary) JSON content. By default, no schema needs to be defined nor do secondary indexes need to be provided for all the documents added to a collection.
Stored procedure (sproc)	Application logic written in JavaScript that is registered with a collection and executed within the database engine as a transaction.
Trigger	Application logic written in JavaScript executed before or after either an insert, replace, or delete operation.

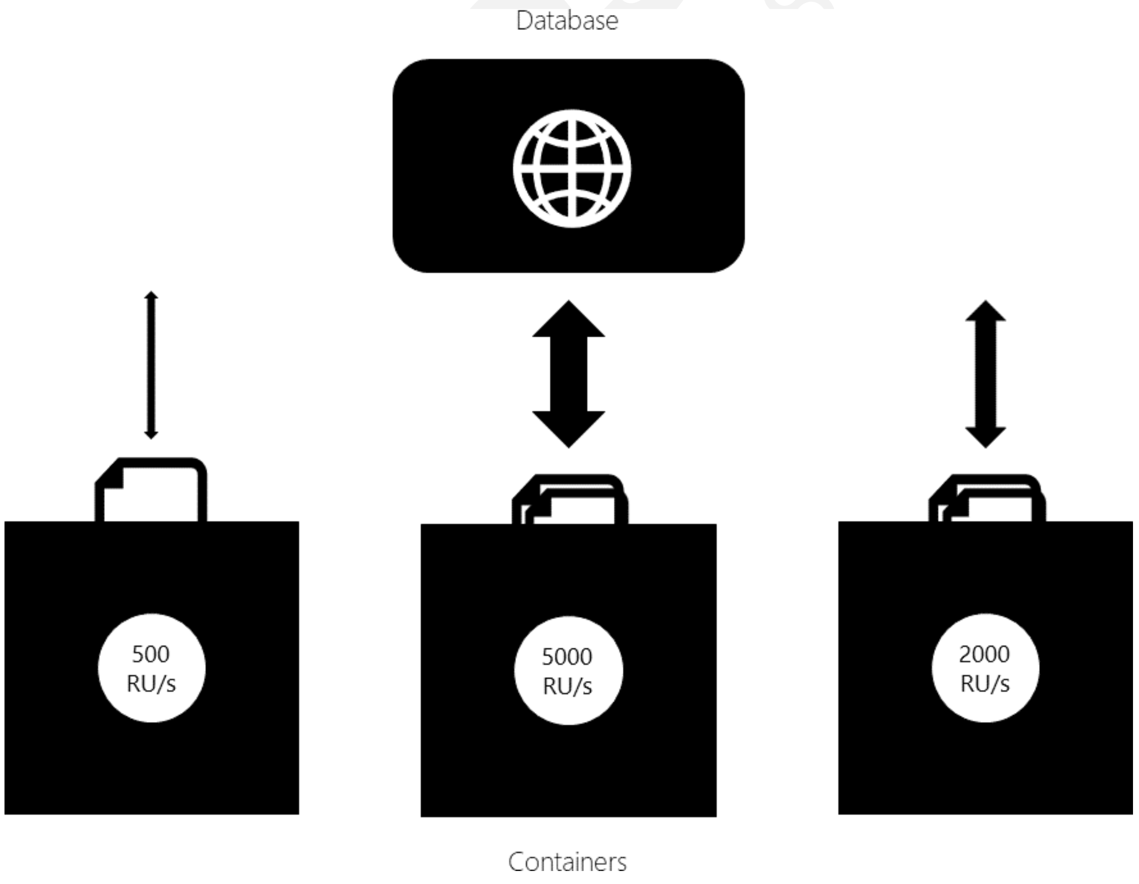
Resource	Description
User-defined function	Application logic written in JavaScript. User-defined functions enable you to model a custom query operator and thereby extend the core SQL API query language.

Collections

In the Azure Cosmos DB SQL API, databases are essentially containers for collections. Collections are where you place individual documents. A collection is intrinsically elastic—it automatically grows and shrinks as you add or remove documents.

Each collection is assigned a throughput value, and that value dictates the maximum throughput for that collection and its corresponding documents. Alternatively, you can assign the throughput at the database level and share the throughput values among the collections in the database. If you have a set of documents that needs throughput beyond the limits of an individual collection, you can distribute the documents among multiple collections. Each collection has its own distinct throughput level.

If a particular collection is seeing spikes in throughput, you can manage its throughput level in isolation by increasing or decreasing the value. This change to the throughput level of a particular collection will not cause side effects for the other collections. This allows you to adjust to meet the performance needs of any workload in isolation.



You can also scale workloads across collections, if you have a workload that needs to be partitioned, you can scale that workload by distributing its associated documents across multiple collections. The SQL API for Azure Cosmos DB includes a client-side partition resolver that allows you to manage transactions and point them in code to the correct partition based on a partition key field.

Collection types

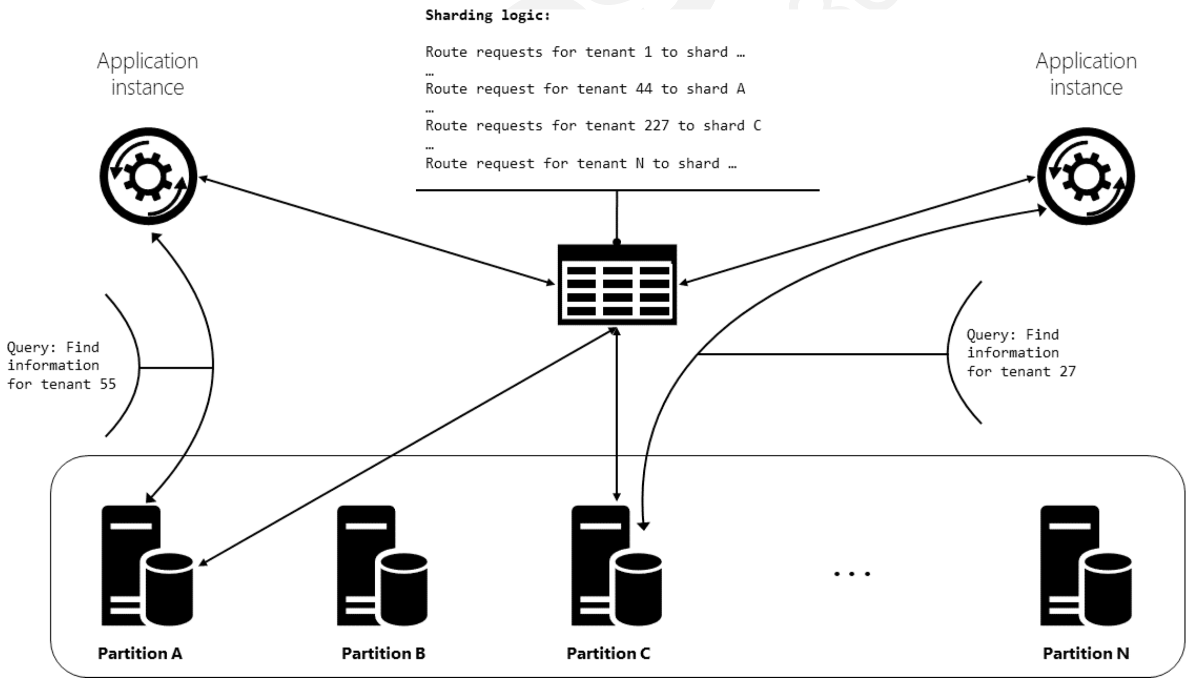
Azure Cosmos DB containers can be created as fixed or unlimited in the Azure portal. Fixed-size containers have a maximum limit of 10 GB and a 10,000 RU/s throughput. To create a container as unlimited, you must specify a partition key and a minimum throughput of 1,000 RU/s. Azure Cosmos DB containers can also be configured to share throughput among the containers in a database.

If you created a fixed container with no partition key or a throughput less than 1,000 RU/s, the container will not automatically scale. To migrate the data from a fixed container to an unlimited container, you need to use the data migration tool or the Change Feed library.

Partitioning

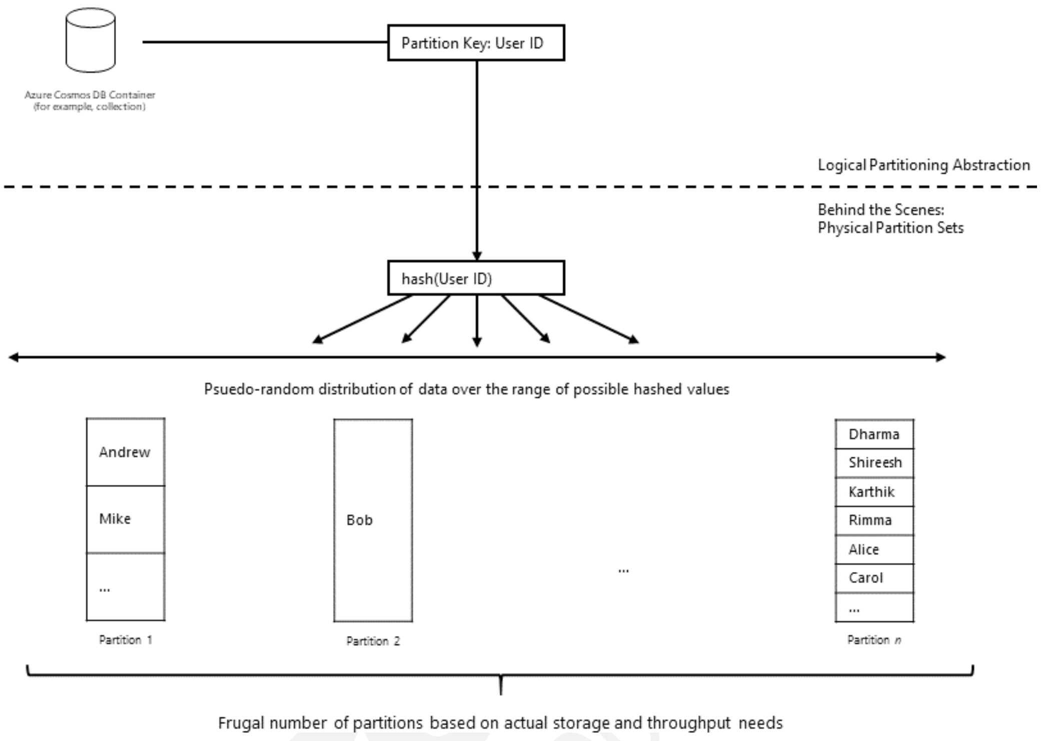
Azure Cosmos DB provides containers for storing data called collections (for documents), graphs, or tables. *Containers* are logical resources and can span one or more physical partitions or servers. The number of partitions is determined by Azure Cosmos DB based on the storage size and throughput provisioned for a container or set of containers.

If you are already familiar with the sharding pattern, the idea of dynamic partitioning is not very different.


Sharding logic:
Route requests for tenant 1 to shard …
…
Route request for tenant 44 to shard A
…
Route requests for tenant 227 to shard C
…
Route request for tenant N to shard …
Application instance
Application instance
Query: Find information for tenant 55
Query: Find information for tenant 27
Partition A
Partition B
Partition C
. . .
Partition N

A *physical partition* is a fixed amount of reserved solid-state drive (SSD) backend storage combined with a variable amount of compute resources (CPU and memory). Each physical partition is replicated for high availability. A physical partition is an internal concept of Azure Cosmos DB, and physical partitions are transient. Azure Cosmos DB will automatically scale the number of physical partitions based on your workload.

A *logical partition* is a partition within a physical partition that stores all the data associated with a single partition key value. Partition ranges can be dynamically subdivided to seamlessly grow the database as the application grows while simultaneously maintaining high availability. When a container meets the partitioning prerequisites, partitioning is completely transparent to your application. Azure Cosmos DB handles distributing data across physical and logical partitions and routing query requests to the right partition.


Azure Cosmos DB Container
(for example, collection)
Partition Key: User ID
Logical Partitioning Abstraction
Behind the Scenes:
Physical Partition Sets
hash(User ID)
Psuedo-random distribution of data over the range of possible hashed values
Andrew
Mike
...
Partition 1
Bob
Partition 2
...
Dharma
Shireesh
Karthik
Rimma
Alice
Carol
...
Partition n
Frugal number of partitions based on actual storage and throughput needs

Create and update documents by using code

Manage collections and documents by using the Microsoft .NET SDK

To get started with the Azure Cosmos DB SQL API, you will need the **Microsoft.Azure.DocumentDB.Core**[1] package from NuGet.

First, you will need to add the following `using` directives to the top of your class file:

```
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
```

Then, you can create a `DocumentClient` instance by using the endpoint from your Azure Cosmos DB account and one of your keys:

```
DocumentClient client = new DocumentClient(new Uri("[endpoint]"), "[key]");
```

To reference any resource in the software development kit (SDK), you will need a URI. The `UriFactory` class contains a series of static helper methods that can create URIs for common Azure Cosmos DB resources. In this example, we will create a URI for a collection:

```
Uri collectionUri = UriFactory.CreateDocumentCollectionUri(databaseName, 
collectionName);
```

Now, you can use the `CreateDocumentAsync` method of the `DocumentClient` class to insert a C# object into the collection. You can use any C# type you want for your documents, because the SDK doesn't require a specific base type:

```
var document = new {
firstName = "Alex",
lastName = "Leh"
}

await this.client.CreateDocumentAsync(collectionUri, document);
```

If you want to query the database, you can perform SQL queries by using the `SqlQuerySpec` class:

```
var query = client.CreateDocumentQuery<Family>(
    collectionUri,
    new SqlQuerySpec()
    {
          QueryText = "SELECT * FROM f WHERE (f.surname = @lastName)",
          Parameters = new SqlParameterCollection()
          {
                new SqlParameter("@lastName", "Andt")
          }
    },
    DefaultOptions
);
```

1 https://www.nuget.org/packages/Microsoft.Azure.DocumentDB.Core

```
var families = query.ToList();
```

Alternatively, you can use the language-integrated query (LINQ) feature of C# with the SDK. The LINQ expressions will be automatically translated into the appropriate SQL query:

```
var query = client.CreateDocumentQuery<Family>(collectionUri)
    .Where(d => d.Surname = "Andt")
    .Select(d => new { Name = d.Id, City = d.Address?.City)
    .AsDocumentQuery();

var families = query.ToList();
```

Stored Procedures

Review Questions

Module 2 review questions

Azure Cosmos DB Core functionality

Azure Cosmos DB has a feature referred to as turnkey global distribution that automatically replicates data to other Azure datacenters across the globe without the need to manually write code or build a replication infrastructure. Can you name, and describe, the different consistency levels?

> Click to see suggested answer

Consistency Level	Description
Strong	When a write operation is performed on your primary database, the write operation is replicated to the replica instances. The write operation is committed (and visible) on the primary only after it has been committed and confirmed by all replicas.
Bounded Stateless	This level is similar to the Strong level with the major difference that you can configure how stale documents can be within replicas. Staleness refers to the quantity of time (or the version count) a replica document can be behind the primary document.
Session	This level guarantees that all read and write operations are consistent within a user session. Within the user session, all reads and writes are monotonic and guaranteed to be consistent across primary and replica instances.
Consistent Prefix	This level has loose consistency but guarantees that when updates show up in replicas, they will show up in the correct order (that is, as prefixes of other updates) without any gaps.
Eventual	This level has the loosest consistency and essentially commits any write operation against the primary immediately. Replica transactions are asynchronously handled and will eventually (over time) be consistent with the primary. This tier has the best performance, because the primary database does not need to wait for replicas to commit to finalize it's transactions.

Partitioning

Azure Cosmos DB provides containers for storing data called collections (for documents), graphs, or tables. Containers are logical resources and can span one or more physical partitions or servers. Can you describe the differences between physical and logical partitions?

> Click to see suggested answer

A *physical partition* is a fixed amount of reserved solid-state drive (SSD) backend storage combined with a variable amount of compute resources (CPU and memory). Each physical partition is replicated for high availability.

A *logical partition* is a partition within a physical partition that stores all the data associated with a single partition key value. Partition ranges can be dynamically subdivided to seamlessly grow the database as the application grows while simultaneously maintaining high availability.

Module 3 Develop solutions that use a relational database

Azure SQL overview

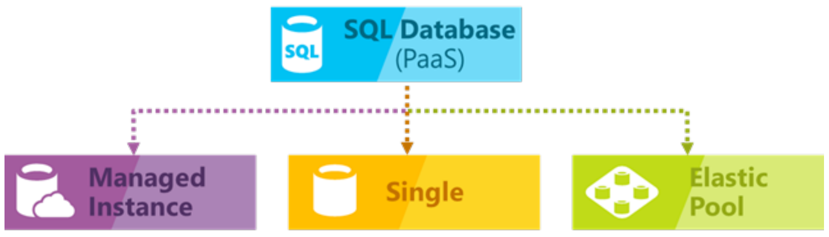
The Azure SQL Database service

SQL Database is a general-purpose relational database managed service in Microsoft Azure that supports structures such as relational data, JSON, spatial, and XML. SQL Database delivers dynamically scalable performance within two different purchasing models: a vCore-based purchasing model and a DTU-based purchasing model. SQL Database also provides options such as columnstore indexes for extreme analytic analysis and reporting, and in-memory OLTP for extreme transactional processing. Microsoft handles all patching and updating of the SQL code base seamlessly and abstracts away all management of the underlying infrastructure.

Azure SQL Database provides the following deployment options for an Azure SQL database:

- As a single database with its own set of resources managed via a logical server
- As a pooled database in an elastic pool with a shared set of resources managed via a logical server
- As a part of a collection of databases known as a managed instance that contains system and user databases and sharing a set of resources

The following illustration shows these deployment options:



SQL Database shares its code base with the Microsoft SQL Server database engine. With Microsoft's cloud-first strategy, the newest capabilities of SQL Server are released first to SQL Database, and then to

SQL Server itself. This approach provides you with the newest SQL Server capabilities with no overhead for patching or upgrading - and with these new features tested across millions of databases.

Choosing the right SQL Server option in Azure

In Azure, you can have your SQL Server workloads running in a hosted infrastructure (IaaS) or running as a hosted service (PaaS). The key question that you need to ask when deciding between PaaS or IaaS is do you want to manage your database, apply patches, take backups, or you want to delegate these operations to Azure? Depending on the answer, you have the following options:

- **Azure SQL Database:** A fully-managed SQL database engine, based on the latest stable Enterprise Edition of SQL Server. This is a relational database-as-a-service (DBaaS) hosted in the Azure cloud that falls into the industry category of Platform-as-a-Service (PaaS). SQL database is built on standardized hardware and software that is owned, hosted, and maintained by Microsoft. With SQL Database, you can use built-in features and functionality that require extensive configuration in SQL Server. When using SQL Database, you pay-as-you-go with options to scale up or out for greater power with no interruption. SQL Database has additional features that are not available in SQL Server, such as built-in intelligence and management. Azure SQL Database offers several deployment options:
    - You can deploy a single database to a logical server. A logical server containing single and pooled databases offers most of database-scoped features of SQL Server. This option is optimized for modern application development of new cloud-born applications.
    - You can deploy to a Azure SQL Database Managed Instances. With Azure SQL Database Managed Instance, Azure SQL Database offers shared resources for databases and additional instance-scoped features. Azure SQL Database Managed Instance supports database migration from on-premises with minimal to no database change. This option provides all of the PaaS benefits of Azure SQL Database but adds capabilities that were previously only available in SQL VMs. This includes a native virtual network (VNet) and near 100% compatibility with on-premises SQL Server.
- **SQL Server on Azure Virtual Machines** falls into the industry category Infrastructure-as-a-Service (IaaS) and allows you to run SQL Server inside a fully-managed virtual machine in the Azure cloud. SQL Server virtual machines also run on standardized hardware that is owned, hosted, and maintained by Microsoft. When using SQL Server on a VM, you can either pay-as you-go for a SQL Server license already included in a SQL Server image or easily use an existing license. You can also stop or resume the VM as needed.SQL Server installed and hosted in the cloud on Windows Server or Linux virtual machines (VMs) running on Azure, also known as an infrastructure as a service (IaaS). SQL Server on Azure virtual machines is a good option for migrating on-premises SQL Server databases and applications without any database change. All recent versions and editions of SQL Server are available for installation in an IaaS virtual machine. The most significant difference from SQL Database is that SQL Server VMs allow full control over the database engine. You can choose when maintenance/patching will start, to change the recovery model to simple or bulk logged to enable faster load less log, to pause or start engine when needed, and you can fully customize the SQL Server database engine. With this additional control comes with added responsibility to manage the virtual machines.

The main differences between these options are listed in the following table:

SQL Server on VM	Azure SQL Database (Managed Instance)	Azure SQL Database (Logical server)
You have full control over the SQL Server engine. Up to 99.95% availability. Full parity with the matching version of on-premises SQL Server. Fixed, well-known database engine version. Easy migration from SQL Server on-premises. Private IP address within Azure VNet. You have ability to deploy application or services on the host where SQL Server is placed.	High compatibility with SQL Server on-premises. 99.99% availability guaranteed. Built-in backups, patching, recovery. Latest stable Database Engine version. Easy migration from SQL Server. Private IP address within Azure VNet. Built-in advanced intelligence and security. Online change of resources (CPU/storage).	The most commonly used SQL Server features are available. 99.99% availability guaranteed. Built-in backups, patching, recovery. Latest stable Database Engine version. Ability to assign necessary resources (CPU/storage) to individual databases. Built-in advanced intelligence and security. Online change of resources (CPU/storage).
You need to manage your backups and patches. You need to implement your own High-Availability solution. There is a downtime while changing the resources(CPU/storage)	There is still some minimal number of SQL Server features that are not available. No guaranteed exact maintenance time (but nearly transparent). Compatibility with the SQL Server version can be achieved only using database compatibility levels.	Migration from SQL Server might be hard. Some SQL Server features are not available. No guaranteed exact maintenance time (but nearly transparent). Compatibility with the SQL Server version can be achieved only using database compatibility levels. Private IP address cannot be assigned (you can limit the access using firewall rules).

Copy a transactionally consistent copy of an Azure SQL database

Azure SQL Database provides several methods for creating a transactionally consistent copy of an existing Azure SQL database on either the same server or a different server. You can copy a SQL database by using the Azure portal, PowerShell, or T-SQL.

Overview

A database copy is a snapshot of the source database as of the time of the copy request. You can select the same server or a different server, its service tier and compute size, or a different compute size within the same service tier (edition). After the copy is complete, it becomes a fully functional, independent database. At this point, you can upgrade or downgrade it to any edition. The logins, users, and permissions can be managed independently.

Note: Automated database backups are used when you create a database copy.

Logins in the database copy

When you copy a database to the same logical server, the same logins can be used on both databases. The security principal you use to copy the database becomes the database owner on the new database. All database users, their permissions, and their security identifiers (SIDs) are copied to the database copy.

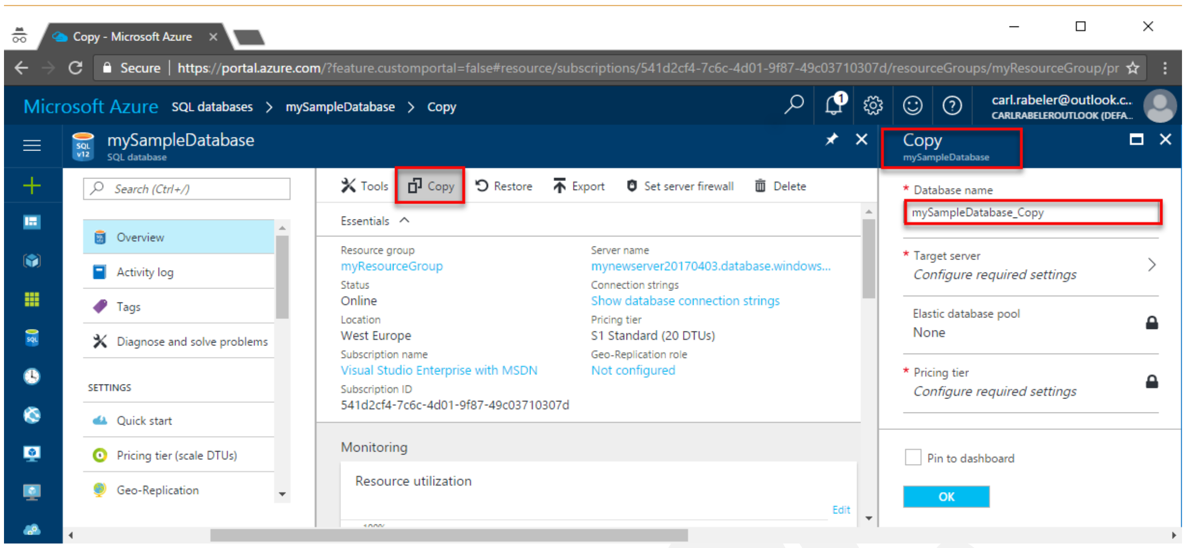
When you copy a database to a different logical server, the security principal on the new server becomes the database owner on the new database. If you use contained database users for data access, ensure that both the primary and secondary databases always have the same user credentials, so that after the copy is complete you can immediately access it with the same credentials.

If you use Azure Active Directory, you can completely eliminate the need for managing credentials in the copy. However, when you copy the database to a new server, the login-based access might not work, because the logins do not exist on the new server. To learn about managing logins when you copy a database to a different logical server, see How to manage Azure SQL database security after disaster recovery.

After the copying succeeds and before other users are remapped, only the login that initiated the copying, the database owner, can log in to the new database.

Copy a database by using the Azure portal

To copy a database by using the Azure portal, open the page for your database, and then click **Copy**.


Copy - Microsoft Azure
Secure | https://portal.azure.com/?feature.customportal=false#resource/subscriptions/541d2cf4-7c6c-4d01-9f87-49c03710307d/resourceGroups/myResourceGroup/pr
Microsoft Azure SQL databases > mySampleDatabase > Copy
carl.rabeler@outlook.c... CARLRABELEROUTLOOK (DEFA...
mySampleDatabase
SQL database
Search (Ctrl+/)
Overview
Activity log
Tags
Diagnose and solve problems
SETTINGS
Quick start
Pricing tier (scale DTUs)
Geo-Replication
Tools
Copy
Restore
Export
Set server firewall
Delete
Essentials
Monitoring
Resource utilization
Edit
Copy
mySampleDatabase
Pin to dashboard
OK

Resource group
myResourceGroup
Status
Online
Location
West Europe
Subscription name
Visual Studio Enterprise with MSDN
Subscription ID
541d2cf4-7c6c-4d01-9f87-49c03710307d
Server name
mynewserver20170403.database.windows...
Connection strings
Show database connection strings
Pricing tier
S1 Standard (20 DTUs)
Geo-Replication role
Not configured

* Database name
mySampleDatabase_Copy
* Target server
Configure required settings
Elastic database pool
None
* Pricing tier
Configure required settings

Copy a database by using PowerShell

To copy a database by using PowerShell, use the `New-AzureRmSqlDatabaseCopy` cmdlet.

```
New-AzureRmSqlDatabaseCopy -ResourceGroupName "myResourceGroup" `
    -ServerName $sourceserver `
    -DatabaseName "MySampleDatabase" `
    -CopyResourceGroupName "myResourceGroup" `
    -CopyServerName $targetserver `
    -CopyDatabaseName "CopyOfMySampleDatabase"
```

Resolve logins

After the new database is online on the destination server, use the `ALTER USER` statement to remap the users from the new database to logins on the destination server. To resolve orphaned users, see Troubleshoot Orphaned Users.

All users in the new database retain the permissions that they had in the source database. The user who initiated the database copy becomes the database owner of the new database and is assigned a new security identifier (SID). After the copying succeeds and before other users are remapped, only the login that initiated the copying, the database owner, can log in to the new database.

Create, read, update, and delete database tables by using code

Entity Framework

Entity Framework is an object-relational mapper library for Microsoft .NET that is designed to reduce the impedance mismatch between the relational and object-oriented worlds. The goal of the library is to enable developers to interact with data stored in relational databases by using strongly-typed .NET objects that represent the application's domain and to eliminate the need for a large portion of the data access "plumbing" code that they usually need to write to access data in a database.

Entity Framework Core and Entity Framework

Entity Framework Core (EF Core) is a recent rewrite of the entire Entity Framework library to target .NET Standard. Entity Framework Core can be used with .NET Framework applications and .NET Core applications. Entity Framework Core was built to be more lightweight and agile than the full Entity Framework by dropping many of the earlier features from Entity Framework and implementing new, modern, and extensible features at an agile pace. For new applications, we recommend considering using Entity Framework Core over Entity Framework.

Note: The examples in this section will assume that you are using Entity Framework Core.

Entity Framework providers

The Entity Framework provider model allows Entity Framework to be used with different types of database servers. For example, one provider can be plugged in to allow Entity Framework to be used against Microsoft SQL Server, whereas another provider can be plugged in to allow Entity Framework to be used against Oracle Database. There are many current providers in the market for databases, including:

- SQL Server
- SQLite
- PostgreSQL
- MySQL
- MariaDB
- MyCAT Server
- SQL Server Compact
- Firebird
- DB2
- Informix
- Oracle
- Microsoft Access

Note: If a provider you need is not available, you can certainly write a provider yourself, although it should not be considered a trivial undertaking.

Entity Framework Core also ships with an InMemory provider. This database provider allows Entity Framework Core to be used with an in-memory database. The InMemory provider is useful when you want to test components by using something that approximates connecting to the real database without the overhead of actual database operations.

Note: EF Core database providers do not have to be relational databases. InMemory is designed to be a general-purpose database for testing and is not designed to mimic a relational database.

SQL Server provider

This database provider allows Entity Framework Core to be used with Microsoft SQL Server (including Microsoft Azure SQL Database). The provider is maintained as an open-source project as part of the Entity Framework Core repository on GitHub (**https://github.com/aspnet/EntityFrameworkCore**).

Many of the examples you will find online assume that you are using the SQL Server provider with Entity Framework Core. It is important to remember that Entity Framework Core has a provider model that abstracts the underlying database away from the actual database access logic. The code samples you see can be used with many of the database providers.

MySQL and PostgreSQL providers

The MySQL team maintains a database provider for both Entity Framework and Entity Framework Core as part of the MySQL Connector for .NET library. Along with the MySQL team, other third-party groups have written providers for MySQL. Two of the MySQL providers are:

- MySql.Data.EntityFrameworkCore (**https://www.nuget.org/packages/MySql.Data.EntityFrameworkCore**).
- Pomelo.EntityFrameworkCore.MySql (**https://www.nuget.org/packages/Pomelo.EntityFrameworkCore.MySql/**).

There are multiple third-party organizations that have written .NET libraries to access PostgreSQL. Many of them have rewritten their Entity Framework providers to support Entity Framework Core. These libraries include:

- Npgsql.EntityFrameworkCore.PostgreSQL (**https://www.nuget.org/packages/Npgsql.EntityFrameworkCore.PostgreSQL/**).
- Devart.Data.PostgreSql.EFCore (**https://www.nuget.org/packages/Devart.Data.PostgreSql.EFCore/**).

These providers allow you to use Entity Framework Core with a MySQL or PostgreSQL database in the same manner as you would use the library with a SQL database.

Modeling a database by using Entity Framework Core

To use Entity Framework to query, insert, update, and delete data using .NET objects, you first need to create a model that maps the entities and relationships defined in your model to tables in a database.

Entity Framework uses a set of conventions to build a model based on the shapes of your entity classes. You can specify additional configuration to supplement and override what was discovered by convention. The conventions can be applied to a model targeting any data store and when targeting any relational database. Providers might also enable a configuration that is specific to a particular data store.

First, let's look at how we can model a database with a single table named **Blogs**.

BlogId	Url	Description
1	/first-post	This is my first post on this platform
2	/follow-up-post	NULL

If we want to use plain-old CLR objects (POCOs), such as existing domain objects, to model this table, we would have a class that looks like this:

```
public class Blog
{
    public int BlogId { get; set; }

    public string Url { get; set; }

    public string Description { get; set; }
}
```

Logically, our database has a table that is a collection of these blog instances. Without knowing anything about Entity Framework, we would probably create a class that looks like this:

```
public class BlogDatabase
{
    public IEnumerable<Blog> Blogs { get; set; }
}
```

Now, we need to find a way to mark these classes in C# as models of our database. Including a type in the model means that Entity Framework has metadata about that type and will attempt to read and write instances from and to the database. There are two methods for modeling a database: the fluent API or data annotations.

Fluent API

You can override the `OnModelCreating` method in your derived context class and use the ModelBuilder API to configure your model. This is the most powerful method of configuration and allows the configuration to be specified without modifying your entity classes. The fluent API configuration has the highest precedence and will override conventions and data annotations:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
           .HasKey(c => c.BlogId)
           .Property(b => b.Url)
           .IsRequired()
           .Property(b => b.Description);
}
```

Data annotations

You can apply attributes (known as data annotations) to your classes and properties. data annotations will override conventions but will be overwritten by a fluent API configuration (if it exists):

```
public class Blog
{
    [Key]
    public int BlogId { get; set; }

    [Required]
    public string Url { get; set; }

    public string Description { get; set; }
}
```

DbContext implementation

After you have a model, the primary class your application interacts with is `System.Data.Entity.DbContext` (often referred to as the *context class*). You can use a `DbContext` class associated to a model to:

- Write and execute queries.
- Materialize query results as entity objects.
- Track changes that are made to those objects.
- Persist object changes back on the database.
- Bind objects in memory to UI controls.

The recommended way to work with the context is to define a class that derives from `DbContext` and exposes `DbSet` properties that represent collections of the specified entities in the context:

```
public class BlogContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
}
```

By convention, types that are exposed in `DbSet` properties on your context are included in your model. In addition, types that are mentioned in the `OnModelCreating` method are also included. Finally, any types that are found by recursively exploring the navigation properties of discovered types are also included in the model.

Querying databases by using Entity Framework Core

The `DbSet<>` generic class includes methods that will allow you to query your database by using language-integrated query (LINQ).

If you are already familiar with the LINQ syntax, you can perform many queries in Entity Framework Core without the need to learn too much. This is because the `DbSet<>` generic class implements the `IEnumerable<>` interface, giving you access to many of the existing LINQ queries.

For example, you can load all the data from a table by enumerating the collection with a call to the `ToList` method:

```
List<Blog> allblogs = context.Blogs.ToList();
```

You can use LINQ methods such as the `Where` method to filter your resulting list:

```
IEnumerable<Blog> someblogs = context.Blogs
    .Where(b => b.Url.Contains("dotnet"))
```

You can also use the `Single` method to get a single instance that matches a specific filter:

```
Blog specificblog = context.Blogs
    .Single(b => b.BlogId == 1);
```

When you call LINQ operators, you are simply building up an in-memory representation of the query. The query is sent to the database only when the results are consumed (enumerated). The most common operations that result in the query being sent to the database are:

- Iterating the results in a **for** loop.
- Using an operator such as *ToList*, *ToArray*, *Single*, or *Count*.
- Data binding the results of a query to a UI.

Although Entity Framework does help protect against SQL injection attacks, it does not do any general validation of input. Therefore, if values being passed to APIs, used in LINQ queries, assigned to entity properties, and so on come from an untrusted source, the appropriate validation per your application requirements should be performed. This includes any user input used to dynamically construct queries. Even when using LINQ, if you are accepting user input to build expressions, you need to make sure that only intended expressions can be constructed.

Review Questions

Module 3 review questions

Azure SQL database

SQL Database is a general-purpose relational database managed service in Microsoft Azure that supports structures such as relational data, JSON, spatial, and XML. What are the three deployment options for Azure SQL database?

> Click to see suggested answer

Azure SQL Database provides the following deployment options for an Azure SQL database:

- As a single database with its own set of resources managed via a logical server
- As a pooled database in an elastic pool with a shared set of resources managed via a logical server
- As a part of a collection of databases known as a managed instance that contains system and user databases and sharing a set of resources

DbContext implementation

To use Entity Framework to query, insert, update, and delete data using .NET objects, you first need to create a model that maps the entities and relationships defined in your model to tables in a database. After you have a model, the primary class your application interacts with is `System.Data.Entity.DbContext` (often referred to as the context class). You can use a `DbContext` class associated to a model to (name as many as you can):

> Click to see suggested answer

- Write and execute queries.
- Materialize query results as entity objects.
- Track changes that are made to those objects.
- Persist object changes back on the database.
- Bind objects in memory to UI controls.

The recommended way to work with the context is to define a class that derives from `DbContext` and exposes `DbSet` properties that represent collections of the specified entities in the context.

Module 4 Develop solutions that use Microsoft Azure Blob storage

Azure Blob storage overview

Introduction to Azure Blob storage

Azure Blob storage is Microsoft's object storage solution for the cloud. Blob storage is optimized for storing massive amounts of unstructured data. Unstructured data is data that does not adhere to a particular data model or definition, such as text or binary data.

About Blob storage

Blob storage is designed for:

- 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.

Users or client applications can access objects in Blob storage via HTTP/HTTPS, from anywhere in the world. Objects in Blob storage are accessible via the Azure Storage REST API, Azure PowerShell, Azure CLI, or an Azure Storage client library. Client libraries are available for a variety of languages, including .NET, Java, Node.js, Python, Go, PHP, and Ruby.

About Azure Data Lake Storage Gen2

Blob storage supports Azure Data Lake Storage Gen2, Microsoft's enterprise big data analytics solution for the cloud. Azure Data Lake Storage Gen2 offers a hierarchical file system as well as the advantages of

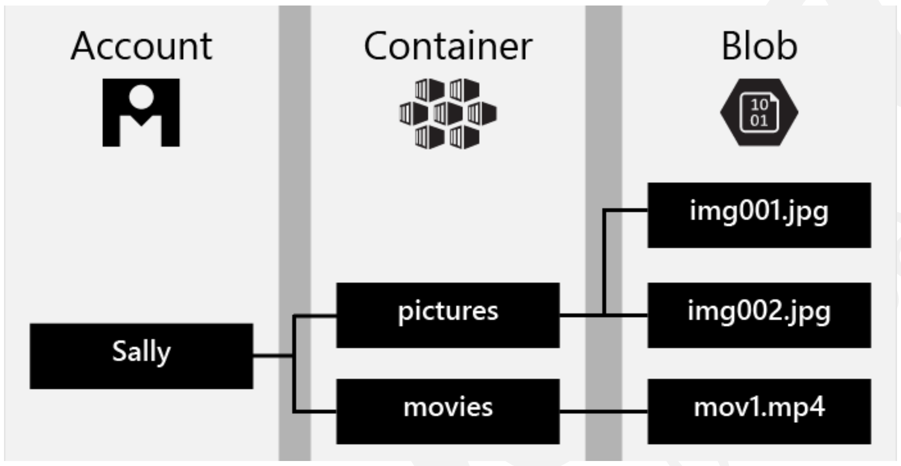
Blob storage, including low-cost, tiered storage; high availability; strong consistency; and disaster recovery capabilities.

Blob storage resources

Blob storage offers three types of resources:

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

The following diagram shows the relationship between these resources.



Storage accounts

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

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

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

To learn more about storage accounts, see **Azure storage account overview**[1].

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.

Note: The container name must be lowercase.

1 https://docs.microsoft.com/en-us/azure/storage/common/storage-account-overview?toc=%2fazure%2fstorage%2fblobs%2ftoc.json

Blobs

Azure Storage supports three types of blobs:

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

Move data to Blob storage

A number of solutions exist for migrating existing data to Blob storage:

- **AzCopy** is an easy-to-use command-line tool for Windows and Linux that copies data to and from Blob storage, across containers, or across storage accounts.
- The **Azure Storage Data Movement library** is a .NET library for moving data between Azure Storage services. The AzCopy utility is built with the Data Movement library.
- **Azure Data Factory** supports copying data to and from Blob storage by using the account key, shared access signature, service principal, or managed identities for Azure resources authentications.
- **Blobfuse** is a virtual file system driver for Azure Blob storage. You can use blobfuse to access your existing block blob data in your Storage account through the Linux file system.
- **Azure Data Box Disk** is a service for transferring on-premises data to Blob storage when large datasets or network constraints make uploading data over the wire unrealistic. You can use Azure Data Box Disk to request solid-state disks (SSDs) from Microsoft. You can then copy your data to those disks and ship them back to Microsoft to be uploaded into Blob storage.
- The **Azure Import/Export service** provides a way to export large amounts of data from your storage account to hard drives that you provide and that Microsoft then ships back to you with your data.

Azure Blob storage tiers

Overview

Azure storage offers different storage tiers which allow you to store Blob object data in the most cost-effective manner. The available tiers include:

- **Hot storage**: is optimized for storing data that is accessed frequently.
- **Cool storage** is optimized for storing data that is infrequently accessed and stored for at least 30 days.
- **Archive storage** is optimized for storing data that is rarely accessed and stored for at least 180 days with flexible latency requirements (on the order of hours).

The following considerations accompany the different storage tiers:

- The Archive storage tier is only available at the blob level and not at the storage account level.
- 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 compared to Hot data are acceptable trade-offs for lower storage costs.

- Archive storage is offline and offers the lowest storage costs but also the highest access costs.
- Only the Hot and Cool storage tiers can be set at the account level. Currently the Archive tier cannot be set at the account level.
- Hot, Cool, and Archive tiers can be set at the object level.

Data stored in the cloud grows 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 to optimize costs. 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 benefits from a different storage tier that is optimized for a particular access pattern. With Hot, Cool, and Archive storage tiers, Azure Blob storage addresses this need for differentiated storage tiers with separate pricing models.

Storage accounts that support tiering

You may only tier your object storage data to Hot, Cool, or Archive in Blob storage or General Purpose v2 (GPv2) accounts. General Purpose v1 (GPv1) accounts do not support tiering. However, customers can easily convert their existing GPv1 or Blob storage accounts to GPv2 accounts through a simple one-click process in the Azure portal. GPv2 provides a new pricing structure for blobs, files, and queues, and access to a variety of other new storage features as well. Furthermore, going forward some new features and prices cuts will only be offered in GPv2 accounts. Therefore, customers should evaluate using GPv2 accounts but only use them after reviewing the pricing for all services as some workloads can be more expensive on GPv2 than GPv1.

Blob storage and GPv2 accounts expose the **Access Tier** attribute at the account level, which allows you to specify the default storage tier as Hot or Cool for any blob in the storage account that does not have an explicit tier set at the object level. For objects with the tier set at the object level, the account tier will not apply. The Archive tier can only be applied at the object level. You can switch between these storage tiers at any time.

Hot access tier

Hot storage has higher storage costs than Cool and Archive storage, but the lowest access costs. 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.

Cool access tier

Cool storage tier has lower storage costs and higher access costs compared to Hot storage. This tier is intended for data that will remain in the Cool tier for at least 30 days. Example usage scenarios for the Cool storage tier include:

- Short-term backup 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. (*For example*, long-term storage of scientific data, raw telemetry data from a manufacturing facility)

Archive access tier

Archive storage has the lowest storage cost and higher data retrieval costs compared to Hot and Cool storage. This tier is intended for data that can tolerate several hours of retrieval latency and will remain in the Archive tier for at least 180 days.

While a blob is in Archive storage, it is offline and cannot be read (except the metadata, which is online and available), copied, overwritten, or modified. Nor can you take snapshots of a blob in Archive storage. However, you may use existing operations to delete, list, get blob properties/metadata, or change the tier of your blob.

Example usage scenarios for the Archive storage 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. (For *example*, 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. (*For example*, Security camera footage, old X-Rays/MRIs for healthcare organizations, audio recordings, and transcripts of customer calls for financial services)

Blob rehydration

To read data in Archive storage, you must first change the tier of the blob to Hot or Cool. This process is known as rehydration and can take up to 15 hours to complete. Large blob sizes are recommended for optimal performance. Rehydrating several small blobs concurrently may add additional time.

During rehydration, you may check the **Archive Status** blob property to confirm if the tier has changed. The status reads "rehydrate-pending-to-hot" or "rehydrate-pending-to-cool" depending on the destination tier. Upon completion, the Archive status property is removed, and the **Access Tier** blob property reflects the new Hot or Cool tier.

Comparison of the storage tiers

The following table shows a comparison of the Hot, Cool, and Archive storage tiers.

	Hot storage tier	**Cool storage tier**	**Archive storage tier**
Availability	99.9%	99%	N/A
Availability			
(RA-GRS reads)	99.99%	99.9%	N/A
Usage charges	Higher storage costs, lower access and transaction costs	Lower storage costs, higher access and transaction costs	Lowest storage costs, highest access and transaction costs
Minimum object size	N/A	N/A	N/A
Minimum storage duration	N/A	30 days (GPv2 only)	180 days

	Hot storage tier	**Cool storage tier**	**Archive storage tier**
Latency			
(Time to first byte)	milliseconds	milliseconds	< 15 hrs
Scalability and performance targets	Same as general-purpose storage accounts	Same as general-purpose storage accounts	Same as general-purpose storage accounts

Understanding Block Blobs, Append Blobs, and Page Blobs

he storage service offers three types of blobs, *block blobs*, *append blobs*, and *page blobs*. You specify the blob type when you create the blob. Once the blob has been created, its type cannot be changed, and it can be updated only by using operations appropriate for that blob type, i.e., writing a block or list of blocks to a block blob, appending blocks to a append blob, and writing pages to a page blob.

All blobs reflect committed changes immediately. Each version of the blob has a unique tag, called an *ETag*, that you can use with access conditions to assure you only change a specific instance of the blob.

Any blob can be leased for exclusive write access. When a blob is leased, only calls that include the current lease ID can modify the blob or (for block blobs) its blocks. Any blob can be duplicated in a snapshot.

Note: Blobs in the *Azure storage emulator* are limited to a maximum size of 2 GB.

About Block Blobs

Block blobs let you upload large blobs efficiently. Block blobs are comprised of blocks, each of which is identified by a block ID. You create or modify a block blob by writing a set of blocks and committing them by their block IDs. Each block can be a different size, up to a maximum of 100 MB (4 MB for requests using REST versions before 2016-05-31), and a block blob can include up to 50,000 blocks. The maximum size of a block blob is therefore slightly more than 4.75 TB (100 MB X 50,000 blocks). For REST versions before 2016-05-31, the maximum size of a block blob is a little more than 195 GB (4 MB X 50,000 blocks). If you are writing a block blob that is no more than 256 MB (64 MB for requests using REST versions before 2016-05-31) in size, you can upload it in its entirety with a single write operation.

Storage clients default to a 128 MB maximum single blob upload, settable using the `SingleBlobUploadThresholdInBytes` property of the `BlobRequestOptions` object. When a block blob upload is larger than the value in this property, storage clients break the file into blocks. You can set the number of threads used to upload the blocks in parallel on a per-request basis using the `ParallelOperationThreadCount` property of the `BlobRequestOptions` object.

When you upload a block to a blob in your storage account, it is associated with the specified block blob, but it does not become part of the blob until you commit a list of blocks that includes the new block's ID. New blocks remain in an uncommitted state until they are specifically committed or discarded. Writing a block does not update the last modified time of an existing blob.

Block blobs include features that help you manage large files over networks. With a block blob, you can upload multiple blocks in parallel to decrease upload time. Each block can include an MD5 hash to verify the transfer, so you can track upload progress and re-send blocks as needed. You can upload blocks in any order, and determine their sequence in the final block list commitment step. You can also upload a new block to replace an existing uncommitted block of the same block ID. You have one week to commit blocks to a blob before they are discarded. All uncommitted blocks are also discarded when a block list commitment operation occurs but does not include them.

You can modify an existing block blob by inserting, replacing, or deleting existing blocks. After uploading the block or blocks that have changed, you can commit a new version of the blob by committing the new blocks with the existing blocks you want to keep using a single commit operation. To insert the same range of bytes in two different locations of the committed blob, you can commit the same block in two places within the same commit operation. For any commit operation, if any block is not found, the entire commitment operation fails with an error, and the blob is not modified. Any block commitment overwrites the blob's existing properties and metadata, and discards all uncommitted blocks.

Block IDs are strings of equal length within a blob. Block client code usually uses base-64 encoding to normalize strings into equal lengths. When using base-64 encoding, the pre-encoded string must be 64 bytes or less. Block ID values can be duplicated in different blobs. A blob can have up to 100,000 uncommitted blocks, but their total size cannot exceed 200,000 MB.

If you write a block for a blob that does not exist, a new block blob is created, with a length of zero bytes. This blob will appear in blob lists that include uncommitted blobs. If you don't commit any block to this blob, it and its uncommitted blocks will be discarded one week after the last successful block upload. All uncommitted blocks are also discarded when a new blob of the same name is created using a single step (rather than the two-step block upload-then-commit process).

About Page Blobs

Page blobs are a collection of 512-byte pages optimized for random read and write operations. To create a page blob, you initialize the page blob and specify the maximum size the page blob will grow. To add or update the contents of a page blob, you write a page or pages by specifying an offset and a range that align to 512-byte page boundaries. A write to a page blob can overwrite just one page, some pages, or up to 4 MB of the page blob. Writes to page blobs happen in-place and are immediately committed to the blob. The maximum size for a page blob is 8 TB.

About Append Blobs

An append blob is comprised of blocks and is optimized for append operations. When you modify an append blob, blocks are added to the end of the blob only, via the `Append Block` operation. Updating or deleting of existing blocks is not supported. Unlike a block blob, an append blob does not expose its block IDs.

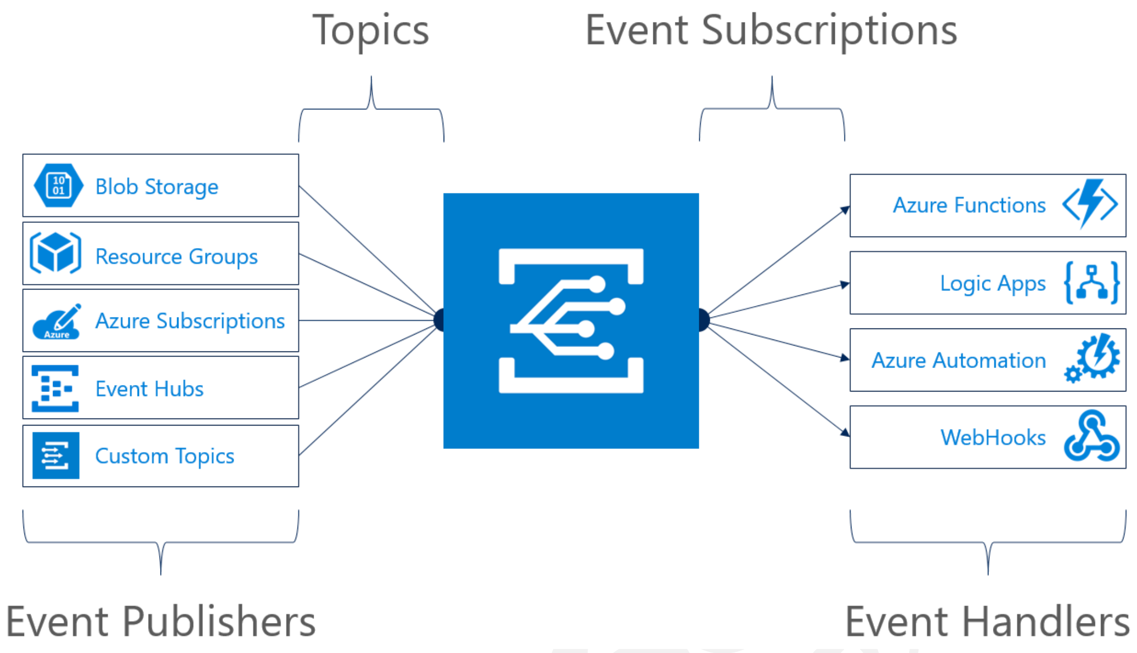
Each block in an append blob can be a different size, up to a maximum of 4 MB, and an append blob can include up to 50,000 blocks. The maximum size of an append blob is therefore slightly more than 195 GB (4 MB X 50,000 blocks).

Reacting to Blob storage events

Azure Storage events allow applications to react to the creation and deletion of blobs using modern serverless architectures. It does so without the need for complicated code or expensive and inefficient polling services. Instead, events are pushed through Azure Event Grid to subscribers such as Azure Functions, Azure Logic Apps, or even to your own custom http listener, and you only pay for what you use.

Blob storage events are reliably sent to the Event grid service which provides reliable delivery services to your applications through rich retry policies and dead-letter delivery.

Common Blob storage event scenarios include image or video processing, search indexing, or any file-oriented workflow. Asynchronous file uploads are a great fit for events. When changes are infrequent, but your scenario requires immediate responsiveness, event-based architecture can be especially efficient.



Blob storage accounts

Blob storage events are available in general-purpose v2 storage accounts and Blob storage accounts. General-purpose v2 storage accounts support all features for all storage services, including Blobs, Files, Queues, and Tables. A Blob storage account is a specialized storage account for storing your unstructured data as blobs (objects) in Azure Storage. Blob storage accounts are like 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.

Available Blob storage events

Event grid uses event subscriptions to route event messages to subscribers. Blob storage event subscriptions can include two types of events:

Event Name	Description
`Microsoft.Storage.BlobCreated`	Fired when a blob is created or replaced through the `PutBlob`, `PutBlockList`, or `CopyBlob` operations
`Microsoft.Storage.BlobDeleted`	Fired when a blob is deleted through a `DeleteBlob` operation

Event Schema

Blob storage events contain all the information you need to respond to changes in your data. You can identify a Blob storage event because the eventType property starts with "Microsoft.Storage". Additional information about the usage of Event Grid event properties is documented in **Event Grid event schema**[2].

2 https://docs.microsoft.com/en-us/azure/event-grid/event-schema

Filtering events

Blob event subscriptions can be filtered based on the event type and by the container name and blob name of the object that was created or deleted. Filters can be applied to event subscriptions either during the creation of the event subscription or at a later time. Subject filters in Event Grid work based on "begins with" and "ends with" matches, so that events with a matching subject are delivered to the subscriber. The subject of Blob storage events uses the format:

```
/blobServices/default/containers/<containername>/blobs/<blobname>
```

To match all events for a storage account, you can leave the subject filters empty. To match events from blobs created in a set of containers *sharing* a prefix, use a `subjectBeginsWith` filter like:

```
/blobServices/default/containers/containerprefix
```

To match events from blobs created in *specific* container, use a `subjectBeginsWith` filter like:

```
/blobServices/default/containers/containername/
```

To match events from blobs created in specific container sharing a blob name prefix, use a `subjectBeginsWith` filter like:

```
/blobServices/default/containers/containername/blobs/blobprefix
```

To match events from blobs created in specific container sharing a blob suffix, use a `subjectEndsWith` filter like ".log" or ".jpg".

Practices for consuming events

Applications that handle Blob storage events should follow a few recommended practices:

- As multiple subscriptions can be configured to route events to the same event handler, it is important not to assume events are from a particular source, but to check the topic of the message to ensure that it comes from the storage account you are expecting.
- Similarly, check that the eventType is one you are prepared to process, and do not assume that all events you receive will be the types you expect.
- As messages can arrive out of order and after some delay, use the etag fields to understand if your information about objects is still up-to-date. Also, use the sequencer fields to understand the order of events on any particular object.
- Use the blobType field to understand what type of operations are allowed on the blob, and which client library types you should use to access the blob. Valid values are either `BlockBlob` or `PageBlob`.
- Use the url field with the `CloudBlockBlob` and `CloudAppendBlob` constructors to access the blob.
- Ignore fields you don't understand. This practice will help keep you resilient to new features that might be added in the future.

Shared Access Signatures

A Shared Access Signature (SAS) is a URI that grants restricted access rights to containers, binary large objects (blobs), queues, and tables for a specific time interval. By providing a client with a Shared Access

Signature, you can enable them to access resources in your storage account without sharing your account key with them.

The Shared Access Signature URI query parameters incorporate all of the information necessary to grant controlled access to a storage resource. The URI query parameters specify the time interval over which the Shared Access Signature is valid, the permissions that it grants, the resource that is to be made available, and the signature that the storage services should use to authenticate the request.

Here is an example of a 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=2012-02
-12&st=2013-04-29T22%3A18%3A26Z&se=2013-04-30T02%3A23%3A26Z&s-
r=b&sp=rw&sig=Z%2FRHIX5Xcg0Mq2rqI3OlWTjEg2tYkboXr1P9ZUXDtkk%3
```

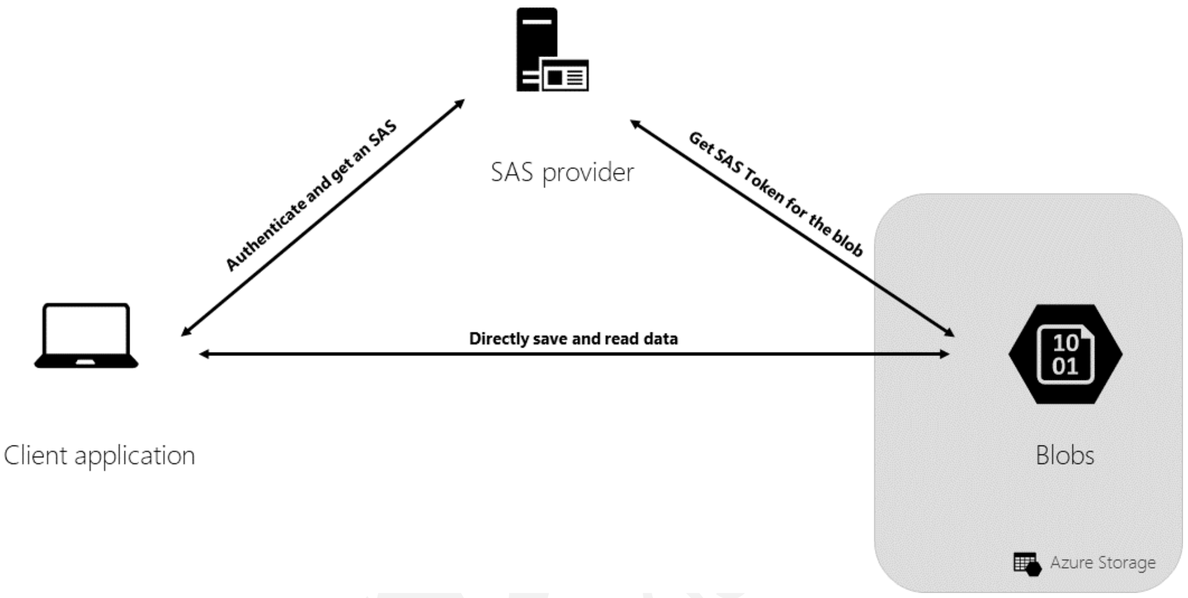
Component	Content	Description
Blob URI	https://myaccount.blob.core.windows.net/sascontainer/sasblob.txt	The address of the blob. Note that using HTTPS is highly recommended.
Storage services version	sv=2012-02-12	For Azure Storage services version 2012-02-12 and later, this parameter indicates the version to use.
Start time	st=2013-04-29T22%3A18%3A26Z	Specified in an International Organization for Standardization (ISO) 8061 format. If you want the SAS to be valid immediately, omit the start time.
Expiration time	se=2013-04-30T02%3A23%3A26Z	Specified in an ISO 8061 format.
Resource	sr=b	The resource is a blob.
Permissions	sp=rw	The permissions granted by the SAS include Read (r) and Write (w).
Signature	sig=Z%2FRHIX5Xcg0Mq2rqI3Ol-WTjEg2tYkboXr1P9ZUXDtkk%3D	Used to authenticate access to the blob. The signature is a HMAC function computed over a string to sign and a key by using the SHA256 algorithm and then encoded by using Base64 encoding.

Valet key pattern that uses Shared Access Signatures

A common scenario where an 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:

- Clients upload and download data via a front-end proxy service, which performs authentication. This front-end proxy service has the advantage of allowing the validation of business rules, but for large amounts of data or high-volume transactions, creating a service that can scale to match demand might be expensive or difficult.

- Using the valet key pattern, a lightweight service authenticates the client as needed and then generates an SAS. After 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.


SAS provider
Authenticate and get an SAS
Get SAS Token for the blob
Directly save and read data
Client application
Blobs
Azure Storage

-

Stored access policies

A Shared Access Signature can take one of two forms:

- **An ad hoc SAS.** When you create an ad hoc SAS, the start time, expiration time, and permissions for the SAS are all specified on the SAS URI (or implied in the case where the start time is omitted). This type of SAS can be created on a container, blob, table, or queue.
- **An SAS with a stored access policy.** A stored access policy is defined on a resource container—a blob container, table, or queue—and can be used to manage constraints for one or more Shared Access Signatures. When you associate an SAS with a stored access policy, the SAS inherits the constraints—the start time, expiration time, and permissions—defined for the stored access policy.

The difference between the two forms is important for one key scenario: revocation. An SAS is a URL, so anyone who obtains the SAS can use it regardless of who requested it to begin with. If an SAS is published publicly, it can be used by anyone in the world. 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 further into the future.

Working with Azure Blob storage

Setting and Retrieving Properties and Metadata for Blob Resources by using REST

Containers and blobs support custom metadata, represented as HTTP headers. Metadata headers can be set on a request that creates a new container or blob resource, or on a request that explicitly creates a property on an existing resource.

Metadata Header Format

Metadata headers are name/value pairs. The format for the header is:

```
x-ms-meta-name:string-value
```

Beginning with version 2009-09-19, metadata names must adhere to the naming rules for C# identifiers.

Names are case-insensitive. Note that metadata names preserve the case with which they were created, but are case-insensitive when set or read. If two or more metadata headers with the same name are submitted for a resource, the Blob service returns status code 400 (Bad Request).

The metadata consists of name/value pairs. The total size of all metadata pairs can be up to 8KB in size.

Metadata name/value pairs are valid HTTP headers, and so they adhere to all restrictions governing HTTP headers.

Operations on Metadata

Metadata on a blob or container resource can be retrieved or set directly, without returning or altering the content of the resource.

Note that metadata values can only be read or written in full; partial updates are not supported. Setting metadata on a resource overwrites any existing metadata values for that resource.

Retrieving Properties and Metadata

The GET and HEAD operations both retrieve metadata headers for the specified container or blob. These operations return headers only; they do not return a response body. The URI syntax for retrieving metadata headers on a container is as follows:

```
GET/HEAD https://myaccount.blob.core.windows.net/mycontainer?restype=con-
tainer
```

The URI syntax for retrieving metadata headers on a blob is as follows:

```
GET/HEAD https://myaccount.blob.core.windows.net/mycontainer/my-
blob?comp=metadata
```

Setting Metadata Headers

The PUT operation sets metadata headers on the specified container or blob, overwriting any existing metadata on the resource. Calling PUT without any headers on the request clears all existing metadata on the resource.

The URI syntax for setting metadata headers on a container is as follows:

```
PUT https://myaccount.blob.core.windows.net/mycontainer?comp=metadata-
?restype=container
```

The URI syntax for setting metadata headers on a blob is as follows:

```
PUT https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=metada-
ta
```

Standard HTTP Properties for Containers and Blobs

Containers and blobs also support certain standard HTTP properties. Properties and metadata are both represented as standard HTTP headers; the difference between them is in the naming of the headers. Metadata headers are named with the header prefix `x-ms-meta-` and a custom name. Property headers use standard HTTP header names, as specified in the Header Field Definitions section 14 of the HTTP/1.1 protocol specification.

The standard HTTP headers supported on containers include:

- `ETag`
- `Last-Modified`

The standard HTTP headers supported on blobs include:

- `ETag`
- `Last-Modified`
- `Content-Length`
- `Content-Type`
- `Content-MD5`
- `Content-Encoding`
- `Content-Language`
- `Cache-Control`
- `Origin`
- `Range`

Manipulating blob container properties in .NET

The `CloudStorageAccount` class contains the `CreateCloudBlobClient` method that gives you programmatic access to a client that manages your file shares:

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

To reference a specific blob container, you can use the `GetContainerReference` method of the `CloudBlobClient` class:

```
CloudBlobContainer container = client.GetContainerReference("images");
```

After you have a reference to the container, you can ensure that the container exists. This will create the container if it does not already exist in the Azure storage account:

```
container.CreateIfNotExists();
```

With a hydrated reference, you can perform actions such as fetching the properties (metadata) of the container by using the `FetchAttributesAsync` method of the `CloudBlobContainer` class:

```
await container.FetchAttributesAsync();
```

After the method is invoked, the local variable is hydrated with values for various container metadata. This metadata can be accessed by using the `Properties` property of the `CloudBlobContainer` class, which is of type `BlobContainerProperties`:

```
container.Properties
```

This class has properties that can be set to change the container, including (but not limited to) those in the following table.

Property	Description
`ETag`	This is a standard HTTP header that gives a value that is unchanged unless a property of the container is changed. This value can be used to implement optimistic concurrency with the blob containers.
`LastModified`	This property indicates when the container was last modified.
`PublicAccess`	This property indicates the level of public access that is allowed on the container. Valid values include Blob, Container, Off, and Unknown.
`HasImmutabilityPolicy`	This property indicates whether the container has an immutability policy. An immutability policy will help ensure that blobs are stored for a minimum amount of retention time.
`HasLegalHold`	This property indicates whether the container has an active legal hold. A legal hold will help ensure that blobs remain unchanged until the hold is removed.

Manipulating blob container metadata in .NET

Using the existing **CloudBlobContainer** variable (named *container*), you can set and retrieve custom metadata for the container instance. This metadata is hydrated when you call the `FetchAttributes` or `FetchAttributesAsync` method on your blob or container to populate the Metadata collection.

The following code example sets metadata on a container. In this example, we use the collection's Add method to set a metadata value:

```
container.Metadata.Add("docType", "textDocuments");
```

In the next example, we set the metadata value by using implicit key/value syntax:

```
container.Metadata["category"] = "guidance";
```

To persist the newly set metadata, you must call the SetMetadataAsync method of the `CloudBlobContainer` class:

```
await container.SetMetadataAsync();
```

The Lease Blob operation

The `Lease Blob` operation establishes and manages a lock on a blob for write and delete operations. The lock duration can be 15 to 60 seconds, or can be infinite. In versions prior to 2012-02-12, the lock duration is 60 seconds.

Important: Starting in version 2012-02-12, some behaviors of the `Lease Blob` operation differ from previous versions. For example, in previous versions of the `Lease Blob` operation you could renew a lease after releasing it. Starting in version 2012-02-12, this lease request will fail, while calls using older versions of `Lease Blob` still succeed.

The `Lease Blob` operation can be called in one of five modes:

- `Acquire`, to request a new lease.
- `Renew`, to renew an existing lease.
- `Change`, to change the ID of an existing lease.
- `Release`, to free the lease if it is no longer needed so that another client may immediately acquire a lease against the blob.
- `Break`, to end the lease but ensure that another client cannot acquire a new lease until the current lease period has expired.

Request

The `Lease Blob` request may be constructed as follows. HTTPS is recommended. Replace *myaccount* with the name of your storage account:

PUT Method Request URI	HTTP Version
`https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=lease`	HTTP/1.1

Emulated Storage Service URI

When making a request against the emulated storage service, specify the emulator hostname and Blob service port as `127.0.0.1:10000`, followed by the emulated storage account name:

PUT Method Request URI	HTTP Version
`http://127.0.0.1:10000/devstoreaccount1/mycontainer/myblob?comp=lease`	HTTP/1.0 HTTP/1.1

URI Parameters

The following additional parameters may be specified on the request URI.

Parameter	Description
`timeout`	Optional. The `timeout` parameter is expressed in seconds.

Request Headers

The following table describes required and optional request headers.

Request Header	Description
`Authorization`	Required. Specifies the authentication scheme, account name, and signature.
`Date` or `x-ms-date`	Required. Specifies the Coordinated Universal Time (UTC) for the request.
`x-ms-version`	Optional. Specifies the version of the operation to use for this request.
`x-ms-lease-id: <ID>`	Required to renew, change, or release the lease. The value of x-ms-lease-id can be specified in any valid GUID string format.

Request Header	Description
`x-ms-lease-action: <acquire ¦ renew ¦ change ¦ release ¦ break>`	`acquire`: Requests a new lease. If the blob does not have an active lease, the Blob service creates a lease on the blob and returns a new lease ID. If the blob has an active lease, you can only request a new lease using the active lease ID, but you can specify a new `x-ms-lease-duration`, including negative one (-1) for a lease that never expires. `renew`: Renews the lease. The lease can be renewed if the lease ID specified on the request matches that associated with the blob. Note that the lease may be renewed even if it has expired as long as the blob has not been modified or leased again since the expiration of that lease. When you renew a lease, the lease duration clock resets. `change`: Version 2012-02-12 and newer. Changes the lease ID of anactive lease. A `change` must include the current lease ID in x-ms-lease-id and a new lease ID in x-ms-proposed-lease-id. `release`: Releases the lease. The lease may be released if the lease ID specified on the request matches that associated with the blob. Releasing the lease allows another client to immediately acquire the lease for the blob as soon as the release is complete. `break`: Breaks the lease, if the blob has an active lease. Once a lease is broken, it cannot be re-newed. Any authorized request can break the lease; the request is not required to specify a matching lease ID. When a lease is broken, the lease break period is allowed to elapse, during which time no lease operation except `break` and `release` can be performed on the blob. When a lease is successfully broken, the response indicates the interval in seconds until a new lease can be acquired. A lease that has been broken can also be released, in which case another client may imme-diately acquire the lease on the blob.

Request Header	Description
`x-ms-lease-break-period: N`	Version 2012-02-12 and newer, optional. For a `break` operation, this is the proposed duration of seconds that the lease should continue before it is broken, between 0 and 60 seconds. This break period is only used if it is shorter than the time remaining on the lease. If longer, the time remaining on the lease is used. A new lease will not be available before the break period has expired, but the lease may be held for longer than the break period. If this header does not appear with a `break` operation, a fixed-duration lease breaks after the remaining lease period elapses, and an infinite lease breaks immediately.
`x-ms-lease-duration: -1 ¦ N`	Version 2012-02-12 and newer, only allowed and required on an `acquire` operation. Specifies the duration of the lease, in seconds, or negative one (-1) for a lease that never expires. A non-infinite lease can be between 15 and 60 seconds. A lease duration cannot be changed using `renew` or `change`.
`x-ms-proposed-lease-id: <ID>`	Version 2012-02-12 and newer, optional for `acquire`, required for `change`. Proposed lease ID, in a GUID string format. The Blob service returns `400 (Invalid request)` if the proposed lease ID is not in the correct format.
`Origin`	Optional. Specifies the origin from which the request is issued. The presence of this header results in cross-origin resource sharing headers on the response.
`x-ms-client-request-id`	Optional. Provides a client-generated, opaque value with a 1 KB character limit that is recorded in the analytics logs when storage analytics logging is enabled. Using this header is highly recommended for correlating client-side activities with requests received by the server.

Request Body

None.

Sample Request

The following sample request shows how to acquire a lease:

```
Request Syntax:
PUT https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=lease 
HTTP/1.1

Request Headers:
```

```
x-ms-version: 2015-02-21
x-ms-lease-action: acquire
x-ms-lease-duration: -1
x-ms-proposed-lease-id: 1f812371-a41d-49e6-b123-f4b542e851c5
x-ms-date: <date>
Authorization: SharedKey testaccount1:esSKMOYdK4o+nGTuTyeOLBI+xqnqi6aBmi-
W4XI699+o=
```

Response

The response includes an HTTP status code and a set of response headers.

Status Code

The success status codes returned for lease operations are the following:

- `Acquire`: A successful operation returns status code 201 (Created).
- `Renew`: A successful operation returns status code 200 (OK).
- `Change`: A successful operation returns status code 200 (OK).
- `Release`: A successful operation returns status code 200 (OK).
- `Break`: A successful operation returns status code 202 (Accepted).

Response Headers

The response for this operation includes the following headers. The response may also include additional standard HTTP headers. All standard headers conform to the HTTP/1.1 protocol specification.

Syntax	Description
`ETag`	The `ETag` header contains a value that you can use to perform operations conditionally. This header is returned for requests made against version 2013-08-15 and later, and the ETag value will be in quotes. The `Lease Blob` operation does not modify this property.
`Last-Modified`	The date/time that the blob was last modified. The date format follows RFC 1123. Any write operation on the blob, including updates on the blob's metadata or properties, changes the last-modified time of the blob. The `Lease Blob` operation does not modify this property.

Syntax	Description
`x-ms-lease-id: <id>`	When you request a lease, the Blob service returns a unique lease ID. While the lease is active, you must include the lease ID with any request to write to the blob, or to renew, change, or release the lease. A successful renew operation also returns the lease ID for the active lease.
`x-ms-lease-time: seconds`	Approximate time remaining in the lease period, in seconds. This header is returned only for a successful request to break the lease. If the break is immediate, 0 is returned.
`x-ms-request-id`	This header uniquely identifies the request that was made and can be used for troubleshooting the request.
`x-ms-version`	Indicates the version of the Blob service used to execute the request. This header is returned for requests made against version 2009-09-19 and later.
`Date`	A UTC date/time value generated by the service that indicates the time at which the response was initiated.
`Access-Control-Allow-Origin`	Returned if the request includes an `Origin` header and CORS is enabled with a matching rule. This header returns the value of the origin request header in case of a match.
`Access-Control-Expose-Headers`	Returned if the request includes an `Origin` header and CORS is enabled with a matching rule. Returns the list of response headers that are to be exposed to the client or issuer of the request.
`Access-Control-Allow-Credentials`	Returned if the request includes an `Origin` header and CORS is enabled with a matching rule that does not allow all origins. This header will be set to true.

Authorization

This operation can be called by the account owner and by any client with a shared access signature that has permission to write to this blob or its container.

Move items in Blob storage by using AzCopy

Like with Azure Files, you can use AzCopy to copy blobs between storage containers. The basic syntax for AzCopy commands is:

```
AzCopy /Source:<source> /Dest:<destination> [Options]
```

Downloading blobs

You can download blobs from a container to your local computer by using the following command:

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer /
Dest:C:\myfolder /SourceKey:key /Pattern:"abc.txt"
```

Alternatively, you can remove the **/Pattern** option to download all the blobs from the container:

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer /
Dest:C:\myfolder /SourceKey:key /S
```

Remember, blobs do not have a folder hierarchy. The entire path of the blob constitutes the name. Assuming you are running the **download** command in the **C:\myfolder** path and are downloading a blob named **vd1\a.txt**, the AzCopy tool will hydrate the blob in the following local path:

```
C:\myfolder\vd1\a.txt
```

You can also perform pattern matching before downloading blobs. In this example, all blobs beginning with the prefix "**a**" are downloaded:

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer /
Dest:C:\myfolder /SourceKey:key /Pattern:a /S
```

Copying blobs between containers

The simplest example of copying blobs would be to copy blobs from one container in the storage account to another:

```
AzCopy /Source:https://myaccount.blob.core.windows.net/mycontainer1 /
Dest:https://myaccount.blob.core.windows.net/mycontainer2 /SourceKey:key /
DestKey:key /Pattern:abc.txt
```

Because you used the full URI to copy blobs within a storage account, you can copy blobs to another storage account using the destination URI. The only difference in this example is the base of the URI referencing a different storage account:

```
AzCopy /Source:https://sourceaccount.blob.core.windows.net/mycontainer1 /
Dest:https://destaccount.blob.core.windows.net/mycontainer2 /SourceKey:key1
/DestKey:key2 /Pattern:abc.txt
```

By default, AzCopy copies data between two storage endpoints asynchronously. Therefore, the copy operation runs in the background by using spare bandwidth capacity that has no Service Level Agreement (SLA) in terms of how fast a blob is copied, and AzCopy periodically checks the copy status until the copying has completed or failed.

The **/SyncCopy** option ensures that the copy operation gets a 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
```

Copying blobs to Azure Files shares

Using your knowledge of AzCopy to copy blobs and file shares, you can use the same commands to copy between storage types. The only things you would need to change are the URIs used in the **/Source** and **/Dest** options. In this first example, we copy content from an Azure Files share to a blob:

```
AzCopy /Source:https://myaccount1.file.core.windows.net/myfileshare/ /
Dest:https://myaccount2.blob.core.windows.net/mycontainer/ /SourceKey:key1
/DestKey:key2 /S
```

In this next example, we copy content from a blob to an Azure Files share:

```
AzCopy /Source:https://myaccount1.blob.core.windows.net/mycontainer/ /
Dest:https://myaccount2.file.core.windows.net/myfileshare/ /SourceKey:key1 /
DestKey:key2 /S
```

Review Questions

Module 4 review questions

Shared Access Signature

A Shared Access Signature (SAS) is a URI that grants restricted access rights to containers, binary large objects (blobs), queues, and tables for a specific time interval. By providing a client with a Shared Access Signature, you can enable them to access resources in your storage account without sharing your account key with them. What are the two forms an SAS can take?

> Click to see suggested answer

- **An ad hoc SAS.** When you create an ad hoc SAS, the start time, expiration time, and permissions for the SAS are all specified on the SAS URI (or implied in the case where the start time is omitted). This type of SAS can be created on a container, blob, table, or queue.
- **An SAS with a stored access policy.** A stored access policy is defined on a resource container—a blob container, table, or queue—and can be used to manage constraints for one or more Shared Access Signatures. When you associate an SAS with a stored access policy, the SAS inherits the constraints—the start time, expiration time, and permissions—defined for the stored access policy.

Copying Blobs between containers

Like with Azure Files, you can use AzCopy to copy blobs between storage containers. By default, does AzCopy copy data synchronously or asynchronously?

> Click to see suggested answer

By default, AzCopy copies data between two storage endpoints asynchronously. Therefore, the copy operation runs in the background by using spare bandwidth capacity that has no Service Level Agreement (SLA) in terms of how fast a blob is copied, and AzCopy periodically checks the copy status until the copying has completed or failed.

Microsoft

Microsoft Official Course