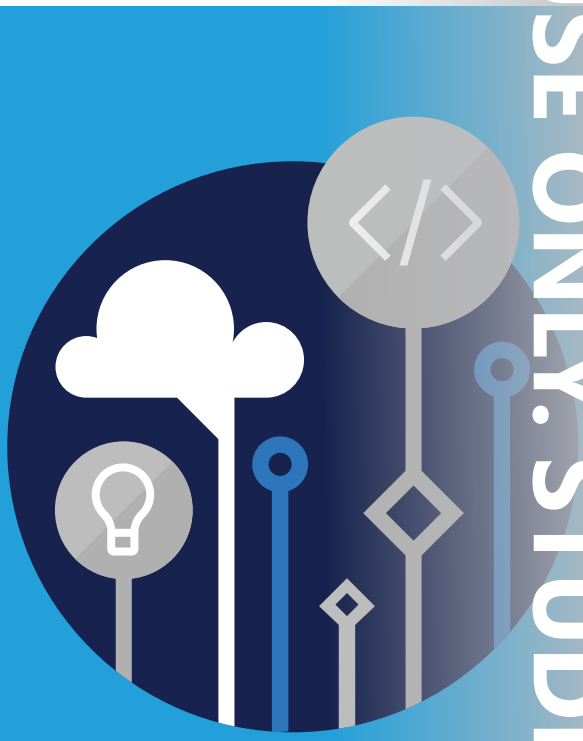
Microsoft

Microsoft Official Course



AZ-300T03

Understanding Cloud Architect Technology Solutions

AZ-300T03

Understanding Cloud Architect Technology Solutions

Contents

Module 0 Start Here

Welcome to Understanding Cloud Architect Technology Solutions

Welcome to Understanding Cloud Architect Technology Solutions

Course Overview: Understanding Cloud Architect Technology Solutions

Welcome to *Understanding Cloud Architect Technology Solutions*. This course is part of a series of five courses to help students prepare for Microsoft's Azure Solutions Architect technical certification exam AZ-300: Microsoft Azure Architect Technologies. These courses are designed for IT professionals and developers with experience and knowledge across various aspects of IT operations, including networking, virtualization, identity, security, business continuity, disaster recovery, data management, budgeting, and governance.

This course educates IT professionals on how operations are accomplished both in parallel and asynchronously. By using the Azure Application Architecture Guide and Azure reference architectures as a basis, you will understand how monitoring and telemetry are critical for gaining insight into a system. You will explore the cloud design patterns that are important. For example, partitioning workloads of a modular application divided into functional units that can be integrated into a larger application. In such cases, each module handles a portion of the application's overall functionality and represents a set of related concerns.

Also, you will understand how load balancing the application traffic, or load, can be distributed among various endpoints using algorithms. For example, load balancers allowing multiple instances of your website to be created and thus allowing them to behave in a predictable manner. In Azure, it is possible to use virtual load balancers, which are hosted in virtual machines, allowing for very specific load balancer configurations.

Lastly, an overview of hybrid networking that includes site-to-site connectivity, point-to-site connectivity, and the combination of the two.

The outline for this course is as follows:

Module 1 - Selecting Compute and Storage Solutions

This module includes the following topics:

- Azure Architecture Center
- Cloud design patterns
- Competing consumers pattern
- Cache-aside pattern
- Sharding patterns to divide a data store into horizontal partitions, or shards

This module contains the online lab Implementing Azure Storage Access Controls.

Module 2 - Hybrid Networking

This module includes the following topics:

- Site-to-site connectivity
- Point-to-site connectivity
- Combining site-to-site and point-to-site connectivity
- Virtual network–to–virtual network connectivity

As well as connecting across cloud providers for failover, backup, or even migration between providers such as AWS.

Module 3 – Measuring Throughput and Structure of Data Access

This module includes the following topics:

- DTUs – Azure SQL Database
- RUs – Azure Cosmos DB
- Structured and unstructured data
- Using structured data stores

This module contains the online lab Implementing Azure Load Balancer Standard.

Module 4 - Implementing Authentication

Topics for this module include:

- Implementing authentication in applications (certificates, Azure AD, Azure AD Connect, token-based)
- Implementing multi-factor authentication
- Claims-based authorization
- Role-based access control (RBAC) authorization

This module contains the online lab Implementing Custom Role Based Access Control (RBAC) Roles.

Module 5 - Implementing Secure Data

Topics for this module include:

- End-to-end encryption
- Implementing Azure confidential computing

- Implementing SSL and TLS communications
- Managing cryptographic keys in Azure Key Vault

Module 6 - Business Continuity and Resiliency in Azure

- Business Continuity and Resiliency
- High Availability and Disaster Recovery
- Resiliency
- Application Design
- Testing, Deployment, and Maintenance
- Data Management
- Monitoring and Disaster Recovery

What You'll Learn:

- Design and Connectivity Patterns
- Hybrid Networking
- Address Durability of Data and Caching
- Measure Throughput and Structure of Data Access

Prerequisites:

Successful Cloud Solutions Architects begin this role with practical experience with operating systems, virtualization, cloud infrastructure, storage structures, billing, and networking.

Module 1 Module Selecting Compute and Storage Solutions

Design and Connectivity Patterns

Azure Architecture Center

The cloud is changing the way applications are designed. Instead of being monoliths, applications are decomposed into smaller, decentralized services. These services communicate through APIs or by using asynchronous messaging or eventing. Applications scale horizontally, adding new instances as demand requires.

These trends bring new challenges. The application state is distributed. Operations are done in parallel and asynchronously. The system as a whole must be resilient when failures occur. Deployments must be automated and predictable. Monitoring and telemetry are critical for gaining insight into the system. The Azure Architecture Center is designed to help you navigate these changes.

Azure Application Architecture Guide

To view the guide, refer to **https://docs.microsoft.com/azure/architecture/guide/**

The Microsoft Azure Application Architecture Guide is intended for application architects, developers, and operations teams and describes how to design and implement common software architectures. It is not service specific and includes the following sections:

- List of architecture styles
- Technology choices for each component of a design
- High-level design principles for applications
- Software quality metrics

Azure reference architectures

To view the guide, refer to **https://docs.microsoft.com/azure/architecture/reference-architectures/**

The Azure Reference Architectures landing page is a collection of architectural diagrams and explanations for the most-common cloud solution designs. These architectures range from data-centric applications to n-tier web applications to DevOps platforms and even to infrastructure-only deployments. Each reference architecture includes:

- A description of the architectural diagram
- Common recommendations
- Considerations in the following categories:

--Scalability

--Security

--Availability

--Manageability

--Steps on how to deploy an example solution

Cloud design patterns

To view the guide, refer to **https://docs.microsoft.com/azure/architecture/patterns/**

The Azure Architecture Center contains a guide published by the Patterns & Practices team that provides not just guidance but also over 20 examples of the most-common design patterns used for cloud applications. These patterns are neither specific to Microsoft ASP.NET or to Microsoft. Each pattern describes the problem that the pattern addresses, considerations for applying the pattern, and an example based on Microsoft Azure. Most of the patterns include code samples or snippets that show how to implement the pattern in Azure. However, most of the patterns are relevant to any distributed system, whether hosted in Azure or in other cloud platforms.

Application design concepts

Before you dive into the cloud design patterns, it is important to understand a few key design concepts.

Partitioning workloads

A modular application is divided into functional units, also referred to as modules, which can be integrated into a larger application. Each module handles a portion of the application's overall functionality and represents a set of related concerns. Modular applications make it easier to design both current and future iterations of your application. Existing modules can be extended, revised, or replaced to iterate changes to your full application. Modules can also be tested, distributed, and otherwise verified in isolation. Modular design benefits are well understood by many developers and architects in the software industry.

Load balancing

Load balancing is a computing concept where the application traffic, or load, is distributed among various endpoints by using algorithms. When you use a load balancer, multiple instances of your website can be created, and they can behave in a predictable manner. This provides the flexibility to grow or shrink the number of instances in your application without changing the expected behavior.

Load balancing strategy

There are a couple of things to consider when choosing a load balancer. First, you must decide whether you want to use a physical or a virtual load balancer. In Azure, it is possible to use virtual load balancers, which are hosted in virtual machines, if a company requires a very specific load balancer configuration.

After you select a specific load balancer, you need to select a load balancing algorithm. You can use various algorithms, such as round robin or random choice. For example, round robin selects the next instance for each request based on a predetermined order that includes all of the instances.

Other configuration options, such as affinity or stickiness, exist for load balancers. For example, stickiness allows you to determine whether a subsequent request from the same client machine should be routed to the same service instance. This might be required in scenarios where your application servers have a concept of state.

Transient fault handling

One of the primary differences between developing applications on-premises and in the cloud is the way you design your application to handle transient errors. Transient errors are as errors that occur due to temporary interruptions in the service or to excess latency. Many of these temporary issues are self-healing and can be resolved by exercising a retry policy.

Retry policies define when and how often a connection attempt should be retried when a temporary failure occurs. Simply retrying in an infinite loop can be just as dangerous as infinite recursion. A break in the circuit must eventually be defined so that the retries are aborted if the error is determined to be of a serious nature and not just a temporary issue.

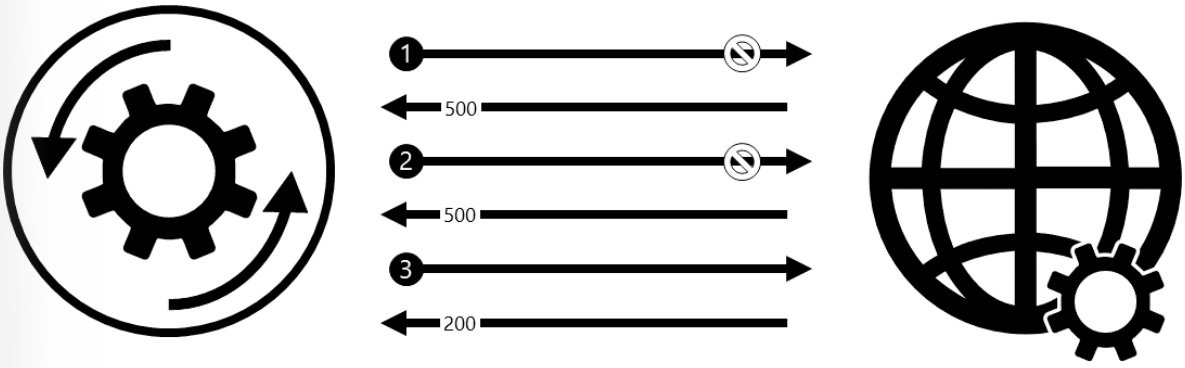
Transient fault handling is a pattern that makes your application more resilient by handling temporary issues in a robust manner. This is done by managing connections and implementing a retry policy. This pattern is already implemented in many common Microsoft .NET libraries, such as Entity Framework, and in the Azure software development kit (SDK). This pattern is also implemented in the Microsoft Enterprise Library in such a generic manner that it can be brought into a wide variety of application scenarios.

Queues

Queueing is both a mathematical theory and a messaging concept in computer science. In cloud applications, queues are critical for managing requests between application modules in a manner such that they provide a degree of consistency regardless of the behavior of the modules.

An application might already have a direct connection to other application modules using direct method invocation, a two-way service, or any other streaming mechanism. If one of the application modules experiences a transient issue, this connection is severed and causes an immediate application failure. You can use a third-party queue to persist the requests beyond a temporary failure. Requests can also be audited independently of the primary application, because they are stored in the queue mechanism.

Retry Pattern



Problem: intermittent errors with cloud services

An application that communicates with elements running in the cloud must be sensitive to the transient faults that can occur in this environment. Such faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that arise when a service is busy.

These faults are typically self-correcting, and if the action that triggered a fault is repeated after a suitable delay, it is likely to be successful. For example, a database service that is processing a large number of concurrent requests may implement a throttling strategy that temporarily rejects any further requests until its workload has eased. An application attempting to access the database may fail to connect, but if it tries again after a suitable delay, it may succeed.

Solution: application logic to retry requests that have temporarily failed

In the cloud, transient faults are not uncommon, and an application should be designed to handle them elegantly and transparently, minimizing the effects that such faults might have on the business tasks that the application is performing.

If an application detects a failure when it attempts to send a request to a remote service, it can handle the failure by retrying the application logic after a short wait. For the more-common transient failures, the period between retries should be chosen so as to spread requests from multiple instances of the application as evenly as possible. This can reduce the chance of a busy service continuing to be overloaded. If many instances of an application are continually bombarding a service with retry requests, it may take the service longer to recover.

If the request still fails, the application can wait again and make another attempt. There should be a limit on attempts to avoid sending endless requests to a service that may actually be completely inoperable. All code that accesses the remote service should be implemented using a retry policy such as the one described here.

Competing consumers pattern

Problem: handling variable quantities of requests

An application running in the cloud may be expected to handle a large number of requests. The number of requests could vary significantly over time for many reasons. A sudden burst in user activity or aggregated requests coming from multiple tenants may cause an unpredictable workload. At peak hours, a system might need to process many hundreds of requests per second, while at other times, the number could be very small. Additionally, the nature of the work performed to handle these requests might be highly variable.

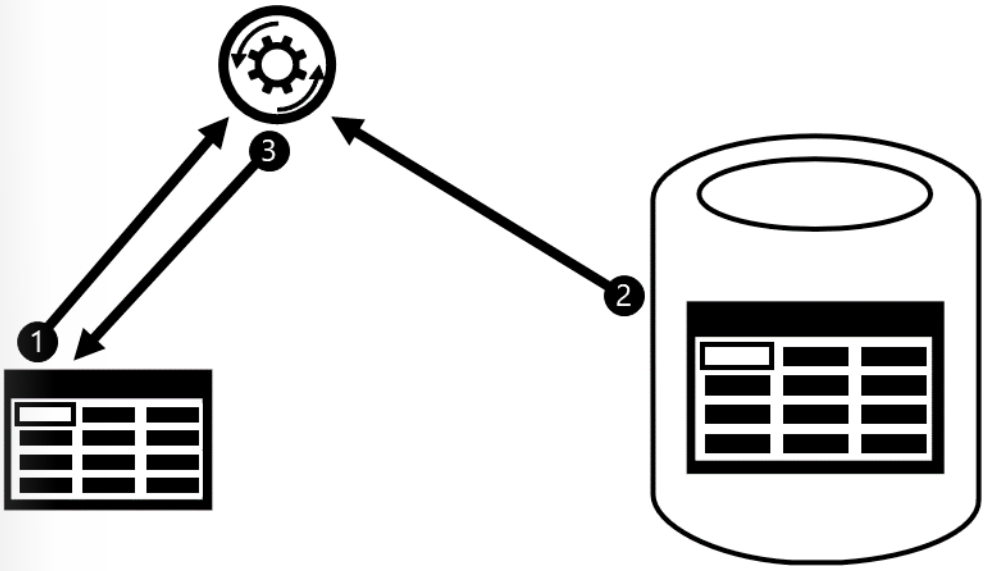
Using a single instance of the consumer service might cause that instance to become flooded with requests, or the messaging system may be overloaded by an influx of messages coming from the application.

Solution: asynchronous messaging with variable quantities of message producers and consumers

Rather than processing each request synchronously, a common technique is for the application to pass them through a messaging system to another service (a consumer service) that handles them asynchronously. This strategy helps to ensure that the business logic in the application is not blocked while the requests are being processed.

A message queue can be used to implement the communication channel between the application and the instances of the consumer service. To handle fluctuating workloads, the system can run multiple instances of the consumer service. The application posts requests in the form of messages to the queue, and the consumer service instances receive messages from the queue and process them. This approach enables the same pool of consumer service instances to handle messages from any instance of the application.

Cache-aside pattern



Problem: cached data consistency

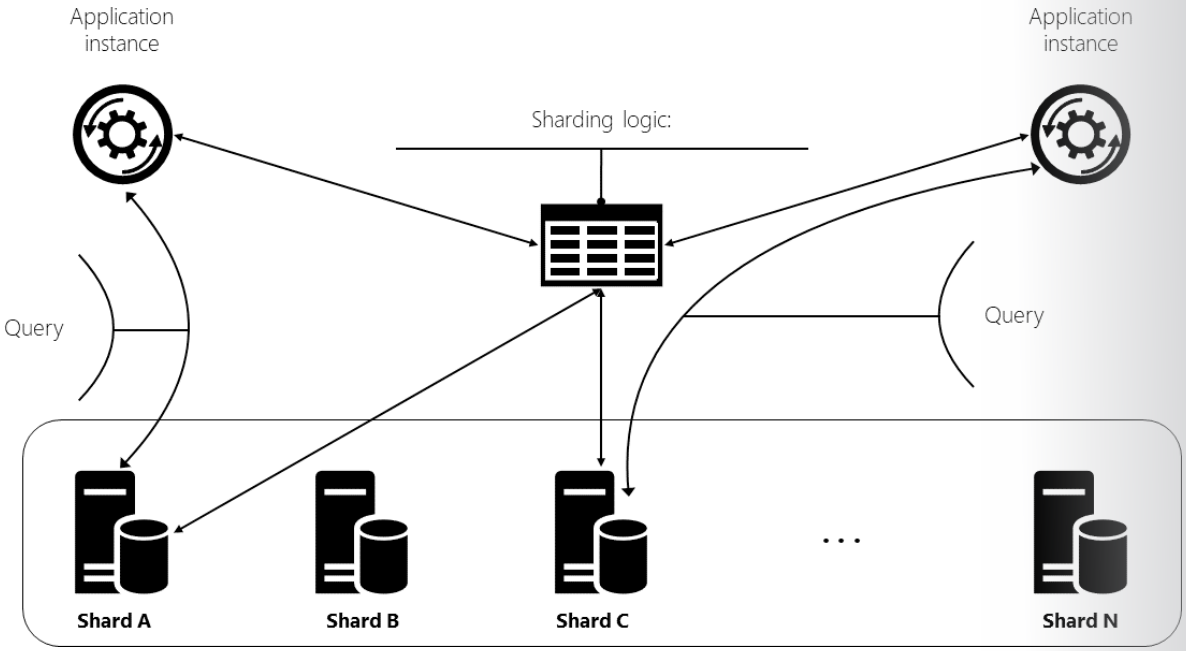
Applications use a cache to optimize repeated access to information held in a data store. However, it is usually impractical to expect that cached data will always be completely consistent with the data in the data store. Application developers should consider a strategy that helps to ensure that the data in the cache is up-to-date as much as possible but that can also detect and handle situations that arise when the data in the cache has become stale.

Solution: read-through and write-through caching

Many commercial caching systems provide read-through and write-through/write-behind operations. In these systems, an application retrieves data by referencing the cache. If the data is not in the cache, it is transparently retrieved from the data store and added to the cache. Any modifications to data held in the cache are automatically written back to the data store, as well.

For caches that do not provide this functionality, it is the responsibility of the applications that use the cache to maintain the data in the cache. An application can emulate the functionality of read-through caching by implementing the cache-aside strategy. This strategy effectively loads data into the cache on demand if it's not already available in the cache.

Sharding pattern



Problem: hosting large volumes of data in a traditional single-instance store

A data store hosted by a single server may be subject to limitations in the following areas:

- **Storage space.** A data store for a large-scale cloud application may be expected to contain a huge volume of data that could increase significantly over time. A server typically provides only a finite amount of disk storage, but it may be possible to replace existing disks with larger ones or to add disks to a machine as data volumes grow. However, the system will eventually reach a hard limit whereby it is not possible to easily increase the storage capacity on a given server.
- **Computing resources.** A cloud application may be required to support a large number of concurrent users, each of whom runs queries that retrieve information from the data store. A single server hosting the data store may not be able to provide the necessary computing power to support this load, resulting in extended response times for users and frequent failures as applications attempting to store and retrieve data time out. It may be possible to add memory or upgrade processors, but the system will reach a limit when it is not possible to increase the compute resources any further.
- **Network bandwidth.** Ultimately, the performance of a data store running on a single server is governed by the rate at which the server can receive requests and send replies. It is possible that the volume of network traffic might exceed the capacity of the network used to connect to the server, resulting in failed requests.
- **Geography.** It may be necessary to store data generated by specific users in the same region as those users for legal, compliance, or performance reasons or to reduce the latency of data access. If the users are dispersed across different countries and regions, it may not be possible to store all the data for the application in a single data store.

Scaling vertically by adding more disk capacity, processing power, memory, and network connections may postpone the effects of some of these limitations, but that is likely to be only a temporary solution. A

commercial cloud application capable of supporting large numbers of users and high volumes of data must be able to scale almost indefinitely, so vertical scaling is not necessarily the best solution.

Solution: partitioning data horizontally across many nodes

Divide the data store into horizontal partitions, or shards. Each shard has the same schema but holds its own distinct subset of the data. A shard is a data store in its own right (it can contain the data for many entities of different types) running on a server acting as a storage node.

Sharding physically organizes the data. When an application stores and retrieves data, the sharding logic directs the application to the appropriate shard. This sharding logic may be implemented as part of the data access code in the application, or it could be implemented by the data storage system if it transparently supports sharding.

Abstracting the physical location of the data in the sharding logic provides a high level of control over which shards contain which data, and it enables data to migrate between shards without a reworking of the business logic of an application if the data in the shards needs to be redistributed later (for example, if the shards become unbalanced). The tradeoff is the additional data access overhead required in determining the location of each data item as it is retrieved.

To help ensure optimal performance and scalability, it is important to split the data in a way that is appropriate for the types of queries the application performs. In many cases, it is unlikely that the sharding scheme will exactly match the requirements of every query. For example, in a multitenant system, an application may need to retrieve tenant data by using the tenant ID, but it may also need to look up this data based on some other attribute, such as the tenant's name or location. To handle these situations, implement a sharding strategy with a shard key that supports the most commonly performed queries.

Online Lab - Implementing Azure Storage Access Controls

Lab Steps

Online Lab: Implementing Azure Storage access controls

NOTE: For the most recent version of this online lab, see: **https://github.com/MicrosoftLearning/AZ-300-MicrosoftAzureArchitectTechnologies**

Scenario

Adatum Corporation wants to protect content residing in Azure Storage

Objectives

After completing this lab, you will be able to:

- Create an Azure Storage account.
- Upload data to Azure Storage.
- Implement Azure Storage access controls

Lab Setup

Estimated Time: 30 minutes

User Name: **Student**

Password: **Pa55w.rd**

Exercise 1: Creating and configuring an Azure Storage account

The main tasks for this exercise are as follows:

1. Create a storage account in Azure
2. View the properties of the storage account

Task 1: Create a storage account in Azure

1. From the lab virtual machine, start Microsoft Edge and browse to the Azure portal at **http://portal.azure.com** and sign in by using the Microsoft account that has the Owner role in the target Azure subscription.
2. From Azure Portal, create a new storage account with the following settings:
   - Subscription: the name of the target Azure subscription
   - Resource group: a new resource group named **az3000201-LabRG**
   - Storage account name: any valid, unique name between 3 and 24 characters consisting of lowercase letters and digits
   - Location: the name of the Azure region that is available in your subscription and which is closest to the lab location
   - Performance: **Standard**
   - Account kind: **Storage (general purpose v1)**
   - Replication: **Locally-redundant storage (LRS)**
   - Secure transfer required: **Disabled**
   - Virtual network: **All networks**
   - Hierarchical namespace: **Disabled**
3. Wait for the storage account to be provisioned. This will take about a minute.

Task 2: View the properties of the storage account

1. In Azure Portal, with your storage account blade open, review the **Overview** section, including the location, replication, and performance settings.
2. Display the **Access keys** blade. On the access keys blade, note that you have the option of copying the values of storage account names including key1 and key2. You also have the ability to regenerate both keys.
3. Display the **Configuration** blade.
4. On the **Configuration** blade, notice that you have the option of performing an upgrade to **General Purpose v2** account and changing the replication settings. However, you cannot change the performance setting (this can only be assigned when the storage account is created).

Result: After you completed this exercise, you have created your Azure Storage and examined its properties.

Exercise 2: Creating and managing blobs

The main tasks for this exercise are as follows:

1. Create a container
2. Upload data to the container by using the Azure portal
3. Access content of Azure Storage account by using a SAS token

Task 1: Create a container

1. In the Azure portal, navigate to the blade displaying the properties of the storage account you created in the previous task.
2. From the storage account blade, create a new blob container with the following settings:
   - Name: **labcontainer**
   - Access type: **Private**

Task 2: Upload data to the container by using the Azure portal

1. In the Azure portal, navigate to the **labcontainer** blade.
2. From the **labcontainer** blade, upload the file: **C:\Windows\ImmersiveControlPanel\images\splashscreen.contrast-white_scale-400.png**.

Task 3: Access content of Azure Storage account by using a SAS token

1. From the **labcontainer** blade, identify the URL of the newly uploaded blob.
2. Start Microsoft Edge and navigate to that URL.
3. Note the **ResourceNotFound** error message. This is expected since the blob is residing in a private container, which requires authenticated access.
4. Switch to the Microsoft Edge window displaying the Azure portal and, on the **splashscreen.contrast-white_scale-400.png** blade, switch to the **Generate SAS** tab.
5. On the **Generate SAS** tab, enable the **HTTP** option and generate blob SAS token and the corresponding URL.
6. Open a new Microsoft Edge window and, in the navigate to the URL generated in the previous step.

7. Note that you can view the image. This is expected since this time you are authorized to access the blob based on the SAS token included in the URL.
8. Close the Microsoft Edge window displaying the image.

Task 4: Access content of Azure Storage account by using a SAS token and a stored access policy.

1. In the Azure portal, navigate to the **labcontainer** blade.
2. From the **labcontainer** blade, navigate to the **labcontainer - Access policy** blade.
3. Add a new policy with the following settings:
   - Identifier: **labcontainer-read**
   - Permissions: **Read**
   - Start time: current date and time
   - Expiry time: current date and time + 24 hours
4. In the Azure portal, in the Microsoft Edge window, start a **PowerShell** session within the **Cloud Shell**.
5. If you are presented with the **You have no storage mounted** message, configure storage using the following settings:
   - Subsciption: the name of the target Azure subscription
   - Cloud Shell region: the name of the Azure region that is available in your subscription and which is closest to the lab location
   - Resource group: **az3000201-LabRG**
   - Storage account: a name of a new storage account
   - File share: a name of a new file share
6. From the Cloud Shell pane, run the following to identify the storage account resource you created in the first exercise of this lab and store it in a variable:

```
$storageAccount = (Get-AzStorageAccount -ResourceGroupName az3000201-LabRG)
[0]
```

7. From the Cloud Shell pane, run the following to establish security context granting full control to the storage account:

```
$keyContext = $storageAccount.Context
```

8. From the Cloud Shell pane, run the following to create a blob-specific SAS token based on the access policy you created in the previous task:

```
$sasToken = New-AzStorageBlobSASToken -Container 'labcontainer' -Blob
'splashscreen.contrast-white_scale-400.png' -Policy labcontainer-read
-Context $keyContext
```

9. From the Cloud Shell pane, run the following to establish security context based on the newly created SAS token:

```
$sasContext = New-AzStorageContext $storageAccount.StorageAccountName
-SasToken $sasToken
```

10. From the Cloud Shell pane, run the following to retrieve properties of the blob:

```
Get-AzStorageBlob -Container 'labcontainer' -Blob 'splashscreen.con-
trast-white_scale-400.png' -Context $sasContext
```

11. Verify that you successfully accessed the blob.
12. Minimize the Cloud Shell pane.

Task 5: Invalidate a SAS token by modifying its access policy.

1. In the Azure portal, navigate to the **labcontainer - Access policy** blade.
2. Edit the existing policy **labcontainer-read** by setting its start and expiry time to yesterday's date.
3. Reopen the Cloud Shell pane.
4. From the Cloud Shell pane, re-run the following to attempt retrieving properties of the blob:

```
Get-AzStorageBlob -Container 'labcontainer' -Blob 'splashscreen.con-
trast-white_scale-400.png' -Context $sasContext
```

5. Verify that you no longer can access the blob.

Result: After you completed this exercise, you have created a blob container, uploaded a file into it, and tested access control by using a SAS token and a stored access policy.

Review Question

Module 1 Review Questions

Design Concepts

You are designing a solution for an organization. The solution will be entirely cloud-based.

You must implement a distributed model for workloads.

Which design patterns Should you consider?

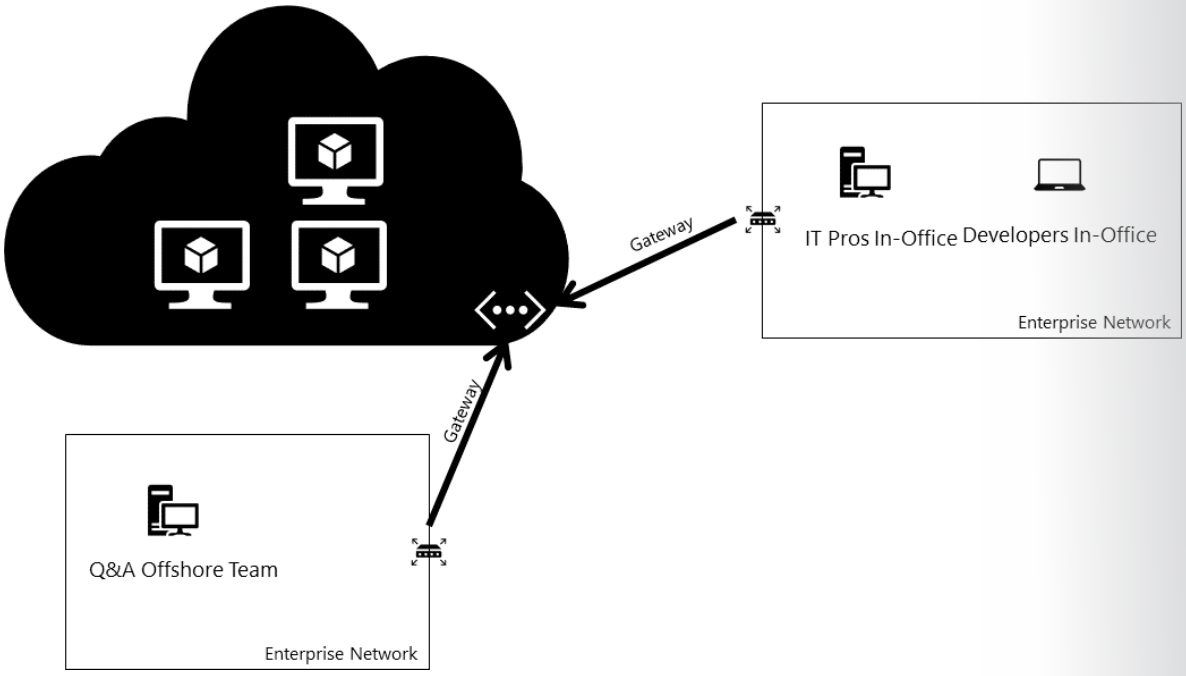
Suggested Answer ↓

• **Partitioning workloads**
A modular application is divided into functional units that each handles a portion of the application's functionality.
• **Load balancing**
Load balancing is a computing concept where the application traffic, or load, is distributed among various endpoints by using algorithms.
• **Load balancing strategy**
There are a couple of things to consider when choosing a load balancer. First, you must decide whether you want to use a physical or a virtual load balancer. You need to select a load-balancing algorithm suc as round robin or random choice.
• **Transient fault handling**
Transient errors are as errors that occur due to temporary interruptions in the service or to excess latency. You can use retries to handle these types of errors.
• **Queues**
You can implement queues to ensure that messages are received and processed in a specific order.

Module 2 Module Hybrid Networking

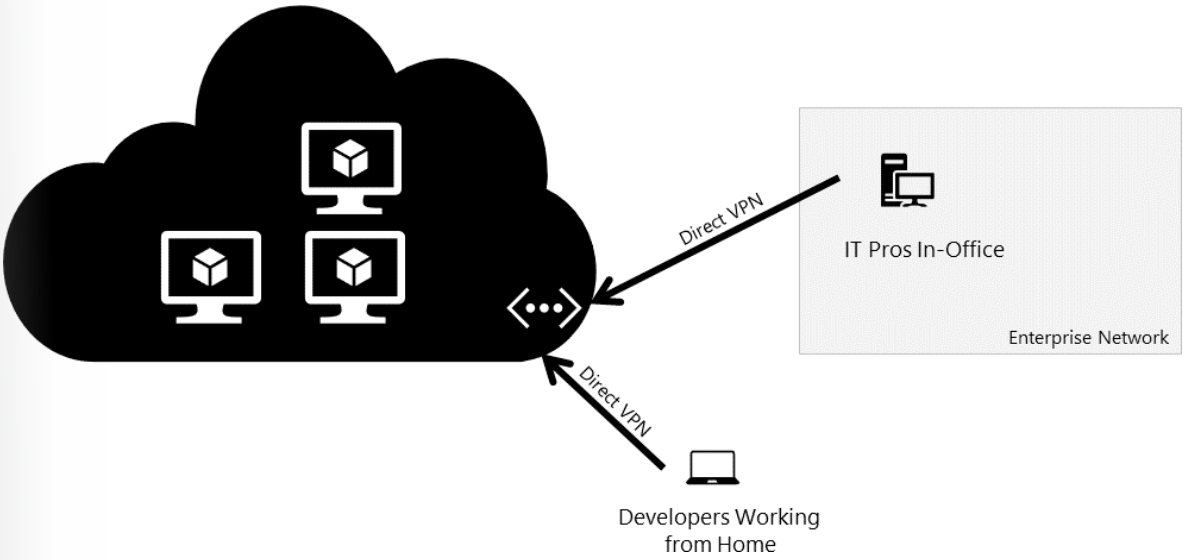
Hybrid Networking

Site-to-site connectivity



A site-to-site VPN allows you to create a security-enhanced connection between your on-premises site and your virtual network. To create a site-to-site connection, a VPN device that is located on your on-premises network is configured to create a security-enhanced connection with the Azure Virtual Network gateway. Once the connection is created, resources on your local network and resources located in your virtual network can directly and more-securely communicate. Site-to-site connections do not require you to establish a separate connection for each client computer on your local network to access resources in the virtual network.

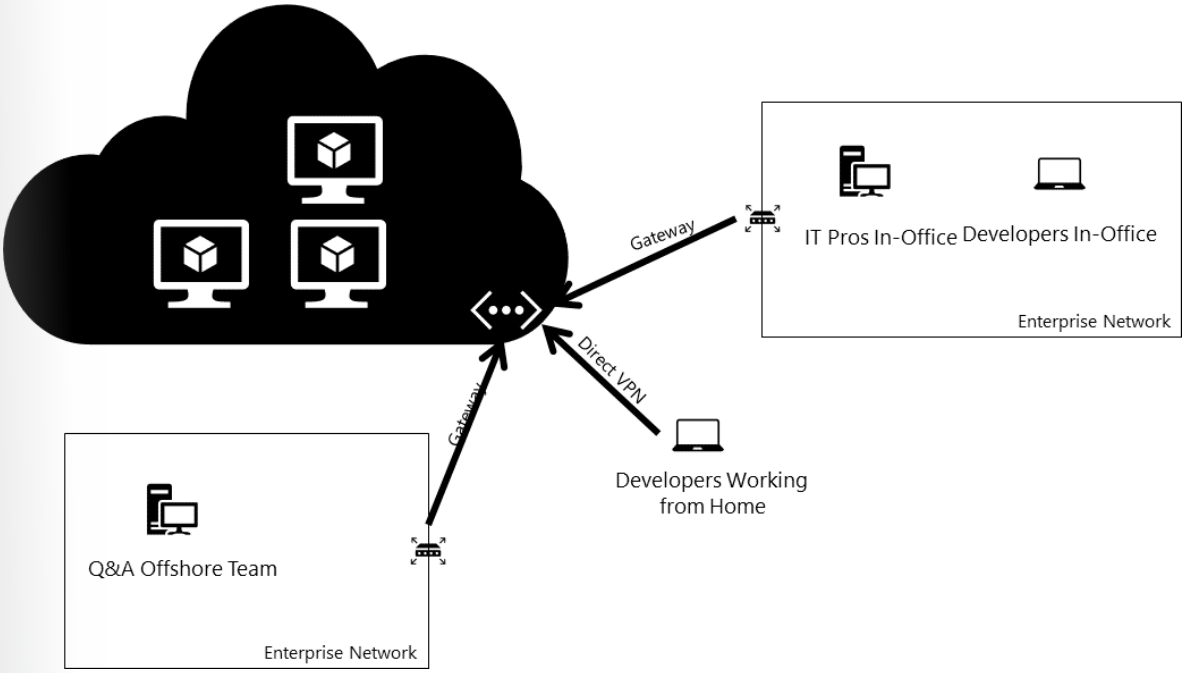
Point-to-site connectivity



A point-to-site VPN also allows you to create a security-enhanced connection to your virtual network. In a point-to-site configuration, the connection is configured individually on each client computer that you want to connect to the virtual network. Point-to-site connections do not require a VPN device. They work by using a VPN client that you install on each client computer. The VPN is established by manually starting the connection from the on-premises client computer. You can also configure the VPN client to automatically restart.

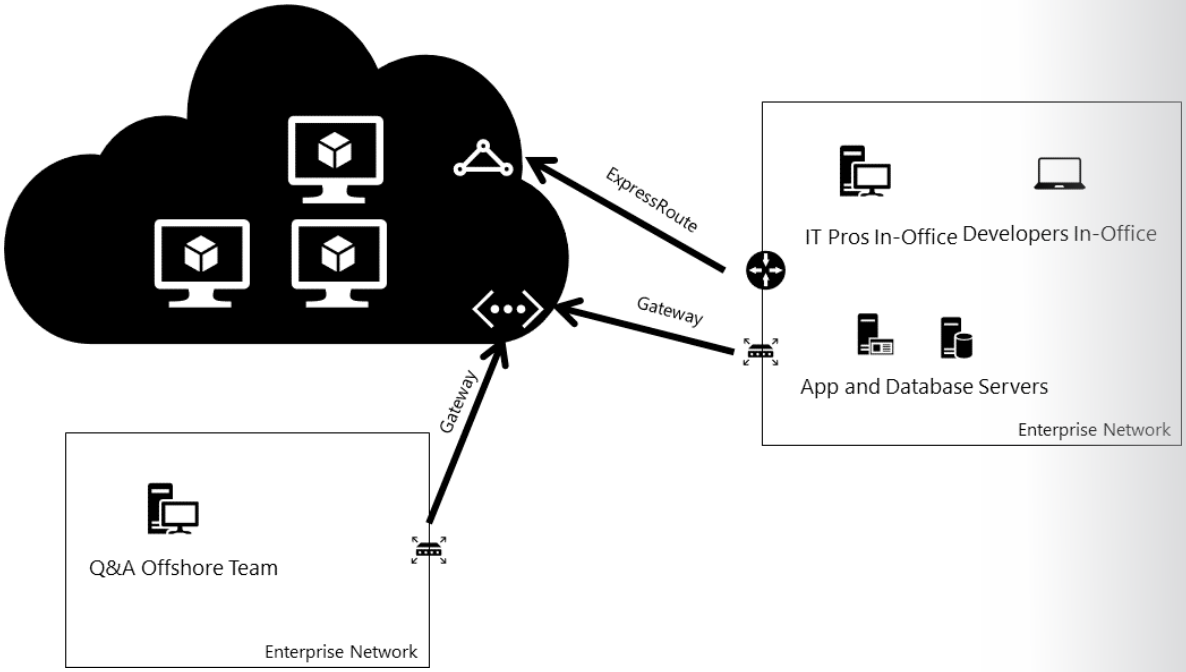
Note: Point-to-site and site-to-site configurations can exist concurrently.

Combining site-to-site and point-to-site connectivity



Site-to-site and point-to-site connections can be combined for a variety of reasons. In the following diagram, the enterprise has decided to use a site-to-site connection to connect the in-office networks to Azure. Developers who are working remotely can connect to the virtual networks directly using a point-to-site connection.

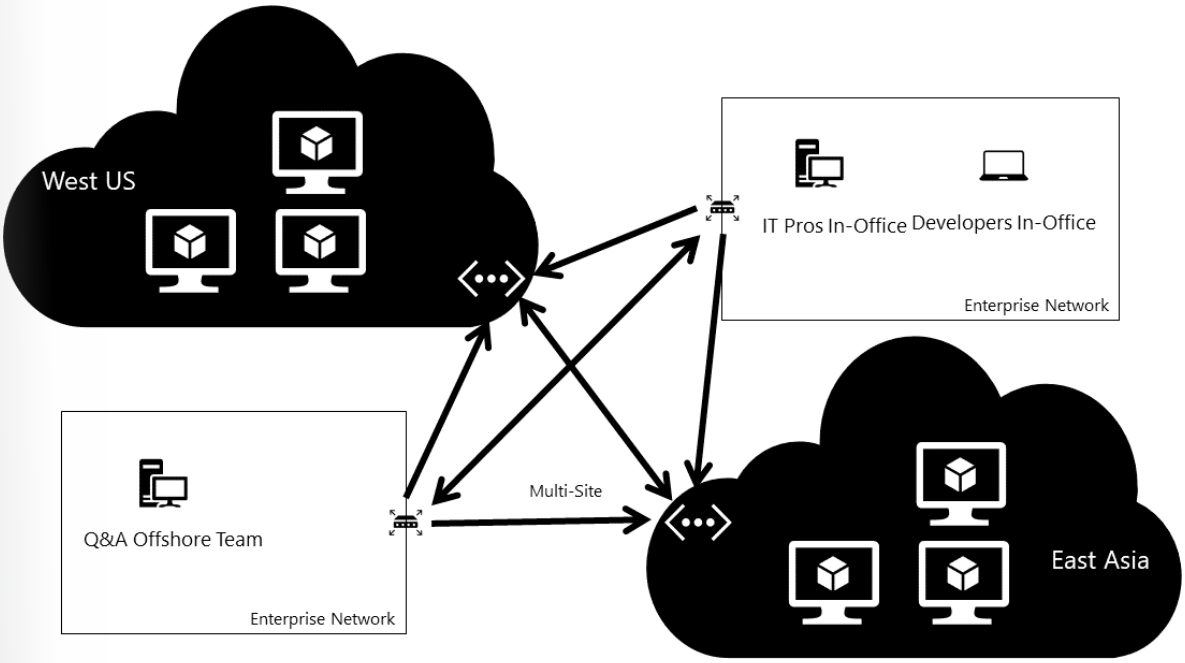
Combining ExpressRoute and site-to-site Connectivity



You can connect ExpressRoute and a site-to-site VPN on the same virtual network. There are many reasons you may want to do this:

- You may have multiple branch offices, and it would be cost prohibitive to purchase peering for every location. You can use a site-to-site VPN for the locations that don't require the fastest or most reliable connections.
- You may have multiple networks within your enterprise and may want to connect one to Azure using ExpressRoute and one to Azure using a site-to-site VPN so there are two active connections. The ExpressRoute connection could be used for higher-risk traffic.
- You can use the site-to-site VPN as a failover link if the ExpressRoute connection fails.

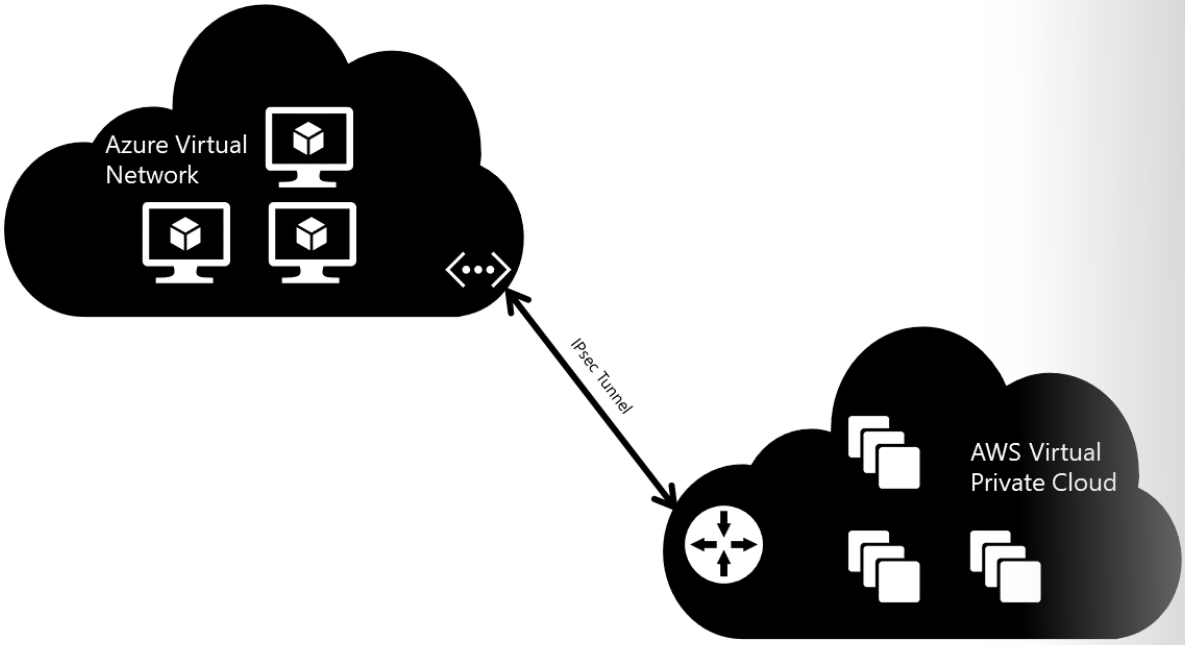
Virtual network–to–virtual network connectivity



Virtual network–to–virtual network connectivity utilizes the Azure VPN gateways to more-securely connect two or more virtual networks together with Internet Protocol security (IPsec) / Internet Key Exchange (IKE) S2S VPN tunnels. Together with the multi-site VPNs, you can connect your virtual networks and on-premises sites together in a topology that suits your business needs. The diagram in the following section shows a simple example of a fully connected topology between virtual networks and on-premises sites.

Connecting across cloud providers

Since a site-to-site connection is simply an IPsec tunnel, you can connect to networks across cloud providers. This scenario could be used for failover, backup, or even migration between providers.

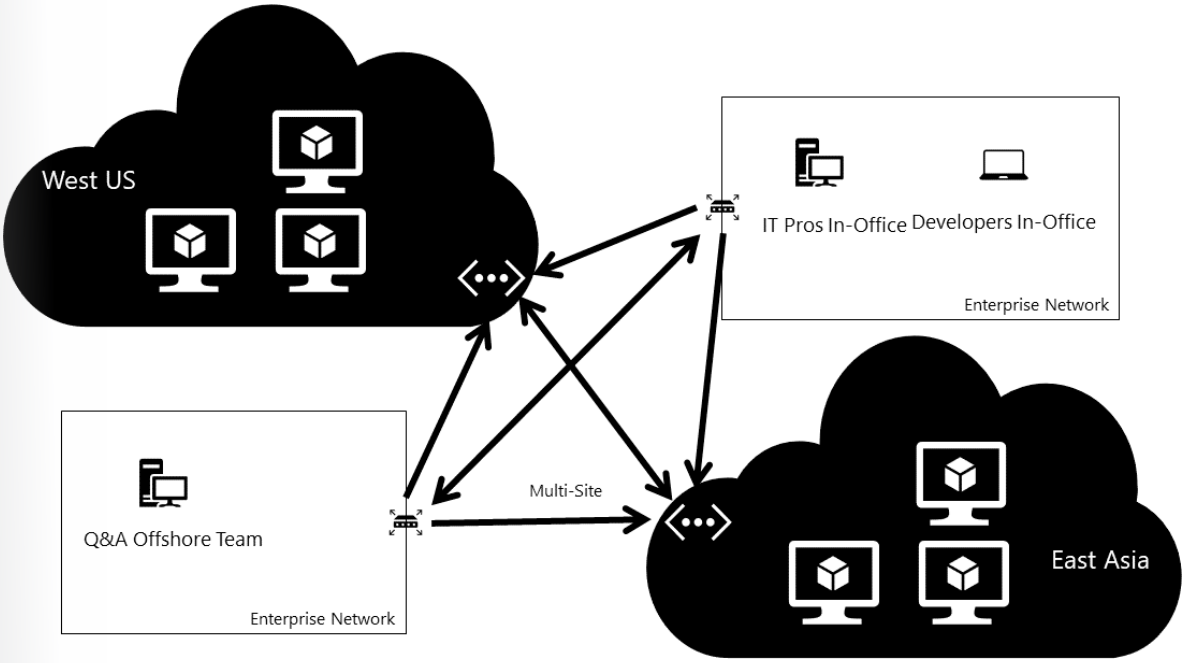


Amazon Web Services (AWS)

In AWS, you can create a virtual private cloud that provides network capabilities similar to those of a virtual network in Azure. An Amazon Elastic Compute Cloud (EC2) instance with Openswan (VPN software) can then be created for VPN functionality. After those instances are running, you simply create a gateway on the Azure virtual network side using static routing. The gateway IP address from Azure is then used to configure Openswan for a tunnel connection between the two virtual networks.

Virtual Network-to-Network

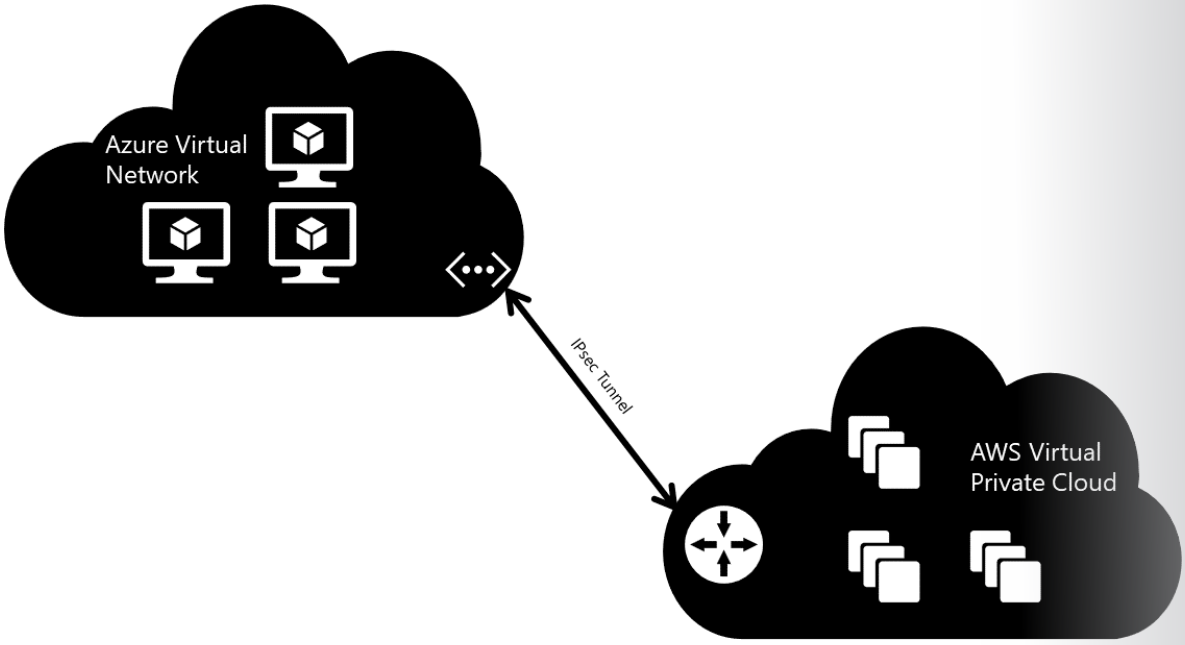
Virtual network–to–virtual network connectivity



Virtual network–to–virtual network connectivity utilizes the Azure VPN gateways to more-securely connect two or more virtual networks together with Internet Protocol security (IPsec) / Internet Key Exchange (IKE) S2S VPN tunnels. Together with the multi-site VPNs, you can connect your virtual networks and on-premises sites together in a topology that suits your business needs. The diagram in the following section shows a simple example of a fully connected topology between virtual networks and on-premises sites.

Connecting across cloud providers

Since a site-to-site connection is simply an IPsec tunnel, you can connect to networks across cloud providers. This scenario could be used for failover, backup, or even migration between providers.



Amazon Web Services (AWS)

In AWS, you can create a virtual private cloud that provides network capabilities similar to those of a virtual network in Azure. An Amazon Elastic Compute Cloud (EC2) instance with Openswan (VPN software) can then be created for VPN functionality. After those instances are running, you simply create a gateway on the Azure virtual network side using static routing. The gateway IP address from Azure is then used to configure Openswan for a tunnel connection between the two virtual networks.

Review Question

Module 2 Review Question

Combining site-to-site and point-to-site connectivity

An organization is developing a new application with the help of a consulting company. The consulting company is developing part of the solution in a remote location.

Management of your company is concerned to giving teams of external developers access to internal resources.

Which hybrid networking solution will minimize risk and maximize connectivity? Why might you choose one networking solution over another?

Suggested Answer ↓

Site-to-site and point-to-site connections can be combined for a variety of reasons. In this scenario, the enterprise has decided to use a site-to-site connection to connect the in-office networks to Azure, which can then be accessed by an off-shore development team, without exposing internal resources to the offshore team. Developers who are working remotely can connect to the virtual networks directly using a point-to-site connection.

Module 3 Module Measure Throughput and Structure of Data Access

Address Durability of Data and Caching

Data Concurrency

ACID

The acronym ACID stands for atomic, consistent, isolated, and durable. To ensure predictable behavior, all transactions must possess these basic properties, reinforcing the role of mission-critical transactions as all-or-none propositions:

- **Atomic:** A transaction must execute exactly once and must be atomic, meaning all work completes or none of it does. Operations within a transaction usually share a common intent and are interdependent. By performing only a subset of these operations, the system could compromise the overall intent of the transaction. Atomicity eliminates the chance of processing only a subset of operations.
- **Consistent:** A transaction must preserve the consistency of data, transforming one consistent state of data into another consistent state of data. Typically, the application developer is responsible for maintaining consistency.
- **Isolated:** A transaction must be a unit of isolation, which means that concurrent transactions should behave as if each were the only transaction running in the system. Because a high degree of isolation can limit the number of concurrent transactions, some applications reduce the isolation level in exchange for better throughput.
- **Durable:** A transaction must be recoverable and therefore must have durability. If a transaction commits, the system guarantees that its updates can persist even if the computer crashes immediately after the commit. Specialized logging allows the system's restart procedure to complete unfinished operations required by the transaction, making the transaction durable.

A *transaction* in a database system is a set of operations, which are related, that seek to achieve some or all the ACID properties. In most relational database management systems (RDBMS), a transaction is a single unit of work. If a transaction is successful, all of the data modifications made during the transaction are committed and become a permanent part of the database. The database system erases all data

modifications based on that transaction if a transaction encounters any errors or must be rolled back for another reason.

Caching in distributed applications

Caching is a common technique that aims to improve the performance and scalability of a system. It does this by temporarily copying frequently accessed data to fast storage that's located close to the application. If this fast data storage is located closer to the application than the original source, then caching can significantly improve response times for client applications by serving data more quickly.

Caching is most effective when a client instance repeatedly reads the same data, especially if all the following conditions apply to the original data store:

- It remains relatively static.
- It's slow compared to the cache's speed.
- It's subject to a significant level of contention.
- It's far away when network latency can cause access to be slow.

Distributed applications typically implement either or both of the following strategies when caching data:

- Using a private cache, where data is held locally on the computer that's running an instance of an application or service.
- Using a shared cache, serving as a common source which multiple processes and/or machines can access.

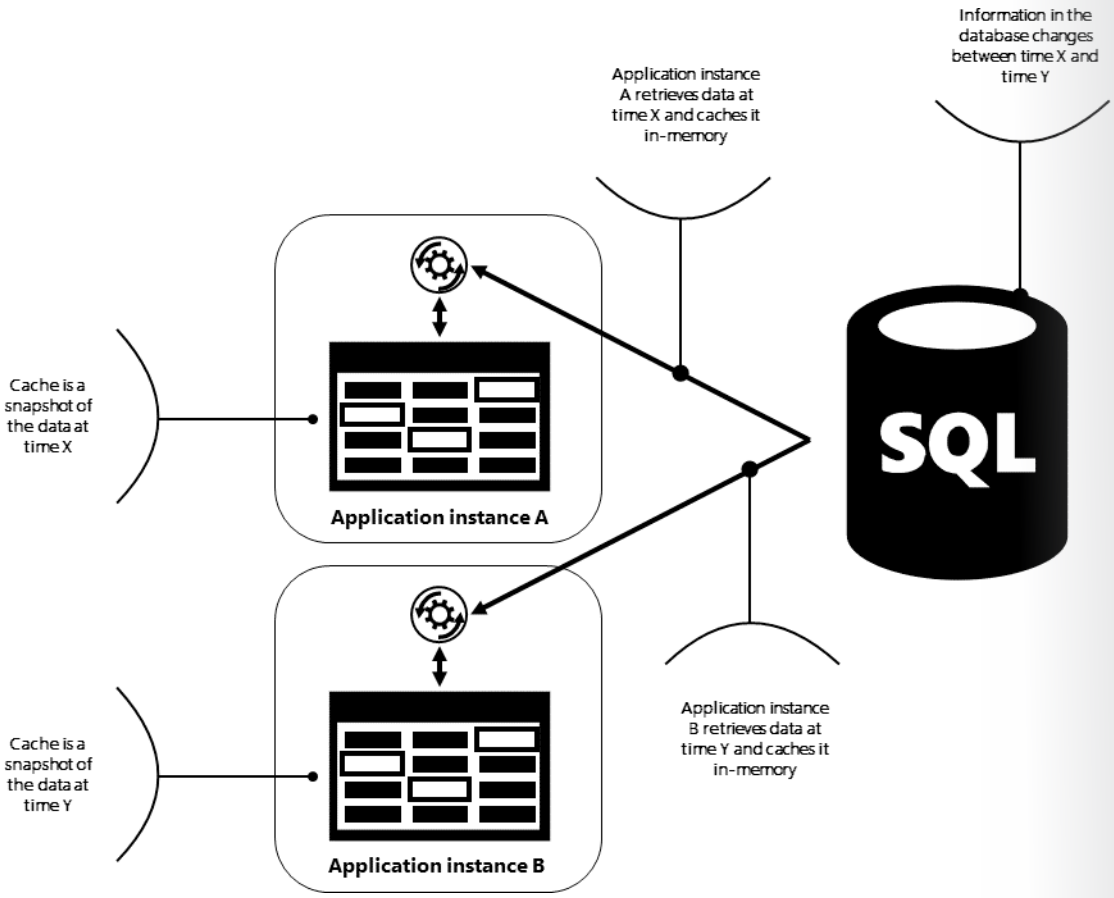
In both cases, caching can occur on the client-side and the server-side. The process that provides the user interface for a system, such as a web browser or desktop application, performs client-side caching, while the process that provides the business services that are running remotely performs the server-side caching.

Private caching

The most basic type of cache is an in-memory store. It's held in the address space of a single process and accessed directly by the code that runs in that process. This type of cache is very quick to access. It can also provide an extremely effective means for storing modest amounts of static data, since the size of a cache is typically constrained by the volume of memory that's available on the machine hosting the process.

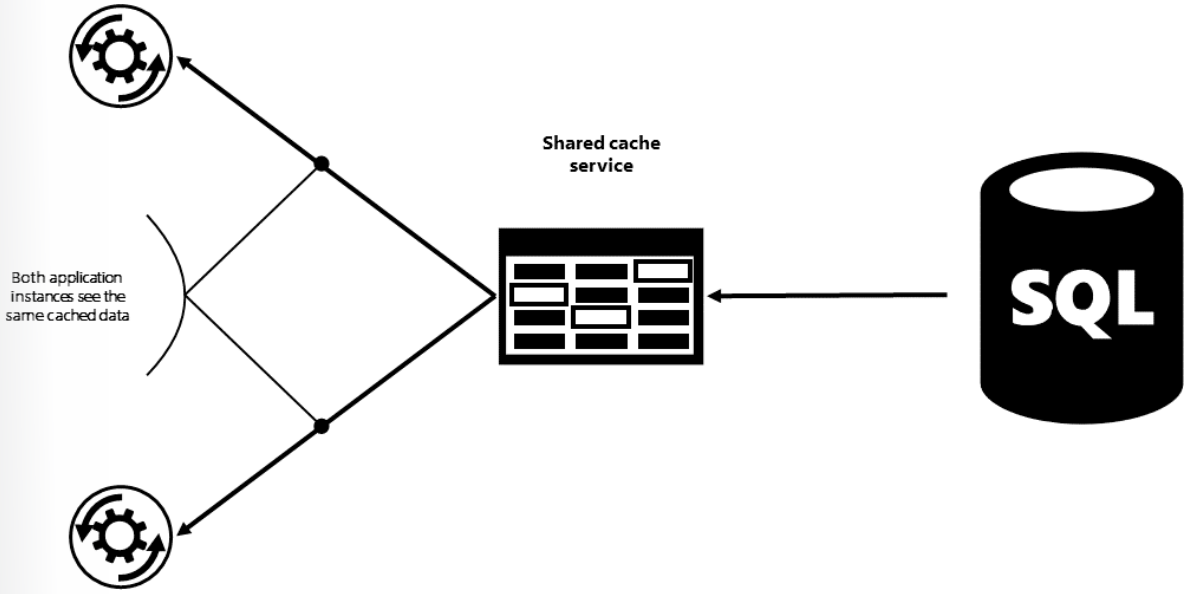
If you need to cache more information than is physically possible in memory, you can write cached data to the local file system. This will be slower to access than data that's held in-memory but should still be faster and more dependable than retrieving data across a network. If you have multiple instances of an application that uses this model running concurrently, each application instance has its own independent cache holding its own copy of the data.

Think of a cache as a snapshot of the original data at a point in the past. If this data is not static, it is likely that different application instances hold different versions of the data in their caches. Therefore, the same query performed by these instances can return different results.



Shared caching

Using a shared cache can help alleviate concerns that data might differ in each cache, which can occur with in-memory caching. Shared caching ensures that different application instances see the same view of cached data. It does this by locating the cache in a separate location, typically hosted as part of a separate service.



An important benefit of the shared caching approach is the scalability it provides. Many shared cache services are implemented by using a cluster of servers and utilize software that distributes the data across the cluster in a transparent manner. An application instance simply sends a request to the cache service. The underlying infrastructure is responsible for determining the location of the cached data in the cluster. You can easily scale the cache by adding more servers.

There are two main disadvantages of the shared caching approach:

- The cache is slower to access because it isn't held locally to each application instance.
- The requirement to implement a separate cache service might add complexity to the solution.

Caching considerations

When to cache data

Caching can dramatically improve performance, scalability, and availability. The more data that you have and the larger the number of users that need to access this data, the greater the benefits of caching become. That's because caching reduces the latency and contention that's associated with handling large volumes of concurrent requests in the original data store.

For example, a database might support a limited number of concurrent connections. Retrieving data from a shared cache, however, rather than the underlying database, makes it possible for a client application to access this data even if the number of available connections is currently exhausted. Additionally, if the database becomes unavailable, client applications might be able to continue by using the data that's held in the cache.

How to cache data effectively

The key to using a cache effectively lies in determining the most appropriate data to cache and caching it at the appropriate time. You can add the data to the cache on demand the first time it is retrieved by an application. This means that the application needs to fetch the data only once from the data store, and that subsequent access can be satisfied by using the cache.

Alternatively, a cache can be partially or fully populated with data in advance, typically when the application starts (an approach known as seeding). However, it might not be advisable to implement seeding for a large cache because this approach can impose a sudden, high load on the original data store when the application starts running. Caching typically works well with data that is immutable or that changes infrequently.

Manage data expiration in a cache

In most cases, data that's held in a cache is a copy of data that's held in the original data store. The data in the original data store might change after it was cached, causing the cached data to become stale. Many caching systems enable you to configure the cache to expire data and reduce the period for which data may be out of date.

When cached data expires, it's removed from the cache, and the application must retrieve the data from the original data store (it can put the newly-fetched information back into cache). You can set a default expiration policy when you configure the cache. In many cache services, you can also stipulate the expiration period for individual objects when you store them programmatically in the cache.

Redis Cache

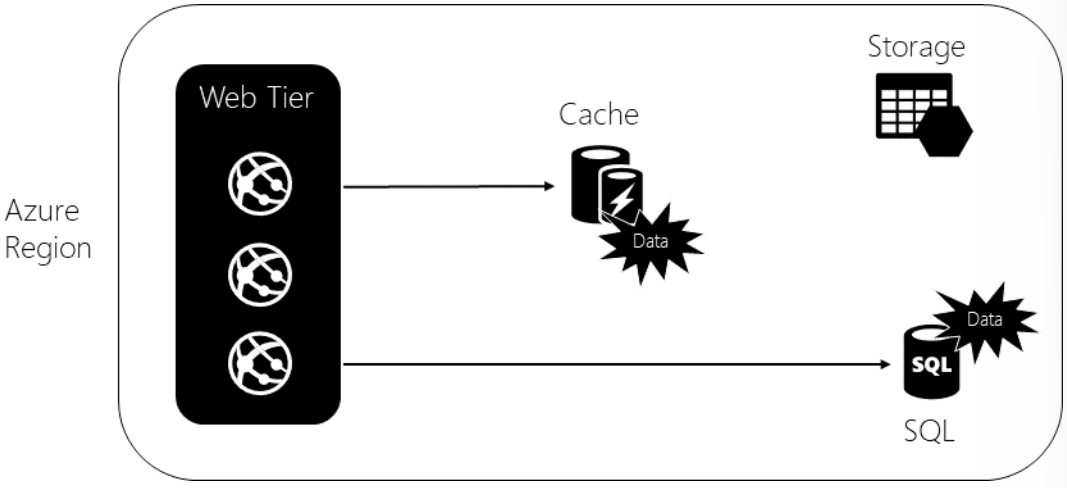
Note: There are two primary cache mechanisms available in Azure–Azure Cache and Azure Redis Cache. Azure Cache is deprecated and only exists to support existing cloud applications. All new applications should use the Redis Cache.

Redis Cache is an open-source *not only SQL* (NoSQL) storage mechanism that is implemented in the key-value pair pattern common among other NoSQL stores. Redis Cache is unique because it allows complex data structures for its keys.

Azure Redis Cache is a managed service based on Redis Cache that provides you secure nodes as a service. There are only two tiers for this service currently available:

- **Basic:** Includes a single node.
- **Standard:** Includes two nodes in the Primary/Replica configuration and also includes replication support and a Service Level Agreement (SLA).

Azure Redis Cache provides a high degree of compatibility with existing tools and applications that already integrate with Redis Cache. You can use the Redis Cache documentation that already exists on the open source community for Azure Redis Cache.



Redis Cache Console

Measure Throughput and Structure of Data Access

Normalized Units

In a world of hyperscale database services, it can be difficult to determine how much performance you need or how powerful an allocated database is. To help ease this challenge, many cloud vendors have provided normalized units of measurements that can be used to compare database tiers. Sometimes these units of measurement have a direct relation to on-premises database equivalents, but it is simpler to think of them as relative performance guarantees.

For example, if your application uses 20 database units today, 40 database units will guarantee you approximately double your performance, while 10 database units will guarantee you half of your performance.

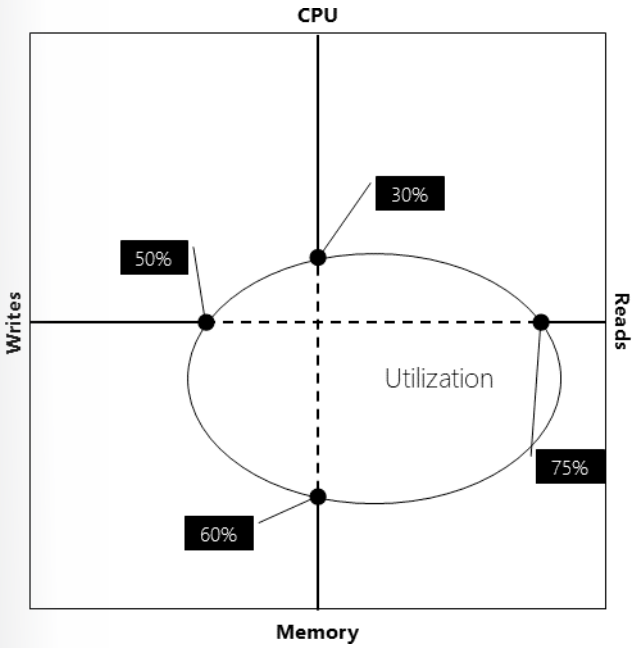
Let's look at a few examples of normalized units in Azure and examine how you can use them to compare database service tiers.

DTUs – Azure SQL Database

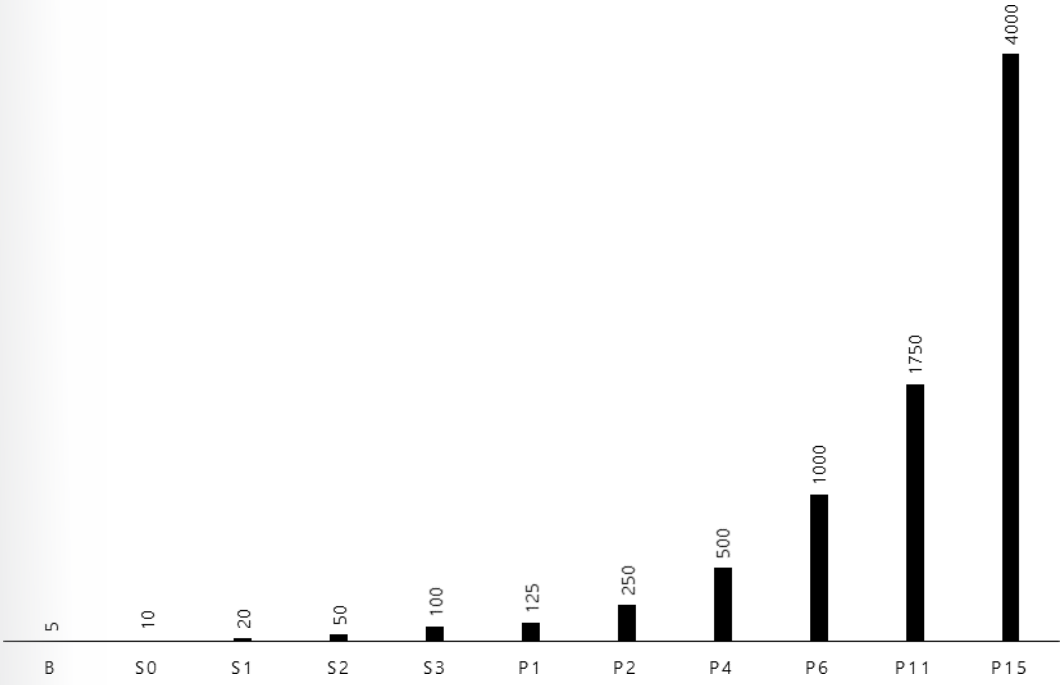
In Azure SQL Database, we measure database performance in terms of **database throughput units (DTUs)**. The DTU model is based on a bundled measure of compute, storage, and IO resources. Performance levels are expressed in terms of database transaction units (DTUs) for single databases and elastic database transaction units (eDTUs) for elastic pools.

DTUs describe the capacity for a specific tier and performance level, and they are designed to be relative so that you can directly compare the tiers and performance levels. For example, the Basic tier has a single performance level (B) that is rated at 5 DTU. The S2 performance level in the Standard tier is rated at 50 DTU. This means that you can expect ten times the power for a database at the S2 performance level than a database at the B performance level in the Basic tier.

The easiest way to visualize a DTU is to think about it in the context of a bounding box. The box represents the relative power (or resources) assigned to the database. This relative power is a natural blended measurement of the central processing unit (CPU), memory, and read-write performance:



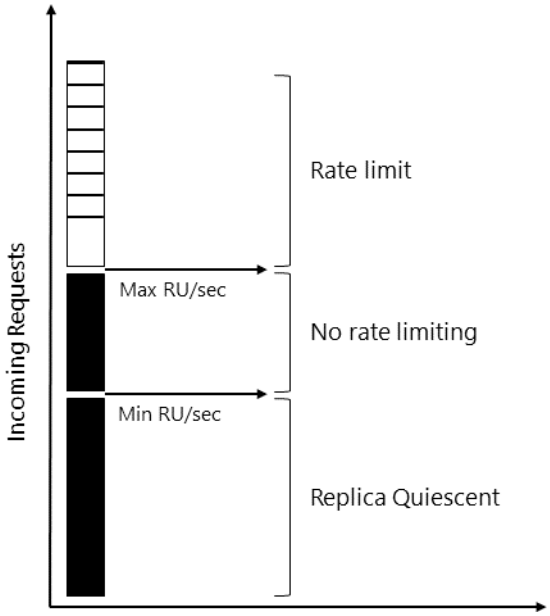
Every tier has one or more performance levels. In general, the performance levels in the Premium tier have a higher rating than the performance levels in the Standard tier, which have a higher rating than those in the Basic tier. The following chart illustrates this distinction. Service tiers are differentiated by a range of performance levels with a fixed amount of included storage, fixed retention period for backups, and fixed price. All service tiers provide flexibility of changing performance levels without downtime.



RUs – Azure Cosmos DB

Azure Cosmos DB reserves resources to manage the throughput of an application. Because, application load and access patterns change over time, Azure Cosmos DB has support built-in to increase or decrease the amount of reserved throughput available at any time.

With Azure Cosmos DB, reserved throughput is specified in terms of **request unit processing per second (RU/s)**. You reserve several guaranteed request units to be available to your application on a per-second basis. Each operation in Azure Cosmos DB, including writing a document, performing a query, and updating a document, consumes CPU, memory, and Input/output operations per second (IOPS). That is, each operation incurs a request charge, which is expressed in request units.



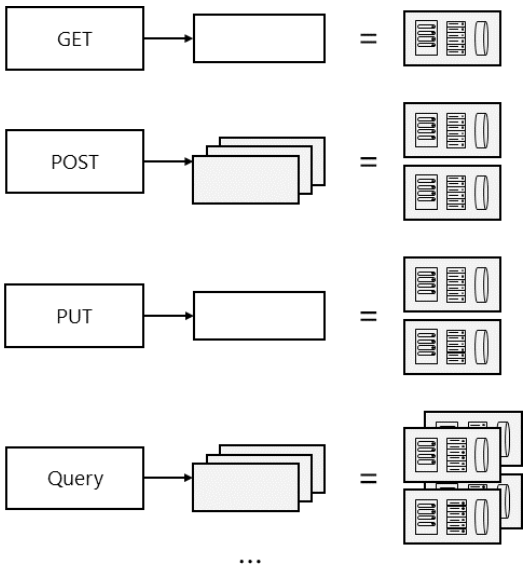
A request unit is a normalized measure of request processing cost. A single request unit represents the processing capacity that's required to read, via self-link or ID, a single item that is 1 kilobyte (KB) and that consists of 10 unique property values (excluding system properties). A request to create (insert), replace, or delete the same item consumes more processing from the service and thereby requires more request units.

Normalized across various access methods

1 RU = 1 read of 1 KB document

Each request consumes fixed RUs

Applies to reads, writes, queries, and stored procedure execution



Structured and Unstructured Data

Modern business systems manage increasingly large volumes of data, typically ingesting data from external services that the system generates or that users create. These data sets may have extremely varied characteristics and processing requirements. Businesses use data to assess trends, trigger business processes, audit their operations, analyze customer behavior, and other factors.

This heterogeneity means that a single data store is usually not the best approach. Instead, it's often better to store diverse types of data in different data stores, each focused towards a specific workload or usage pattern. The term polyglot persistence describes solutions that use a mix of data store technologies.

Selecting the right data store for your requirements is a key design decision. There are hundreds of implementations to choose from among SQL and NoSQL databases. Data stores typically are categorized by how they structure data and the types of operations they support. This article describes several common storage models.

Using structured data stores

Relational databases organize data as a series of two-dimensional tables with rows and columns. Each table has its own columns, and every row in a table has the same set of columns. This model is mathematically based, and most vendors provide a dialect of the Structured Query Language (SQL) for retrieving and managing data. An RDBMS typically implements a transactionally consistent mechanism that conforms to the ACID (Atomic, Consistent, Isolated, Durable) model for updating information.

An RDBMS typically supports a schema-on-write model, where you define the data structure and then all read or write operations use the schema. An RDBMS is especially useful when strong consistency guarantees are important — where all changes are atomic, and transactions always leave the data in a consistent state. However, the underlying structures do not lend themselves to scaling out by distributing storage and processing across machines.

Structured data stores in Azure include:

- Azure SQL Database
- Azure Database for MySQL

- Azure Database for PostgreSQL

Using Unstructured or semi-structured data stores

A non-relational database doesn't use the tabular schema of rows and columns that most traditional database systems use. Rather, non-relational databases utilize an optimized storage model that is based on specific requirements of the type of data it's story. For example, a non-relational database might store date as simple key/value pairs, as JSON documents, or as a graph consisting of edges and vertices.

What all of these data stores have in common is that they don't use a relational model. Also, they tend to be more specific in the type of data they support and how you can query that data. For example, time series data stores are optimized for queries over time-based sequences of data, while graph data stores are optimized for exploring weighted relationships between entities. Neither format would generalize well to the task of managing transactional data.

The term *NoSQL* refers to data stores that do not use SQL for queries, and instead use other programming languages and constructs to query the data. In practice, "NoSQL" means "non-relational database," even though several of these databases do support SQL-compatible queries. However, the underlying query execution strategy is usually quite different from the way a traditional RDBMS would execute the same SQL query. There are several types of NoSQL data stores, and we'll detail the most common in the next sections of this article.

Document databases

A document database is conceptually similar to a key/value store, except that it stores a collection of named fields and data (known as documents), each of which could be simple scalar items or compound elements such as lists and child collections. There are several ways in which you can encode the data in a document's fields, including using Extensible Markup Language (XML), YAML, JavaScript Object Notation (JSON), Binary JSON (BSON), or even storing it as plain text. Unlike key/value stores, the fields in documents are exposed to the storage management system, enabling an application to query and filter data by using the values in these fields.

Typically, a document contains the entire data for an entity. What items constitute an entity are application specific. For example, an entity could contain the details of a customer, an order, or a combination of both. A single document may contain information that would be spread across several relational tables in an RDBMS.

Key	Document

1001	[
	  {
	  "customerId": 344,
	  "orderItems": [
	  {
	  "productId": 4524,
	  "quantity": 1,
	  "price": 125.67
	  },
	  {
	  "productId": 3311,
	  "quantity": 4,
	  "price": 73.06
	  }
	  ],
	  "orderDate": "2017-10-18T12:27:30 +04:00"
	  }
	]

1002	[
	  {
	  "customerId": 263,
	  "orderItems": [
	  {
	  "productId": 4076,
	  "quantity": 3,
	  "price": 257.64
	  }
	  ],
	  "orderDate": "2014-01-31T02:09:02 +05:00"
	  }
	]

1003	[
	  {
	  "customerId": 308,
	  "orderItems": [
	  {
	  "productId": 1957,
	  "quantity": 1,
	  "price": 279.63
	  }
	  ],
	  "orderDate": "2016-09-18T01:33:53 +04:00"
	  }
	]

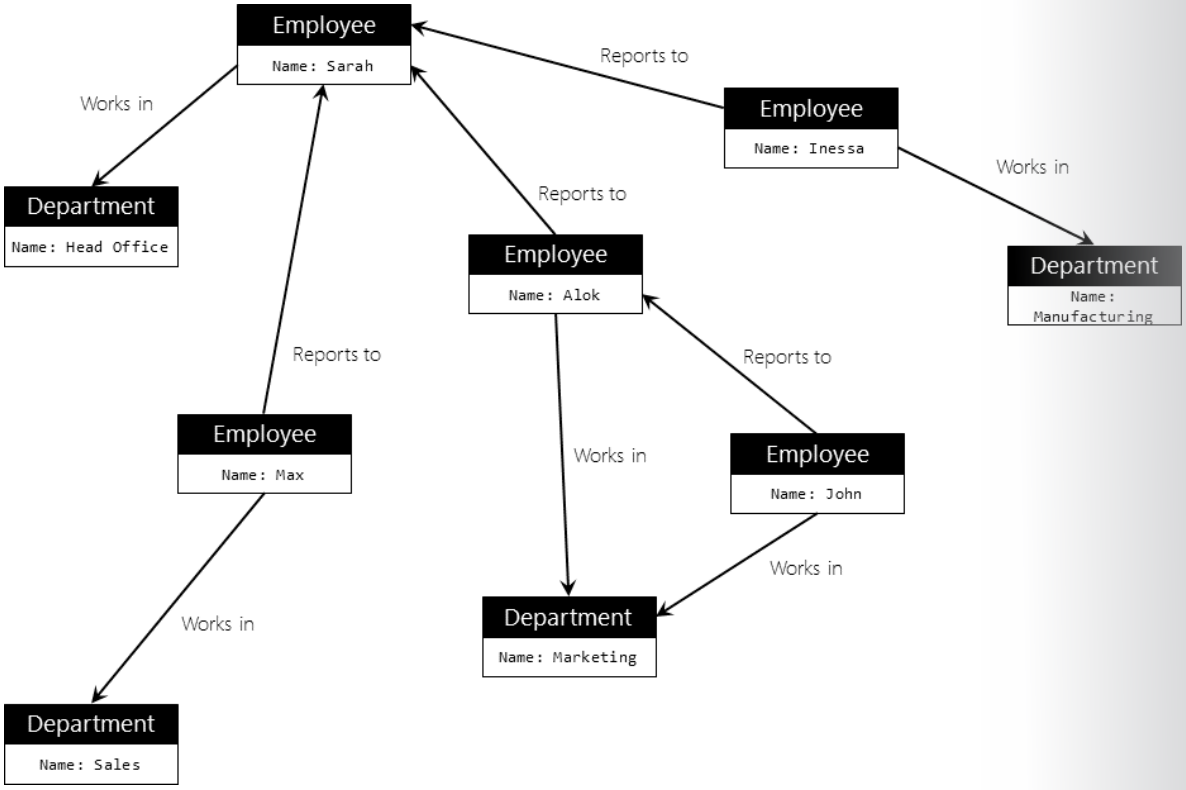
Document stores in Azure include:

- Azure Cosmos DB

Graph databases

A graph database stores two types of information, nodes and edges. You can think of nodes as entities. Edges which specify the relationships between nodes. Both nodes and edges can have properties that provide information about that node or edge, similar to columns in a table. Edges can also have a direction indicating the nature of the relationship.

The purpose of a graph database is to allow an application to efficiently perform queries that traverse the network of nodes and edges, and to analyze the relationships between entities. The following diagram shows an organization's personnel database structured as a graph. The entities are employees and departments, and the edges indicate reporting relationships and the department in which employees work. In this graph, the arrows on the edges show the direction of the relationships.



Graph stores in Azure include:

- Azure Cosmos DB

Online Lab - Implementing Azure Load Balancer Standard

Lab Steps

Online Lab: Implementing Azure Load Balancer Standard

NOTE: For the most recent version of this online lab, see: **https://github.com/MicrosoftLearning/AZ-300-MicrosoftAzureArchitectTechnologies**

Scenario

Adatum Corporation wants to implement Azure Load Balancer Standard to direct inbound and outbound traffic of Azure VMs.

Objectives

After completing this lab, you will be able to:

- Implement inbound load balancing by using Azure Load Balancer Standard
- Configure outbound SNAT traffic by using Azure Load Balancer Standard

Lab Setup

Estimated Time: 45 minutes

User Name: **Student**

Password: **Pa55w.rd**

Exercise 1: Implement inbound load balancing and NAT by using Azure Load Balancer Standard

The main tasks for this exercise are as follows:

1. Deploy Azure VMs in an availability set by using an Azure Resource Manager template
2. Create an instance of Azure Load Balancer Standard
3. Create a load balancing rule of Azure Load Balancer Standard
4. Create a NAT rule of Azure Load Balancer Standard
5. Test functionality of Azure Load Balancer Standard

Task 1: Deploy Azure VMs in an availability set by using an Azure Resource Manager template

1. From the lab virtual machine, start Microsoft Edge and browse to the Azure portal at **http://portal.azure.com** and sign in by using the Microsoft account that has the Owner role in the target Azure subscription.
2. In the Azure portal, in the Microsoft Edge window, start a **Bash** session within the **Cloud Shell**.
3. If you are presented with the **You have no storage mounted** message, configure storage using the following settings:
   - Subsciption: the name of the target Azure subscription
   - Cloud Shell region: the name of the Azure region that is available in your subscription and which is closest to the lab location
   - Resource group: the name of a new resource group **az3000800-LabRG**
   - Storage account: a name of a new storage account
   - File share: a name of a new file share
4. From the Cloud Shell pane, create a resource groups by running (replace the `<Azure region>` placeholder with the name of the Azure region that is available in your subscription and which is closest to the lab location)

   ```
   az group create --name az3000801-LabRG --location <Azure region>
   ```

5. From the Cloud Shell pane, upload the Azure Resource Manager template **\allfiles\AZ-300T03\Module_03\azuredeploy0801.json** into the home directory.
6. From the Cloud Shell pane, upload the parameter file **\allfiles\AZ-300T03\Module_03\azuredeploy0801.parameters.json** into the home directory.
7. From the Cloud Shell pane, deploy a pair of Azure VMs hosting Windows Server 2016 Datacenter by running:

   ```
   az group deployment create --resource-group az3000801-LabRG --template-file
   azuredeploy0801.json --parameters @azuredeploy0801.parameters.json
   ```

8. **Note**: Wait for the deployment before you proceed to the next task. This might take about 10 minutes.
9. In the Azure portal, close the Cloud Shell pane.

Task 2: Create an instance of Azure Load Balancer Standard

1. In the Azure portal, create a new Azure Load Balancer with the following settings:
   - Subsciption: the name of the target Azure subscription
   - Resource group: **az3000801-LabRG**

- Name: **az3000801-lb**
- Region: the name of the Azure region in which you deployed Azure VMs in the previous task of this exercise
- Type: **Public**
- SKU: **Standard**
- Public IP address: **Create new** named **az3000801-lb-pip01**
- Availability zone: **Zone-redundant**

Task 3: Create a load balancing rule of Azure Load Balancer Standard

1. In the Azure portal, navigate to the blade displaying the properties of the newly deployed Azure Load Balancer **az3000801-lb**.
2. On the **az3000801-lb** blade, click **Backend pools**.
3. On the **az3000801-lb - Backend pools** blade, click **+ Add**.
4. On the **Add backend pool** blade, specify the following settings and click **Add**:
   - Name: **az3000801-bepool**
   - Virtual network: **az3000801-vnet (2 VM)**
   - VIRTUAL MACHINE: **az3000801-vm0** IP ADDRESS: **ipconfig1 (10.0.0.4)** or **ipconfig1 (10.0.0.5)**
   - VIRTUAL MACHINE: **az3000801-vm1** IP ADDRESS: **ipconfig1 (10.0.0.5)** or **ipconfig1 (10.0.0.4)**
5. **Note**: It is possible that the IP addresses of virtual machines are asssigned in the reversed order.
6. **Note**: Wait for the operation to complete. This should not take more than 1 minute.
7. Back on the **az3000801-lb - Backend pools** blade, click **Health probes**.
8. On the **az3000801-lb - Health probes** blade, click **+ Add**.
9. On the **Add health probe** blade, specify the following settings and click **OK**:
   - Name: **az3000801-healthprobe**
   - Protocol: **TCP**
   - Port: **80**
   - Interval: **5**
   - Unhealthy threshold: **2**
10. **Note**: Wait for the operation to complete. This should not take more than 1 minute.
11. Back on the **az3000801-lb - Health probes** blade, click **Load balancing rules**.
12. On the **az3000801-lb - Load balancing rules** blade, click **+ Add**.
13. On the **Add load balancing rule** blade, specify the following settings and click **OK**:
    - Name: **az3000801-lbrule01**
    - IP Version: **IPv4**

- Frontend IP address: select the public IP address assigned to the **LoadBalancedFrontEnd** from the drop-down list
- Protocol: **TCP**
- Port: **80**
- Backend port: **80**
- Backend pool: **az3000801-bepool (2 virtual machines)**
- Health probe: **az3000801-healthprobe (TCP:80)**
- Session persistence: **None**
- Idle timeout (minutes): **4**
- Floating IP (direct server return): **Disabled**

14. **Note**: Wait for the operation to complete. This should not take more than 1 minute.

Task 4: Create a NAT rule of Azure Load Balancer Standard

1. In the Azure portal, on the **az3000801-lb** blade, click **Inbound NAT rules**.
2. On the **az3000801-lb - Inbound NAT rules** blade, click **+ Add**.
3. On the **Add inbound NAT rule** blade, specify the following settings and click **OK**:
   - Name: **az3000801-vm0-RDP**
   - Frontend IP address: select the public IP address assigned to the **LoadBalancedFrontEnd** from the drop-down list
   - IP Version: **IPv4**
   - Service: **RDP**
   - Protocol: **TCP**
   - Port: **33890**
   - Target virtual machine: **az3000801-vm0**
   - Network IP configuration: **ipconfig1 (10.0.0.4)** or **ipconfig1 (10.0.0.5)**
   - Port mapping: **Custom**
   - Floating IP (direct server return): **Disabled**
   - Target port: **3389**
4. **Note**: Wait for the operation to complete. This should not take more than 1 minute.
5. Back on the **az3000801-lb - Inbound NAT rules** blade, click **+ Add**.
6. On the **Add inbound NAT rule** blade, specify the following settings and click **OK**:
   - Name: **az3000801-vm1-RDP**
   - Frontend IP address: select the public IP address assigned to the **LoadBalancedFrontEnd** from the drop-down list
   - IP Version: **IPv4**

- Service: **RDP**
- Protocol: **TCP**
- Port: **33891**
- Target virtual machine: **az3000801-vm1**
- Network IP configuration: **ipconfig1 (10.0.0.5)** or **ipconfig1 (10.0.0.4)**
- Port mapping: **Custom**
- Floating IP (direct server return): **Disabled**
- Target port: **3389**

7. **Note**: Wait for the operation to complete. This should not take more than 1 minute.

Task 5: Test functionality of Azure Load Balancer Standard

1. In the Azure portal, navigate to the **az3000801-lb** blade and note the value of the **Public IP address** entry.
2. On the lab computer, start Microsoft Edge and navigate to the IP address you identified in the previous step.
3. Verify that you are presented with the default **Internet Information Services Welcome** page.
4. On the lab computer, right-click **Start**, click **Run**, and, from the **Open** text box, run the following (replace the `<IP address>` placeholder with the public IP address you identified earlier in this task):

   ```
   mstsc /v:<IP address>:33890
   ```

5. When prompted, authenticate by specifying the following values:
   - User name: **Student**
   - Password: **Pa55w.rd1234**
6. Within the Remote Desktop session, switch to the **Local Server** view in the Server Manager window and verify that you are connected to **az3000801-vm0** Azure VM.
7. Switch to the lab computer, right-click **Start**, click **Run**, and, from the **Open** text box, run the following (replace the `<IP address>` placeholder with the IP address you identified earlier in this task):

   ```
   mstsc /v:<IP address>:33891
   ```

8. When prompted, authenticate by specifying the following values:
   - User name: **Student**
   - Password: **Pa55w.rd1234**
9. Within the Remote Desktop session, switch to the **Local Server** view in the Server Manager window and verify that you are connected to **az3000801-vm1** Azure VM.
10. Within the Remote Desktop session, start a Windows PowerShell session and run the following to determine your current public IP address:

```
Invoke-RestMethod http://ipinfo.io/json
```

11. Review the output of the cmdlet and verify that the IP address entry matches the public IP address you identified earlier in this task.
12. Leave the Remote Desktop sessions open. You will use them in the next exercise.

Result: After you completed this exercise, you have implemented and tested Azure Load Balancer Standard inbound load balancing and NAT rules

Exercise 2: Configure outbound SNAT traffic by using Azure Load Balancer Standard

The main tasks for this exercise are as follows:

1. Deploy Azure VMs into an existing virtual network by using an Azure Resource Manager template
2. Create an Azure Standard Load Balancer and configure outbound SNAT rules
3. Test outbound rules of Azure Standard Load Balancer

Task 1: Deploy Azure VMs into an existing virtual network by using an Azure Resource Manager template

1. From the lab virtual machine, start Microsoft Edge and browse to the Azure portal at **http://portal.azure.com** and sign in by using the Microsoft account that has the Owner role in the target Azure subscription.
2. In the Azure portal, in the Microsoft Edge window, start a **Bash** session within the **Cloud Shell**.
3. From the Cloud Shell pane, upload the Azure Resource Manager template **\allfiles\AZ-300T03\Module_03\azuredeploy0802.json** into the home directory.
4. From the Cloud Shell pane, upload the parameter file **\allfiles\AZ-300T03\Module_03\azuredeploy0802.parameters.json** into the home directory.
5. From the Cloud Shell pane, deploy a pair of Azure VMs hosting Windows Server 2016 Datacenter by running:

```
az group deployment create --resource-group az3000801-LabRG --template-file
azuredeploy0802.json --parameters @azuredeploy0802.parameters.json
```

6. **Note**: Wait for the deployment before you proceed to the next task. This might take about 5 minutes.
7. In the Azure portal, close the Cloud Shell pane.

Task 2: Create an Azure Standard Load Balancer and configure outbound SNAT rules

1. In the Azure portal, in the Microsoft Edge window, start a **Bash** session within the **Cloud Shell**.
2. In the Azure portal, from the Cloud Shell pane, run the following to create an outbound public IP address of the load balancer:

```
az network public-ip create --resource-group az3000801-LabRG --name
az3000802-lb-pip01 --sku standard
```

3. In the Azure portal, from the Cloud Shell pane, run the following to create an Azure Load Balancer Standard:

```
LOCATION=$(az group show --name az3000801-LabRG --query location --out tsv)
az network lb create --resource-group az3000801-LabRG --name az3000802-lb
--sku standard --backend-pool-name az3000802-bepool --frontend-ip-name
loadBalancedFrontEndOutbound --location $LOCATION --public-ip-address
az3000802-lb-pip01
```

4. From the Cloud Shell pane, run the following to create an outbound rule:

```
az network lb outbound-rule create --resource-group az3000801-LabRG --lb-
name az3000802-lb --name outboundRuleaz30000802 --frontend-ip-configs load-
BalancedFrontEndOutbound --protocol All --idle-timeout 15 --outbound-ports
10000 --address-pool az3000802-bepool
```

5. **Note**: Wait for the operation to complete. This should not take more than 1 minute.
6. Close the Cloud Shell pane.
7. In the Azure portal, navigate to the blade displaying the properties of the Azure Load Balancer **az3000802-lb**.
8. On the **az3000802-lb** blade, click **Backend pools**.
9. On the **az3000802-lb - Backend pools** blade, click **az3000802-bepool**.
10. On the **az3000802-bepool** blade, specify the following settings and click **Save**:
    - Virtual network: **az3000801-vnet (4 VM)**
    - VIRTUAL MACHINE: **az3000802-vm0** IP ADDRESS: **ipconfig1 (10.0.1.4)** or **ipconfig1 (10.0.1.5)**
    - VIRTUAL MACHINE: **az3000802-vm1** IP ADDRESS: **ipconfig1 (10.0.1.5)** or **ipconfig1 (10.0.1.4)**
11. **Note**: Wait for the operation to complete. This should not take more than 1 minute.

Task 3: Verify that the outbound rule took effect

1. In the Azure portal, navigate to the **az3000802-lb** blade and note the value of the **Public IP address** entry.
2. On the lab computer, from the Remote Desktop session to **az3000801-vm0**, run the following to start a Remote Desktop session to **az3000802-vm0**.

   ```
   mstsc /v:az3000802-vm0
   ```

3. When prompted, authenticate by specifying the following values:
   - User name: **Student**
   - Password: **Pa55w.rd1234**
4. Within the Remote Desktop session to **az3000802-vm0**, start a Windows PowerShell session and run the following to determine your current public IP address:

   ```
   Invoke-RestMethod http://ipinfo.io/json
   ```

5. Review the output of the cmdlet and verify that the IP address entry matches the public IP address you identified earlier in this task.

Result: After you completed this exercise, you have configured and tested Azure Load Balancer Standard outbound rules

Review Question

Module 3 Review Question

Redis cache

A company has several applications in Azure that use Azure Cache.

You plan to migrate the applications to use Azure Redis Cache. You must prepare the environment for the change

What options are available for preparing the environment? What should you do?

Suggested Answer ↓

Before you can migrate to Azure Redis Cache you need to register a resource provider for your subscription. You can create a new Azure Redis Cache instance by using the Azure portal, Azure CLI, or Azure PowerShell.

Module 4 Module Implementing Authentication

Implementing authentication in applications

Certificate-based authentication

Client certificate authentication enables each web-based client to establish its identity to a server by using a digital certificate, which provides additional security for user authentication. In the context of Microsoft Azure, certificate-based authentication enables you to be authenticated by Azure Active Directory (Azure AD) with a client certificate on a Windows or mobile device when connecting to different services, including (but not limited to):

- Custom services authored by your organization
- Microsoft SharePoint Online
- Microsoft Office 365 (or Microsoft Exchange)
- Skype for Business
- Azure API Management
- Third-party services deployed in your organization

Helping to secure back-end services

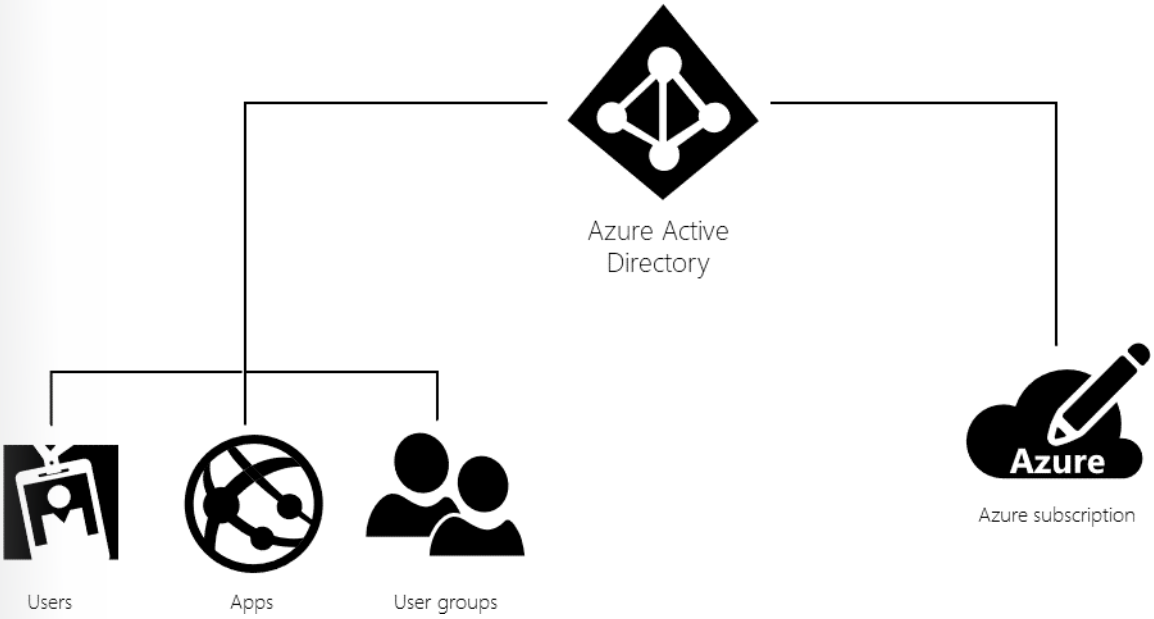
Certificate-based authentication can be useful in scenarios where your organization has multiple front-end applications communicating with back-end services. Traditionally, the certificates are installed on each server, and the machines trust each other after validating certificates. This same traditional structure can be used for infrastructure in Azure.

With cloud-native applications, you can use certificates to help secure connections in hybrid scenarios. For example, you can restrict access to your Azure web app by enabling different types of authentication for it. One way to do so is to authenticate using a client certificate when the request is over Transport Layer Security (TLS) / Secure Sockets Layer (SSL). This mechanism is called TLS mutual authentication or

client certificate authentication. As another example, API Management allows more-secure access to the back-end service of an API using client certificates.

Azure Active Directory (Azure AD)

Azure AD is an identity and access management cloud solution that provides directory services, identity governance, and application access management. Azure AD quickly enables single sign-on (SSO) to thousands of pre-integrated commercial and custom apps in the Azure AD application gallery. A single Azure AD directory is automatically associated with an Azure subscription when it is created. As the identity service in Azure, Azure AD then provides all identity management and access control functions for cloud-based resources. These resources can include users, apps, and groups for an individual tenant (organization), as shown in the following diagram:



Azure offers several ways to leverage identity as a service (IDaaS) with varying levels of complexity.

Understanding the difference between Active Directory Domain Services and Azure Active Directory

Both Azure AD and Active Directory Domain Services (AD DS) are systems that store directory data and manage communication between users and resources, including user logon processes, authentication, and directory searches.

If you are already familiar with AD DS, first introduced with Windows 2000 Server, then you probably understand the basic concept of an identity service. However, it's also important to understand that Azure AD is not just a domain controller in the cloud. It is an entirely new way of providing IDaaS in Azure that requires an entirely new way of thinking to fully embrace cloud-based capabilities and help protect your organization from modern threats.

AD DS is a server role in Windows Server, which means that it can be deployed on physical machines or virtual machines (VMs). It has a hierarchical structure based on X.500. It uses DNS for locating objects, can be interacted with using Lightweight Directory Access Protocol (LDAP), and primarily uses Kerberos for authentication. Windows Server Active Directory enables organizational units (OUs) and Group Policy Objects (GPOs) in addition to joining machines to the domain, and trusts are created between domains.

IT has protected its security perimeter for years using AD DS, but modern, perimeter-less enterprises supporting identity needs for employees, customers, and partners require a new control plane. Azure AD is that identity control plane. Security has moved beyond the corporate firewall to the cloud, where Azure AD help protect company resources and access by providing one common identity for users (either on-premises or in the cloud). This gives your users the flexibility to more securely access the apps they need to get their work done from almost any device. Seamless, risk-based data protection controls, backed by machine-learning capabilities and in-depth reporting, which IT needs to help keep company data secure, are also provided.

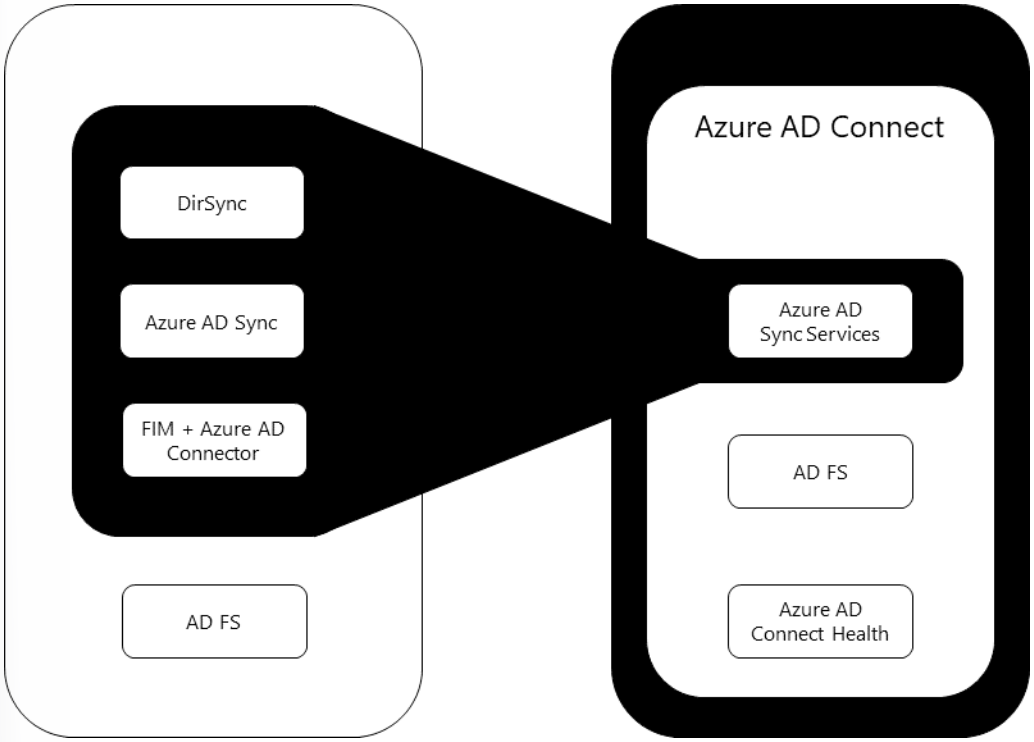
Azure AD is a multi-customer public directory service, which means that within Azure AD, you can create a tenant for your cloud servers and applications, such as Office 365. Users and groups are created in a flat structure without OUs or GPOs. Authentication is performed through protocols such as SAML, WS-Federation, and Open Authorization (OAuth). It's possible to query Azure AD, but instead of using LDAP, you must use a REST API called Azure AD Graph API. These all work over HTTP and HTTPS.

Azure AD Connect

Azure AD Connect integrates on-premises directories with Azure AD. This allows you to provide a common identity for enterprise users in Office 365, Azure, and software as a service (SaaS) applications.

Azure AD Connect is made up of three primary components: the synchronization services, the optional Active Directory Federation Services (AD FS) component, and the monitoring component named Azure AD Connect Health.

- **Synchronization** - This component is responsible for creating users, groups, and other objects. It is also responsible for making sure identity information for your on-premises users and groups is matching the cloud.
- **Active Directory Federation Services** - Federation is an optional part of Azure AD Connect and can be used to configure a hybrid environment using an on-premises AD FS infrastructure. This can be used by organizations to address complex deployments, such as domain-join SSO, the enforcement of Azure AD sign-in policy, and smart card or third-party multi-factor authentication.
- **Health monitoring** - Azure AD Connect Health can provide robust monitoring and a central location in the Azure portal to view this activity.



Azure AD Connect comes with several features you can optionally turn on or that are enabled by default. Some features might sometimes require more configuration in certain scenarios and topologies.

- **Filtering** is used when you want to limit which objects are synchronized to Azure AD. By default, all users, contacts, groups, and Windows 10 computers are synchronized. You can change the filtering based on domains, OUs, or attributes.
- **Password hash synchronization** synchronizes the password hash in Active Directory to Azure AD. The end user can use the same password on-premises and in the cloud but only manage it in one location. Since it uses your on-premises Active Directory as the authority, you can also use your own password policy.
- **Password writeback** will allow your users to change and reset their passwords in the cloud and have your on-premises password policy applied.
- **Device writeback** will allow a device registered in Azure AD to be written back to on-premises Active Directory so it can be used for conditional access.
- The **prevent accidental deletes** feature is turned on by default and helps protect your cloud directory from numerous delete operations at the same time. By default, it allows 500 delete operations per run. You can change this setting depending on your organization size.
- **Automatic upgrade** is enabled by default for express settings installations and helps ensure that your Azure AD Connect is always up-to-date with the latest release.

Legacy authentication methods

Most cloud-native applications will use a token-based or certificate-based authentication scheme. However, many applications are migrated to the cloud or connected to the cloud in a hybrid way. These applications may already have significant developer investment that makes changing the authentication scheme a significant resource challenge.

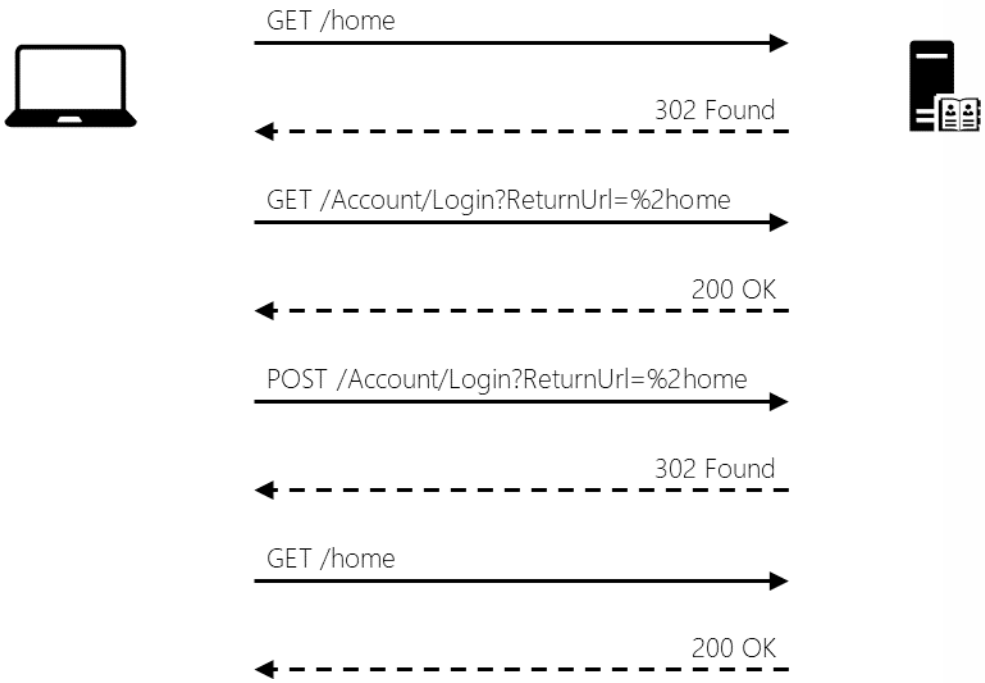
Forms-based authentication

Forms authentication uses an HTML form to send the user's credentials to the server. It is not an internet standard. Forms authentication is appropriate only for web APIs that are called from a web application so that the user can interact with the HTML form. Forms authentication does have a few disadvantages, including:

- It requires a browser client to use the HTML form.
- It requires measures to prevent cross-site request forgery (CSRF).
- User credentials are sent in plaintext as part of an HTTP request.

The most common workflow for forms-based authentication works like this:

1. The client requests a resource that requires authentication.
2. If the user is not authenticated, the server returns HTTP 302 (Found) and redirects to a login page.
3. The user enters credentials and submits the form.
4. The server returns another HTTP 302 that redirects back to the original URI. This response includes an authentication cookie.
5. The client requests the resource again. The request includes the authentication cookie, so the server grants the request.



6.

In the context of Azure, many applications using forms-based authentication are legacy applications that were shifted to Azure without being refactored or rewritten. Using Microsoft ASP.NET forms authentication as an example, migration to the cloud would require only changing the connection string for the database that is used to store the forms authentication data. Using Azure as an example, you can migrate the identity database from Microsoft SQL Server to Azure SQL Database to continue to use forms-based authentication in Azure.

Windows-based authentication

Integrated Windows authentication enables users to log in with their Windows credentials using Kerberos or NTLM. The client sends credentials in the **Authorization** header. Windows authentication is best suited for an intranet environment. Windows authentication does have a few disadvantages, including:

- It's difficult to use in internet applications without exposing the entire user directory.
- It can't be used in Bring Your Own Device (BYOD) scenarios.
- It Requires Kerberos or Integrated Windows Authentication (NTLM) support in the client browser or device.
- The client must be joined to the Active Directory Domain.

In a hybrid deployment, it is common to see the main responsibilities of identity moved from on-premises Active Directory to Azure AD. The on-premises Active Directory servers remain as a way to manage physical machines and to enable simple Windows-based authentication. **Azure AD Connect** is used to synchronize identity from Azure AD to the on-premises Active Directory servers.

Token-based authentication

Claims-based authentication in .NET

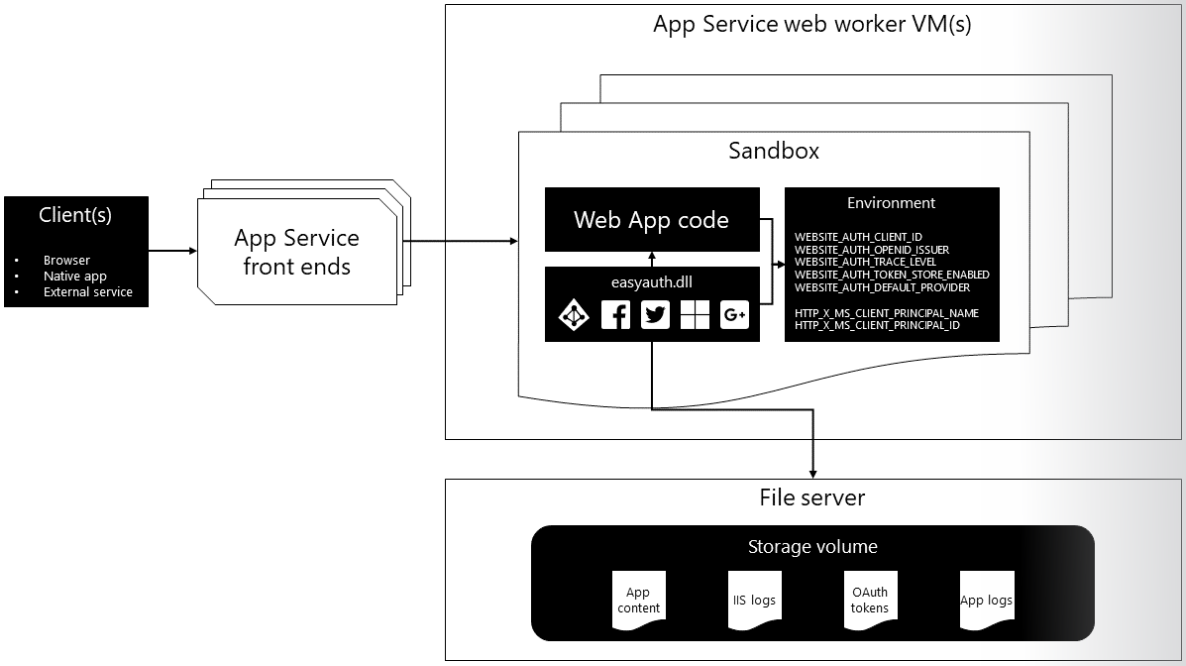
Historically, ASP.NET applications used forms authentication to solve member requirements that were common in the early 2000s. These requirements revolved mostly around authoring login forms and managing a SQL Server database for user names, passwords, and profile data. Today, there is a much broader array of data storage options for web applications, and most developers want to enable their sites to use social identity providers for authentication and authorization functionality. While it's possible to implement these new features in a database, it is unnecessarily difficult when many identity providers implement storage, tokens, and claims already.

ASP.NET Identity is a unified identity platform for ASP.NET applications that can be used across all flavors of ASP.NET and that can be used in web, phone, store, or hybrid applications. ASP.NET Identity implements two core features that makes it ideal for token-based authentication:

- ASP.NET Identity implements a **provider model for logins**. Today you may want to log in using a local Active Directory server, but tomorrow you may want to migrate to Azure AD. In ASP.NET Identity, you can simply add, remove, or replace providers. If your company decides to implement social network logins, you can keep adding providers or write your own providers without changing any other code in your application.
- ASP.NET Identity supports **claims-based authentication**, where the user's identity is represented as a set of claims. Claims allow developers to be a lot more expressive in describing a user's identity than roles allow. Whereas role membership is just a Boolean value (member or non-member), a claim can include rich information about the user's identity and membership. Most social providers return metadata about the logged-in user as a series of claims.

App Service authentication and authorization

Azure App Service provides built-in authentication and authorization support, so you can sign in users and access data by writing minimal or no code in your app instance. The authentication and authorization module runs in the same sandbox as your application code. When it's enabled, every incoming HTTP request passes through it before being handled by your application code.



All AuthN/AuthZ logic, including cryptography for token validation and session management, executes in the worker sandbox and outside of the web app code. The module runs separately from your application code and is configured using app settings. No software development kits (SDKs), specific languages, or changes to your application code are required.

Identity information flows directly into the application code. For all language frameworks, App Service makes the user's claims available to your code by injecting them into the request headers. For Microsoft .NET applications, App Service populates **ClaimsPrincipal.Current** with the authenticated user's claims, so you can follow the standard .NET code pattern, including the **[Authorize]** attribute. Similarly, for PHP apps, App Service populates the **_SERVER['REMOTE_USER']** variable.

App Service provides a built-in token store, which is a repository of tokens that are associated with the users of your web apps, APIs, or native mobile apps. You typically must write code to collect, store, and refresh these tokens in your application. With the token store, you just retrieve the tokens when you need them and tell App Service to refresh them when they become invalid. When you enable authentication with any provider, this token store is immediately available to your app. The token information can be used in your application code to perform tasks such as:

- Posting to the authenticated user's Facebook timeline.
- Reading the user's corporate data from the Azure AD Graph API or even from the Microsoft Graph.

Implement multi-factor authentication

Multi-factor authentication

When a user logs into an application, they typically provide a username and password. The password is provided by the user as a piece of evidence to the authentication system that the user is who they claim to be. The password is considered **one factor** proving the user's identity. A user could have other factors that proves their identity, such as:

- A physical badge from the company.
- Knowledge of the answers to security questions.
- A mobile device, registered with the company, that can receive notifications, phone calls, or SMS messages.
- Their physical appearance that can be captured by a camera device.
- Their fingerprint that could be captured by a biometric scanner.

Unfortunately, a single factor can potentially be compromised either intentionally or unintentionally. A badge can be stolen and used by an unauthorized party. During a robbery, someone could ask you to use your fingerprint on a device. A mobile company could accidentally send SMS messages to another device.

In security best practices, it is recommended to use two or more factors when authenticating users. This practice is referred to as **multi-factor authentication**. Using an enterprise as an example, the company could require employees to scan their badges and then enter their passwords as two factors of authentication. In the world of security, it is often recommended to have two of the following factors:

- **Knowledge** – Something that only the user knows (security questions, password, or PIN).
- **Possession** – Something that only the user has (corporate badge, mobile device, or security token).
- **Inherence** – Something that only the user is (fingerprint, face, voice, or iris).

The security of two-step verification lies in its layered approach. Compromising multiple authentication factors presents a significant challenge for attackers. Even if an attacker manages to learn the user's password, it is useless without possession of the additional authentication method.

Multi-factor authentication with Azure AD

Azure Multi-Factor Authentication (MFA) is a two-step verification solution that is built in to Azure AD. Administrators can configure approved authentication methods to ensure that at least two factors are used while still keeping the sign-in process as streamlined as possible.

There are two ways to enable MFA:

- The first option is to **enable each user** for MFA. When users are enabled individually, they perform two-step verification each time they sign in. There are a few exceptions, such as when they sign in from trusted IP addresses or when the remembered devices feature is turned on.
- The second option is to set up a **conditional access policy** that requires two-step verification under certain conditions. This method uses the Azure AD Identity Protection risk policy to require two-step verification based only on the sign-in risk for all cloud applications.

Once MFA is enabled, administrators can choose which methods of authentication are available to users. Once users enroll, they must choose at least one method from the list that the administrator has enabled. These methods include:

Method	Description
Call to phone	Places an automated voice call. The user answers the call and presses # on the phone keypad to authenticate. The phone number is not synchronized to on-premises Active Directory.
Text message to phone	Sends a text message that contains a verification code. The user is prompted to enter the verification code into the sign-in interface. This process is called one-way SMS. Two-way SMS means that the user must text back a particular code.
Notification through mobile app	Sends a push notification to your phone or registered device. The user views the notification and selects Verify to complete the verification.
Verification code from mobile app	The Microsoft Authenticator app generates a new OAuth verification code every 30 seconds. The user enters the verification code into the sign-in interface.

The Microsoft Authenticator app helps to prevent unauthorized access to accounts and to stop fraudulent transactions by offering an additional level of security for Azure AD accounts or Microsoft accounts. It can be used either as a second verification method or as a replacement for a password when using phone sign-in. The Authenticator app fully supports both the **Verification code** and **Notification** methods of verification in MFA. The Authenticator app is available for Windows phone, Android, and iOS.

Implementing custom multi-factor authentication using .NET

The Multi-Factor Authentication SDK lets you build two-step verification directly into the sign-in or transaction processes of applications in your Azure AD tenant.

The Multi-Factor Authentication SDK is available for C#, Visual Basic (.NET), Java, Perl, PHP, and Ruby. The SDK provides a thin wrapper around two-step verification. It includes everything you need to write your code, including commented source code files, example files, and a detailed ReadMe file. Each SDK also includes a certificate and private key for encrypting transactions that are unique to your MFA provider. As long as you have a provider, you can download the SDK in as many languages and formats as you need.

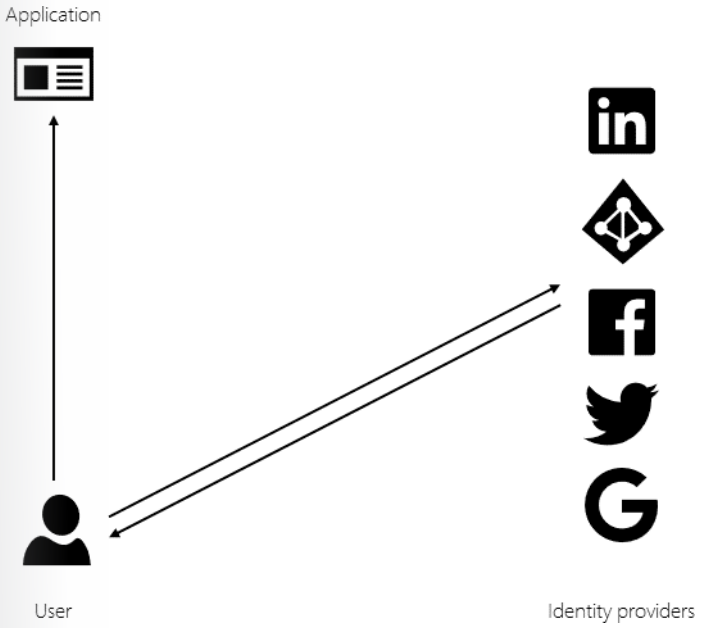
Because the APIs do not have access to users registered in Azure AD, you must provide user information in a file or database. Also, the APIs do not provide enrollment or user management features, so you need to build these processes into your application.

Claims-based authorization

Claims

Authorization is the process of determining which entities have permission to change, view, or otherwise access a computer resource. For example, in a business, only managers may be allowed to access the files of their employees. In the past, this was simple to accomplish with identity databases using protocols like Lightweight Directory Access Protocol (LDAP) or tools like Active Directory Domain Services. Whenever a user attempted to access an application, the application would query the identity database.

In a world where identity is usually managed by third-party providers, like Microsoft Azure Active Directory, Facebook, Google, LinkedIn, and Twitter, this information needs to be shared in a standardized way to applications. In the simplest workflow, the user needs to access an application, so they first log in using their social identity. Once they are logged in, the identity provider is trusted by the organization's application and can share claims about that user with the application.



When an identity is created, it may be assigned one or more claims issued by a trusted party. A claim is a name/value pair that represents what the subject is and not what the subject can do. For example, you may have a driver's license issued by a local driving license authority. Your driver's license has your date of birth on it. In this case, the claim name would be **DateOfBirth**, the claim value would be your date of birth — for example, **June 8, 1970** — and the issuer would be the driving license authority. An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

Note: The terms authentication and authorization can be confusing. To keep it simple, authentication is the act of verifying someone's identity. When you authenticate someone, you are determining who they are. Authorization is the act of verifying that someone has access to a certain subsystem or operation. When you authorize someone, you are determining what they can do.

Claims-based authorization

Claims-based authorization is an approach where the authorization decision to grant or deny access is based on arbitrary logic that uses data available in claims to make the decision. Claims-based authoriza-

tion, at its simplest, checks the value of a claim and allows access to a resource based on that value. For example, if you want access to a night club, the authorization process might be:

- The door security officer evaluates the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.

In a relying party application, authorization determines what resources an authenticated identity is allowed to access and what operations it is allowed to perform on those resources. Improper or weak authorization leads to information disclosure and data tampering.

Claim-based authorization checks are declarative—the developer embeds them within their code, against a controller or an action within a controller, specifying claims that the current user must possess and optionally the value the claim must hold to access the requested resource. Claims requirements are policy based; the developer must build and register a policy expressing the claims requirements.

Claims-based authorization in Microsoft ASP.NET

The simplest type of claim policy looks for the presence of a claim and doesn't check the value. First, you need to build and register the policy. This takes place as part of the authorization service configuration, which normally takes place in **ConfigureServices()** in your **Startup.cs** file:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
          options.AddPolicy("EmployeeOnly", policy => policy.Require-
Claim("EmployeeNumber"));
    });
}
```

In this case, the **EmployeeOnly** policy checks for the presence of an **EmployeeNumber** claim on the current identity. You then apply the policy using the **Policy** property on the **AuthorizeAttribute** attribute to specify the policy name:

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    return View();
}
```

Alternatively, the **AuthorizeAttribute** attribute can be applied to an entire controller; in this instance, only identities matching the policy will be allowed access to any action on the controller:

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public ActionResult VacationBalance()
    {
    }
}
```

If you have a controller that's protected by the **AuthorizeAttribute** attribute but want to allow anonymous access to particular actions, you apply the **AllowAnonymousAttribute** attribute:

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public ActionResult VacationBalance()
    {
    }

    [AllowAnonymous]
    public ActionResult VacationPolicy()
    {
    }
}
```

Most claims come with a value. You can specify a list of allowed values when creating the policy. The following example succeeds only for employees whose employee number is 1, 2, 3, 4 or 5:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Founders", policy =>
                          policy.RequireClaim("EmployeeNumber", "1", "2",
"3", "4", "5"));
    });
}
```

Role-based access control (RBAC) authorization

Role-based authorization

Role-based authorization is an authorization approach in which user permissions are managed and enforced by an application based on user roles. If a user has a role that is required to perform an action, access is granted; otherwise, access is denied. When an identity is created, it may belong to one or more roles. For example, Holly may belong to the Administrator and User roles, whereas Adam may belong only to the User role. How these roles are created and managed depends on the backing store of the authorization process.

Role-Based authorization in ASP.NET

Roles are exposed to the developer through the `IsInRole` method on the `ClaimsPrincipal` class. Role-based authorization checks are declarative—the developer embeds them within their code, against a controller or an action within a controller, specifying roles that the current user must be a member of to access the requested resource.

For example, the following code limits access to any actions on the `AdministrationController` to users who are members of the **Administrator** role:

```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{
}
```

You can specify multiple roles as a comma separated list:

```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
}
```

This controller would be accessible only by users who are members of the **HRManager** role or the **Finance** role.

If you apply multiple attributes, an accessing user must be a member of all the roles specified. The following sample requires that a user be a member of both the **PowerUser** and **ControlPanelUser** roles:

```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
}
```

You can further limit access by applying additional role authorization attributes at the action level:

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
```

```
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

In the previous code snippet, members of either the **Administrator** role or the **PowerUser** role can access the controller and the `SetTime` action, but only members of the **Administrator** role can access the `ShutDown` action.

You can also lock down a controller but allow anonymous, unauthenticated access to individual actions:

```
[Authorize]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [AllowAnonymous]
    public ActionResult Login()
    {
    }
}
```

Role requirements can also be expressed using the `Policy` syntax, where a developer registers a policy at startup as part of the authorization service configuration. This normally occurs in `ConfigureServices()` in your `Startup.cs` file:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdministratorRole", policy => policy.
RequireRole("Administrator"));
    });
}
```

Policies are applied using the `Policy` property on the `AuthorizeAttribute` attribute:

```
[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown()
{
    return View();
```

```
}
```

If you want to specify multiple allowed roles in a requirement, you can specify them as parameters to the `RequireRole` method:

```
options.AddPolicy("ElevatedRights", policy =>
                  policy.RequireRole("Administrator", "PowerUser", "Backu-
pAdministrator"));
```

This example authorizes users who belong to the **Administrator**, **PowerUser**, or **BackupAdministrator** roles.

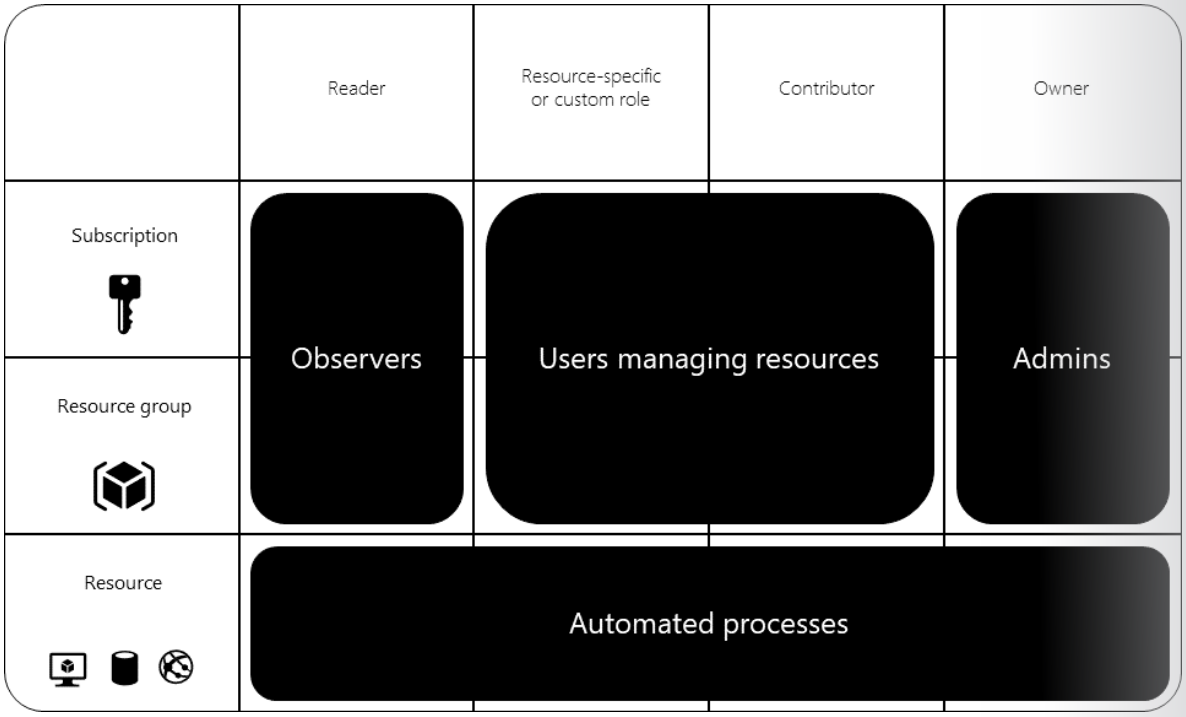
Note: You can mix and match both claims-based authorization and role-based authorization. Is it typical to see the role defined as a special claim. The role claim type is expressed using the following URI: `http://schemas.microsoft.com/ws/2008/06/identity/claims/role`.

Role-based access control (RBAC)

Role-based access control (RBAC) is a system that provides fine-grained access management of resources in Azure. Using RBAC, you can segregate duties within your team and grant only the amount of access to users that they need to perform their jobs. RBAC in Azure is an authorization system built on Azure Resource Manager that provides fine-grained access management to Azure resources, such as compute and storage.

Using RBAC, you can segregate duties within your team and grant only the amount of access to users that they need to perform their jobs. Instead of giving everybody unrestricted permissions in your Azure subscription or resources, you can allow only certain actions at a particular scope.

When planning your access control strategy, it's a best practice to grant users the least privileges to get their work done. The following diagram shows a suggested pattern for using RBAC.

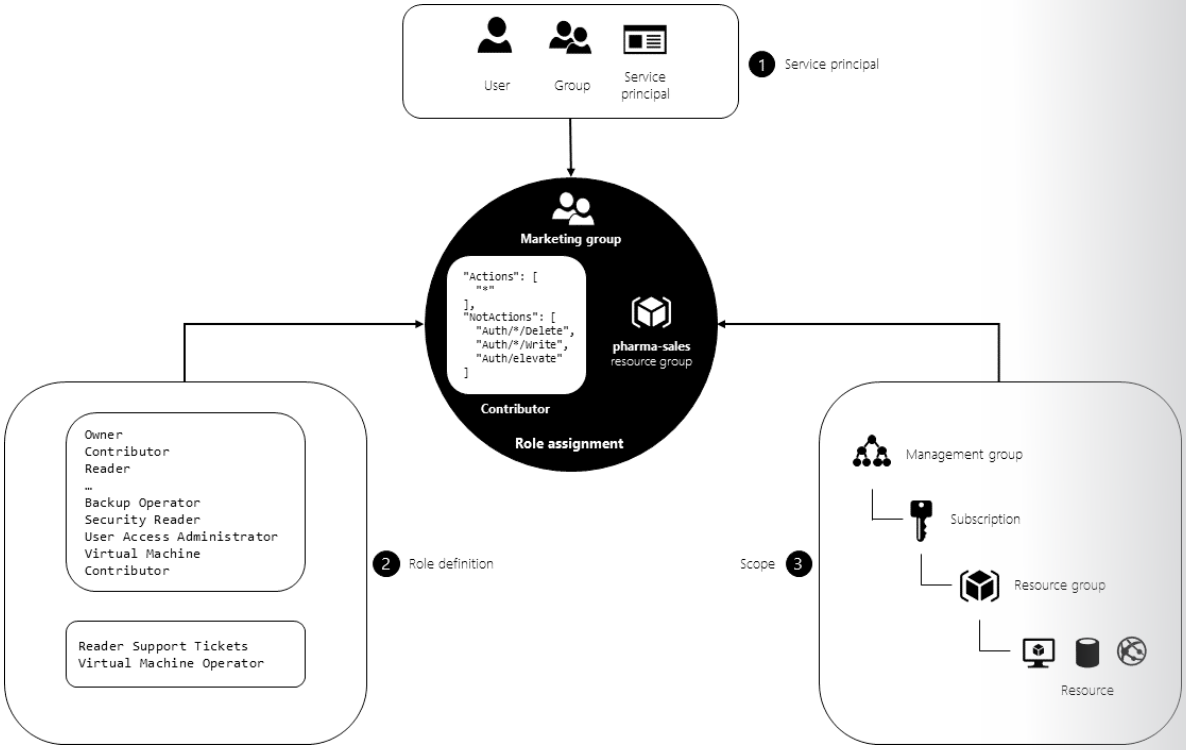


The way you control access to resources using RBAC is to create role assignments. This is a key concept to understand — it's how permissions are enforced. A role assignment consists of three elements: a security principal, a role definition, and the scope.

- A **security principal** is an object that represents a user, group, or service principal that is requesting access to Azure resources.
  - A **user** is an individual who has a profile in Azure Active Directory. You can also assign roles to users in other tenants.
  - A **group** is a set of users created in Azure Active Directory. When you assign a role to a group, all users within that group have that role.
  - A **service principal** is a security identity used by applications or services to access specific Azure resources. You can think of it as a user identity (username and password or certificate) for an application.
- A **role** definition is a collection of permissions. It's sometimes just called a role. A role definition lists the operations that can be performed, such as read, write, and delete. Roles can be high level, like owner, or specific, like virtual machine reader.
- The **Scope** is the boundary that the access applies to. When you assign a role, you can further limit the actions allowed by defining a scope. This is helpful if you want to make someone a Website Contributor but only for one resource group. In Azure, you can specify a scope at multiple levels: management group, subscription, resource group, or resource. Scopes are structured in a parent-child relationship.

A **role assignment** is the process of binding a role definition to a user, group, or service principal at a particular scope for the purpose of granting access. Access is granted by creating a role assignment, and access is revoked by removing a role assignment.

The following diagram shows an example of a role assignment. In this example, the Marketing group has been assigned the Contributor role for the pharma-sales resource group. This means that users in the Marketing group can create or manage any Azure resource in the pharma-sales resource group. Marketing users do not have access to resources outside the pharma-sales resource group, unless they are part of another role assignment.



Built-in roles

RBAC in Azure includes over 70 built-in roles. There are four fundamental RBAC roles. The first three apply to all resource types:

RBAC role in Azure	Permissions	Notes
Owner	Has full access to all resources and can delegate access to others	The Service Administrator and Co-Administrators are assigned the Owner role at the subscription scope. This applies to all resource types.
Contributor	Creates and manages all types of Azure resources but cannot grant access to others	This applies to all resource types.
Reader	Creates and manages all types of Azure resources but cannot grant access to others	This applies to all resource types.
User Access Administrator	Manages user access to Azure resources	

The rest of the built-in roles allow the management of specific Azure resources. For example, the Virtual Machine Contributor role allows the user to create and manage virtual machines.

Note: Only the Azure portal and the Azure Resource Manager APIs support RBAC. Users, groups, and applications that are assigned RBAC roles cannot use the Azure classic deployment model APIs.

Implement OAuth2 authentication

Authorize access to web applications using OpenID Connect

OpenID Connect is a simple identity layer built on top of the OAuth 2.0 protocol. OAuth 2.0 defines mechanisms to obtain and use access tokens to access protected resources, but they do not define standard methods to provide identity information. OpenID Connect implements authentication as an extension to the OAuth 2.0 authorization process. It provides information about the end user in the form of an `id_token` that verifies the identity of the user and provides basic profile information about the user.

OpenID Connect is our recommendation if you are building a web application that is hosted on a server and accessed via a browser.

Register your application with your AD tenant

First, you need to register your application with your Azure Active Directory (Azure AD) tenant. This will give you an Application ID for your application, as well as enable it to receive tokens.

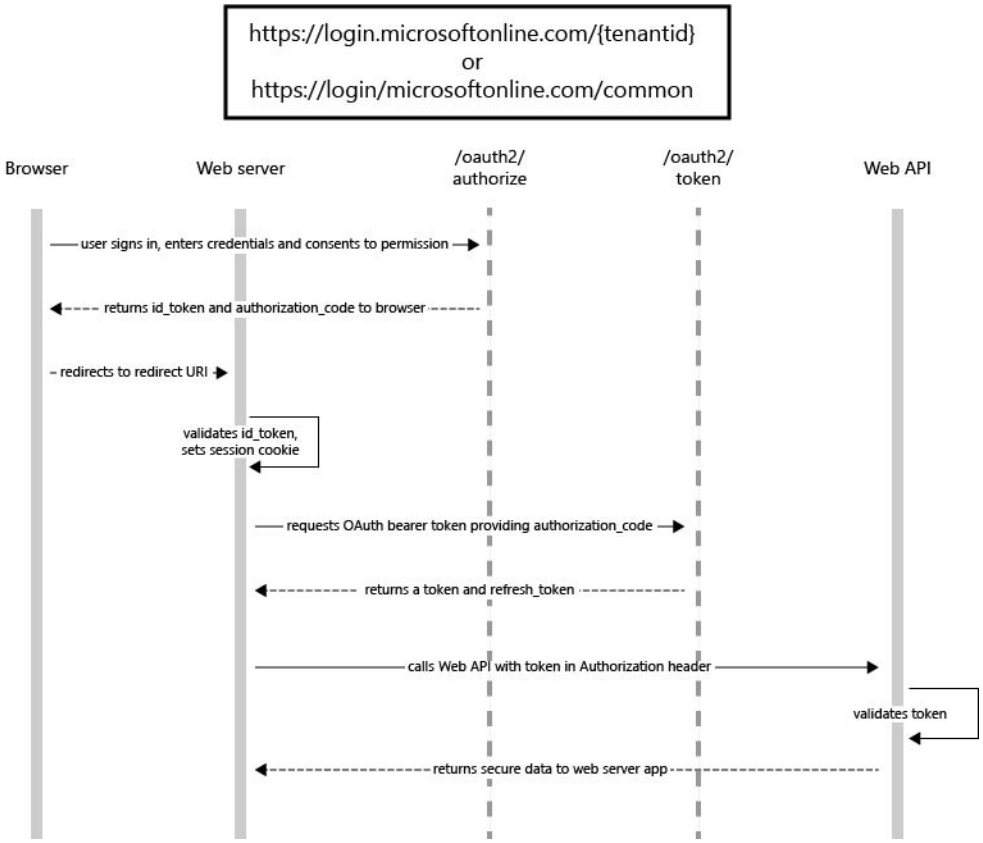
- Sign in to the Azure portal.
- Choose your Azure AD tenant by clicking on your account in the top right corner of the page, followed by clicking on the Switch Directory navigation and then select the appropriate tenant.

--Skip this step, if you've only one Azure AD tenant under your account or if you've already selected the appropriate Azure AD tenant.

- In the left hand navigation pane, click on **Azure Active Directory**.
- Click on **App Registrations** and click on **New application registration**.
- Follow the prompts and create a new application. It doesn't matter if it is a web application or a native application for this tutorial.
- --For Web Applications, provide the Sign-On URL, which is the base URL of your app, where users can sign in e.g `http://localhost:12345`.
- --For Native Applications provide a Redirect URI, which Azure AD will use to return token responses. Enter a value specific to your application, .e.g `http://MyFirstAADApp`.
- Once you've completed registration, Azure AD will assign your application a unique client identifier, the **Application ID**. You need this value in the next sections, so copy it from the application page.
- To find your application in the Azure portal, click **App registrations**, and then click **View all applications**.

Authentication flow using OpenID Connect

The most basic sign-in flow contains the following steps - each of them is described in detail below.



OpenID Connect metadata document

OpenID Connect describes a metadata document that contains most of the information required for an app to perform sign-in. This includes information such as the URLs to use and the location of the service's public signing keys. The OpenID Connect metadata document can be found at:

```
https://login.microsoftonline.com/{tenant}/.well-known/openid-configuration
```

The metadata is a simple JavaScript Object Notation (JSON) document. See the following snippet for an example. The snippet's contents are fully described in the **OpenID Connect specification**[1]. Note that providing tenant rather than `common` in place of {tenant} above will result in tenant-specific URIs in the JSON object returned.

```
{
    "authorization_endpoint": "https://login.microsoftonline.com/common/
oauth2/authorize",
    "token_endpoint": "https://login.microsoftonline.com/common/oauth2/
token",
    "token_endpoint_auth_methods_supported":
    [
        "client_secret_post",
        "private_key_jwt",
        "client_secret_basic"
```

1 https://openid.net/

```
    ],
    "jwks_uri": "https://login.microsoftonline.com/common/discovery/keys"
    "userinfo_endpoint":"https://login.microsoftonline.com/{tenant}/openid/
userinfo",
    ...
}
```

Send the sign-in request

When your web application needs to authenticate the user, it must direct the user to the /authorize endpoint. This request is similar to the first leg of the **OAuth 2.0 Authorization Code Flow**[2], with a few important distinctions:

- The request must include the scope `openid` in the `scope` parameter.
- The `response_type` parameter must include `id_token`.
- The request must include the `nonce` parameter.

So a sample request would look like this:

```
// Line breaks for legibility only

GET https://login.microsoftonline.com/{tenant}/oauth2/authorize?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&response_type=id_token
&redirect_uri=http%3A%2F%2Flocalhost%3a12345
&response_mode=form_post
&scope=openid
&state=12345
&nonce=7362CAEA-9CA5-4B43-9BA3-34D7C303EBA7
```

Parameter		Description
tenant	required	The `{tenant}` value in the path of the request can be used to control who can sign into the application. The allowed values are tenant identifiers, for example, `8eaef023-2b34-4da1-9baa-8bc8c9d6a490` or `contoso.onmicrosoft.com` or `common` for tenant-independent tokens

2 https://docs.microsoft.com/en-us/azure/active-directory/develop/v1-protocols-oauth-code

Parameter		Description
client_id	required	The Application Id assigned to your app when you registered it with Azure AD. You can find this in the Azure Portal. Click **Azure Active Directory**, click **App Registrations**, choose the application and locate the Application Id on the application page.
response_type	required	Must include `id_token` for OpenID Connect sign-in. It may also include other response_types, such as `code` or `token`.
scope	required	A space-separated list of scopes. For OpenID Connect, it must include the scope `openid`, which translates to the "Sign you in" permission in the consent UI. You may also include other scopes in this request for requesting consent.
nonce	required	A value included in the request, generated by the app, that is included in the resulting `id_token` as a claim. The app can then verify this value to mitigate token replay attacks. The value is typically a randomized, unique string or GUID that can be used to identify the origin of the request.
redirect_uri	recommended	The redirect_uri of your app, where authentication responses can be sent and received by your app. It must exactly match one of the redirect_uris you registered in the portal, except it must be url encoded.

Parameter		Description
response_mode	optional	Specifies the method that should be used to send the resulting authorization_code back to your app. Supported values are `form_post` for *HTTP form post* and `fragment` for *URL fragment*. For web applications, we recommend using `response_mode=form_post` to ensure the most secure transfer of tokens to your application. The default for any flow including an id_token is `fragment`.
state	recommended	A value included in the request that is returned in the token response. It can be a string of any content that you wish. A randomly generated unique value is typically used for preventing cross-site request forgery attacks. The state is also used to encode information about the user's state in the app before the authentication request occurred, such as the page or view they were on.
prompt	optional	Indicates the type of user interaction that is required. Currently, the only valid values are 'login', 'none', and 'consent'. `prompt=login` forces the user to enter their credentials on that request, negating single-sign on. `prompt=none` is the opposite - it ensures that the user is not presented with any interactive prompt whatsoever. If the request cannot be completed silently via single-sign on, the endpoint returns an error. `prompt=consent` triggers the OAuth consent dialog after the user signs in, asking the user to grant permissions to the app.

Parameter		Description
login_hint	optional	Can be used to pre-fill the username/email address field of the sign-in page for the user, if you know their username ahead of time. Often apps use this parameter during reauthentication, having already extracted the username from a previous sign-in using the `preferred_username` claim.

At this point, the user is asked to enter their credentials and complete the authentication.

Sample response

A sample response, after the user has authenticated, could look like this:

```
POST / HTTP/1.1
Host: localhost:12345
Content-Type: application/x-www-form-urlencoded

id_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik1uQ19W-
WmNB...&state=12345
```

Parameter	Description
id_token	The `id_token` that the app requested. You can use the `id_token` to verify the user's identity and begin a session with the user.
state	A value included in the request that is also returned in the token response. A randomly generated unique value is typically used for preventing cross-site request forgery attacks. The state is also used to encode information about the user's state in the app before the authentication request occurred, such as the page or view they were on.

Error response

Error responses may also be sent to the `redirect_uri` so the app can handle them appropriately:

```
POST / HTTP/1.1
Host: localhost:12345
Content-Type: application/x-www-form-urlencoded

error=access_denied&error_description=the+user+canceled+the+authentication
```

Parameter	Description
error	An error code string that can be used to classify types of errors that occur, and can be used to react to errors.
error_description	A specific error message that can help a developer identify the root cause of an authentication error.

Error codes for authorization endpoint errors

The following table describes the various error codes that can be returned in the error parameter of the error response.

Error Code	Description	Client Action
invalid_request	Protocol error, such as a missing required parameter.	Fix and resubmit the request. This is a development error, and is typically caught during initial testing.
unauthorized_client	The client application is not permitted to request an authorization code.	This usually occurs when the client application is not registered in Azure AD or is not added to the user's Azure AD tenant. The application can prompt the user with instruction for installing the application and adding it to Azure AD.
access_denied	Resource owner denied consent	The client application can notify the user that it cannot proceed unless the user consents.
unsupported_response_type	The authorization server does not support the response type in the request.	Fix and resubmit the request. This is a development error, and is typically caught during initial testing.
server_error	The server encountered an unexpected error.	Retry the request. These errors can result from temporary conditions. The client application might explain to the user that its response is delayed due to a temporary error.
temporarily_unavailable	The server is temporarily too busy to handle the request.	Retry the request. The client application might explain to the user that its response is delayed due to a temporary condition.
invalid_resource	The target resource is invalid because it does not exist, Azure AD cannot find it, or it is not correctly configured.	This indicates the resource, if it exists, has not been configured in the tenant. The application can prompt the user with instruction for installing the application and adding it to Azure AD.

Validate the id_token

Just receiving an `id_token` is not sufficient to authenticate the user; you must validate the signature and verify the claims in the `id_token` per your app's requirements. The Azure AD endpoint uses JSON Web Tokens (JWTs) and public key cryptography to sign tokens and verify that they are valid.

You can choose to validate the `id_token` in client code, but a common practice is to send the `id_token` to a backend server and perform the validation there. Once you've validated the signature of the `id_token`, there are a few claims you are required to verify.

You may also wish to validate additional claims depending on your scenario. Some common validations include:

- Ensuring the user/organization has signed up for the app.
- Ensuring the user has proper authorization/privileges
- Ensuring a certain strength of authentication has occurred, such as multi-factor authentication.

Once you have validated the `id_token`, you can begin a session with the user and use the claims in the `id_token` to obtain information about the user in your app. This information can be used for display, records, personalization, etc.

Send a sign-out request

When you wish to sign the user out of the app, it is not sufficient to clear your app's cookies or otherwise end the session with the user. You must also redirect the user to the end_session_endpoint for sign-out. If you fail to do so, the user will be able to reauthenticate to your app without entering their credentials again, because they will have a valid single sign-on session with the Azure AD endpoint.

You can simply redirect the user to the end_session_endpoint listed in the OpenID Connect metadata document:

```
GET https://login.microsoftonline.com/common/oauth2/logout?
post_logout_redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
```

Parameter		Description
post_logout_redirect_uri	recommended	The URL that the user should be redirected to after successful logout. If not included, the user is shown a generic message.

Single sign-out

When you redirect the user to the `end_session_endpoint`, Azure AD clears the user's session from the browser. However, the user may still be signed in to other applications that use Azure AD for authentication. To enable those applications to sign the user out simultaneously, Azure AD sends an HTTP GET request to the registered `LogoutUrl` of all the applications that the user is currently signed in to. Applications must respond to this request by clearing any session that identifies the user and returning a `200` response. If you wish to support single sign out in your application, you must implement such a `LogoutUrl` in your application's code. You can set the `LogoutUrl` from the Azure portal:

1. Navigate to the Azure Portal.
2. Choose your Active Directory by clicking on your account in the top right corner of the page.

3. From the left hand navigation panel, choose **Azure Active Directory**, then choose **App registrations** and select your application.
4. Click on **Settings**, then **Properties** and find the **Logout URL** text box.

Token Acquisition

Many web apps need to not only sign the user in, but also access a web service on behalf of that user using OAuth. This scenario combines OpenID Connect for user authentication while simultaneously acquiring an `authorization_code` that can be used to get `access_tokens` using the OAuth Authorization Code Flow.

Get Access Tokens

To acquire access tokens, you need to modify the sign-in request from above:

```
// Line breaks for legibility only

GET https://login.microsoftonline.com/{tenant}/oauth2/authorize?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e        // Your registered
Application Id
&response_type=id_token+code
&redirect_uri=http%3A%2F%2Flocalhost%3a12345          // Your registered
Redirect Uri, url encoded
&response_mode=form_post                              // `form_post' or
'fragment'
&scope=openid
&resource=https%3A%2F%2Fservice.contoso.com%2F        // The identifier of
the protected resource (web API) that your application needs access to
&state=12345                                          // Any value, provid-
ed by your app
&nonce=678910                                         // Any value, provid-
ed by your app
```

By including permission scopes in the request and using `response_type=code+id_token`, the `authorize` endpoint ensures that the user has consented to the permissions indicated in the `scope` query parameter, and return your app an authorization code to exchange for an access token.

Successful response

A successful response using `response_mode=form_post` looks like:

```
POST /myapp/ HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded

id_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik1uQ19W-
WmNB...&code=AwABAAAAvPM1KaPlrEqdFSBzjqfTGBCmLdgfSTLEMPGYuNHSUY-
Brq...&state=12345
```

Parameter	Description
id_token	The `id_token` that the app requested. You can use the `id_token` to verify the user's identity and begin a session with the user.
code	The authorization_code that the app requested. The app can use the authorization code to request an access token for the target resource. Authorization_codes are short lived, and typically expire after about 10 minutes.
state	If a state parameter is included in the request, the same value should appear in the response. The app should verify that the state values in the request and response are identical.

Error response

Error responses may also be sent to the `redirect_uri` so the app can handle them appropriately:

```
POST /myapp/ HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded

error=access_denied&error_description=the+user+canceled+the+authentication
```

Parameter	Description
error	An error code string that can be used to classify types of errors that occur, and can be used to react to errors.
error_description	A specific error message that can help a developer identify the root cause of an authentication error.

For a description of the possible error codes and their recommended client action, see Error codes for authorization endpoint errors above.

Once you've gotten an authorization `code` and an `id_token`, you can sign the user in and get access tokens on their behalf. To sign the user in, you must validate the `id_token` exactly as described above.

Understanding the OAuth2 implicit grant flow in Azure Active Directory

The OAuth2 implicit grant is notorious for being the grant with the longest list of security concerns in the OAuth2 specification. And yet, that is the approach implemented by ADAL JS and the one we recommend when writing SPA applications. What gives? It's all a matter of tradeoffs: and as it turns out, the implicit grant is the best approach you can pursue for applications that consume a Web API via JavaScript from a browser.

What is the OAuth2 implicit grant?

The quintessential OAuth2 authorization code grant is the authorization grant that uses two separate endpoints. The authorization endpoint is used for the user interaction phase, which results in an authori-

zation code. The token endpoint is then used by the client for exchanging the code for an access token, and often a refresh token as well. Web applications are required to present their own application credentials to the token endpoint, so that the authorization server can authenticate the client.

The OAuth2 implicit grant is a variant of other authorization grants. It allows a client to obtain an access token (and id_token, when using OpenId Connect) directly from the authorization endpoint, without contacting the token endpoint nor authenticating the client. This variant was designed for JavaScript based applications running in a Web browser: in the original OAuth2 specification, tokens are returned in a URI fragment. That makes the token bits available to the JavaScript code in the client, but it guarantees they won't be included in redirects toward the server. Returning tokens via browser redirects directly from the authorization endpoint. It also has the advantage of eliminating any requirements for cross origin calls, which are necessary if the JavaScript application is required to contact the token endpoint.

An important characteristic of the OAuth2 implicit grant is the fact that such flows never return refresh tokens to the client. The next section shows how this isn't necessary and would in fact be a security issue.

Suitable scenarios for the OAuth2 implicit grant

The OAuth2 specification declares that the implicit grant has been devised to enable user-agent applications – that is to say, JavaScript applications executing within a browser. The defining characteristic of such applications is that JavaScript code is used for accessing server resources (typically a Web API) and for updating the application user experience accordingly. Think of applications like Gmail or Outlook Web Access: when you select a message from your inbox, only the message visualization panel changes to display the new selection, while the rest of the page remains unmodified. This characteristic is in contrast with traditional redirect-based Web apps, where every user interaction results in a full page postback and a full page rendering of the new server response.

Applications that take the JavaScript based approach to its extreme are called single-page applications, or SPAs. The idea is that these applications only serve an initial HTML page and associated JavaScript, with all subsequent interactions being driven by Web API calls performed via JavaScript. However, hybrid approaches, where the application is mostly postback-driven but performs occasional JS calls, are not uncommon – the discussion about implicit flow usage is relevant for those as well.

Redirect-based applications typically secure their requests via cookies, however, that approach does not work as well for JavaScript applications. Cookies only work against the domain they have been generated for, while JavaScript calls might be directed toward other domains. In fact, that will frequently be the case: think of applications invoking Microsoft Graph API, Office API, Azure API – all residing outside the domain from where the application is served. A growing trend for JavaScript applications is to have no backend at all, relying 100% on third party Web APIs to implement their business function.

Currently, the preferred method of protecting calls to a Web API is to use the OAuth2 bearer token approach, where every call is accompanied by an OAuth2 access token. The Web API examines the incoming access token and, if it finds in it the necessary scopes, it grants access to the requested operation. The implicit flow provides a convenient mechanism for JavaScript applications to obtain access tokens for a Web API, offering numerous advantages in respect to cookies:

- Tokens can be reliably obtained without any need for cross origin calls – mandatory registration of the redirect URI to which tokens are return guarantees that tokens are not displaced
- JavaScript applications can obtain as many access tokens as they need, for as many Web APIs they target – with no restriction on domains
- HTML5 features like session or local storage grant full control over token caching and lifetime management, whereas cookies management is opaque to the app
- Access tokens aren't susceptible to Cross-site request forgery (CSRF) attacks

The implicit grant flow does not issue refresh tokens, mostly for security reasons. A refresh token isn't as narrowly scoped as access tokens, granting far more power hence inflicting far more damage in case it is leaked out. In the implicit flow, tokens are delivered in the URL, hence the risk of interception is higher than in the authorization code grant.

However, a JavaScript application has another mechanism at its disposal for renewing access tokens without repeatedly prompting the user for credentials. The application can use a hidden iframe to perform new token requests against the authorization endpoint of Azure AD: as long as the browser still has an active session (read: has a session cookie) against the Azure AD domain, the authentication request can successfully occur without any need for user interaction.

This model grants the JavaScript application the ability to independently renew access tokens and even acquire new ones for a new API (provided that the user previously consented for them). This avoids the added burden of acquiring, maintaining, and protecting a high value artifact such as a refresh token. The artifact that makes the silent renewal possible, the Azure AD session cookie, is managed outside of the application. Another advantage of this approach is a user can sign out from Azure AD, using any of the applications signed into Azure AD, running in any of the browser tabs. This results in the deletion of the Azure AD session cookie, and the JavaScript application will automatically lose the ability to renew tokens for the signed out user.

Is the implicit grant suitable for my app?

The implicit grant presents more risks than other grants. However, the higher risk profile is largely due to the fact that it is meant to enable applications that execute active code, served by a remote resource to a browser. If you are planning an SPA architecture, have no backend components or intend to invoke a Web API via JavaScript, use of the implicit flow for token acquisition is recommended.

If your application is a native client, the implicit flow isn't a great fit. The absence of the Azure AD session cookie in the context of a native client deprives your application from the means of maintaining a long lived session. Which means your application will repeatedly prompt the user when obtaining access tokens for new resources.

If you are developing a Web application that includes a backend, and consuming an API from its backend code, the implicit flow is also not a good fit. Other grants give you far more power. For example, the OAuth2 client credentials grant provides the ability to obtain tokens that reflect the permissions assigned to the application itself, as opposed to user delegations. This means the client has the ability to maintain programmatic access to resources even when a user is not actively engaged in a session, and so on. Not only that, but such grants give higher security guarantees. For instance, access tokens never transit through the user browser, they don't risk being saved in the browser history, and so on. The client application can also perform strong authentication when requesting a token.

Authorize access to Azure Active Directory web applications using the OAuth 2.0 code grant flow

Azure Active Directory (Azure AD) uses OAuth 2.0 to enable you to authorize access to web applications and web APIs in your Azure AD tenant. This guide is language independent, and describes how to send and receive HTTP messages without using any of our open-source libraries.

The OAuth 2.0 authorization code flow is described in **section 4.1 of the OAuth 2.0 specification**[3]. It is used to perform authentication and authorization in most application types, including web apps and natively installed apps.

Register your application with your AD tenant

First, you need to register your application with your Azure Active Directory (Azure AD) tenant. This will give you an Application ID for your application, as well as enable it to receive tokens.

- Sign in to the Azure portal.
- Choose your Azure AD tenant by clicking on your account in the top right corner of the page, followed by clicking on the Switch Directory navigation and then select the appropriate tenant.
- --Skip this step, if you've only one Azure AD tenant under your account or if you've already selected the appropriate Azure AD tenant.
- In the left hand navigation pane, click on **Azure Active Directory**.
- Click on **App Registrations** and click on **New application registration**.
- Follow the prompts and create a new application. It doesn't matter if it is a web application or a native application for this tutorial.
- --For Web Applications, provide the Sign-On URL, which is the base URL of your app, where users can sign in e.g `http://localhost:12345`.
- --For Native Applications provide a Redirect URI, which Azure AD will use to return token responses. Enter a value specific to your application, .e.g `http://MyFirstAADApp`.
- Once you've completed registration, Azure AD will assign your application a unique client identifier, the **Application ID**. You need this value in the next sections, so copy it from the application page.
- To find your application in the Azure portal, click **App registrations**, and then click **View all applications**.

Request an authorization code

The authorization code flow begins with the client directing the user to the /authorize endpoint. In this request, the client indicates the permissions it needs to acquire from the user. You can get the OAuth 2.0 authorization endpoint for your tenant by selecting App registrations > Endpoints in the Azure portal.

```
// Line breaks for legibility only

https://login.microsoftonline.com/{tenant}/oauth2/authorize?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&response_type=code
&redirect_uri=http%3A%2F%2Flocalhost%3A12345
&response_mode=query
&resource=https%3A%2F%2Fservice.contoso.com%2F
&state=12345
```

3 https://tools.ietf.org/html/rfc6749#section-4.1

Parameter	Need	Description
tenant	required	The `{tenant}` value in the path of the request can be used to control who can sign into the application. The allowed values are tenant identifiers, for example, `8eaef023-2b34-4da1-9baa-8bc8c9d6a490` or `contoso.onmicrosoft.com` or `common` for tenant-independent tokens
client_id	required	The Application ID assigned to your app when you registered it with Azure AD. You can find this in the Azure Portal. Click **Azure Active Directory** in the services sidebar, click **App registrations**, and choose the application.
response_type	required	Must include `code` for the authorization code flow.
redirect_uri	recommended	The redirect_uri of your app, where authentication responses can be sent and received by your app. It must exactly match one of the redirect_uris you registered in the portal, except it must be url encoded. For native & mobile apps, you should use the default value of `urn:ietf:wg:oauth:2.0:oob`.
response_mode	optional	Specifies the method that should be used to send the resulting token back to your app. Can be `query`, `fragment`, or `form_post`. `query` provides the code as a query string parameter on your redirect URI. If you're requesting an ID token using the implicit flow, you cannot use `query` as specified in the OpenID spec. If you're requesting just the code, you can use `query`, `fragment`, or `form_post`. `form_post` executes a POST containing the code to your redirect URI. The default is `query` for a code flow.

Parameter	Need	Description
state	recommended	A value included in the request that is also returned in the token response. A randomly generated unique value is typically used for preventing cross-site request forgery attacks. The state is also used to encode information about the user's state in the app before the authentication request occurred, such as the page or view they were on.
resource	recommended	The App ID URI of the target web API (secured resource). To find the App ID URI, in the Azure Portal, click **Azure Active Directory**, click **Application registrations**, open the application's **Settings** page, then click **Properties**. It may also be an external resource like `https://graph.microsoft.com`. This is required in one of either the authorization or token requests. To ensure fewer authentication prompts place it in the authorization request to ensure consent is received from the user.
scope	**ignored**	For v1 Azure AD apps, scopes must be statically configured in the Azure Portal under the applications **Settings**, **Required Permissions**.

Parameter	Need	Description
prompt	optional	Indicate the type of user interaction that is required. Valid values are: *login*: The user should be prompted to reauthenticate. *select_account*: The user is prompted to select an account, interrupting single sign on. The user may select an existing signed-in account, enter their credentials for a remembered account, or choose to use a different account altogether. *consent*: User consent has been granted, but needs to be updated. The user should be prompted to consent. *admin_consent*: An administrator should be prompted to consent on behalf of all users in their organization
login_hint	optional	Can be used to pre-fill the username/email address field of the sign-in page for the user, if you know their username ahead of time. Often apps use this parameter during reauthentication, having already extracted the username from a previous sign-in using the `preferred_username` claim.
domain_hint	optional	Provides a hint about the tenant or domain that the user should use to sign in. The value of the domain_hint is a registered domain for the tenant. If the tenant is federated to an on-premises directory, AAD redirects to the specified tenant federation server.

Parameter	Need	Description
code_challenge_method	recommended	The method used to encode the `code_verifier` for the `code_challenge` parameter. Can be one of `plain` or `S256`. If excluded, `code_challenge` is assumed to be plaintext if `code_challenge` is included. Azure AAD v1.0 supports both `plain` and `S256`.
code_challenge	recommended	Used to secure authorization code grants via Proof Key for Code Exchange (PKCE) from a native or public client. Required if `code_challenge_method` is included.

Note: If the user is part of an organization, an administrator of the organization can consent or decline on the user's behalf, or permit the user to consent. The user is given the option to consent only when the administrator permits it.

At this point, the user is asked to enter their credentials and consent to the permissions requested by the app in the Azure Portal. Once the user authenticates and grants consent, Azure AD sends a response to your app at the `redirect_uri` address in your request with the code.

Successful response

A successful response could look like this:

```
GET  HTTP/1.1 302 Found
Location: http://localhost:12345/?code= AwABAAAAvPM1KaPlrEqdFSBzjqfTGBCm-
LdgfSTLEMPGYuNHSUYBrqqf_ZT_p5uEAEJJ_nZ3UmphWygRNy2C3jJ239gV_DBnZ2syeg-
95Ki-374WHUP-i3yIhv5i-7KU2CEoPXwURQp6IVYMw-DjAOzn7C3JCu5wpngXmbZKtJdWmiBzH-
pcO2aICJPu1KvJrDLDP20chJBXzVYJtkfjviLNNW7l7Y3ydcHDsBRKZc3GuMQanmcghXPyoDg-
41g8XbwPudVh7uCmUponBQpIhbuffFP_tbV8SNzsPoFz9CLpBCZagJVXeqWoYMPe2dSsPi-
LO9Alf_YIe5zpi-zY4C3aLw5g9at35eZTfNd0gBRpR5ojkMIcZZ6IgAA&session_
state=7B29111D-C220-4263-99AB-6F6E135D75EF&state=D79E5777-702E-4260-9A62-
37F75FF22CCE
```

Parameter	Description
admin_consent	The value is True if an administrator consented to a consent request prompt.
code	The authorization code that the application requested. The application can use the authorization code to request an access token for the target resource.
session_state	A unique value that identifies the current user session. This value is a GUID, but should be treated as an opaque value that is passed without examination.

Parameter	Description
state	If a state parameter is included in the request, the same value should appear in the response. It's a good practice for the application to verify that the state values in the request and response are identical before using the response. This helps to detect Cross-Site Request Forgery (CSRF) attacks against the client.

Error response

Error responses may also be sent to the `redirect_uri` so that the application can handle them appropriately.

```
GET http://localhost:12345/?
error=access_denied
&error_description=the+user+canceled+the+authentication
```

Parameter	Description
error	An error code value defined in Section 5.2 of the OAuth 2.0 Authorization Framework. The next table describes the error codes that Azure AD returns.
error_description	A more detailed description of the error. This message is not intended to be end-user friendly.
state	The state value is a randomly generated non-reused value that is sent in the request and returned in the response to prevent cross-site request forgery (CSRF) attacks.

Error codes for authorization endpoint errors

The following table describes the various error codes that can be returned in the `error` parameter of the error response.

Error Code	Description	Client Action
invalid_request	Protocol error, such as a missing required parameter.	Fix and resubmit the request. This is a development error, and is typically caught during initial testing.
unauthorized_client	The client application is not permitted to request an authorization code.	This usually occurs when the client application is not registered in Azure AD or is not added to the user's Azure AD tenant. The application can prompt the user with instruction for installing the application and adding it to Azure AD.

Error Code	Description	Client Action
access_denied	Resource owner denied consent	The client application can notify the user that it cannot proceed unless the user consents.
unsupported_response_type	The authorization server does not support the response type in the request.	Fix and resubmit the request. This is a development error, and is typically caught during initial testing.
server_error	The server encountered an unexpected error.	Retry the request. These errors can result from temporary conditions. The client application might explain to the user that its response is delayed due to a temporary error.
temporarily_unavailable	The server is temporarily too busy to handle the request.	Retry the request. The client application might explain to the user that its response is delayed due to a temporary condition.
invalid_resource	The target resource is invalid because it does not exist, Azure AD cannot find it, or it is not correctly configured.	This indicates the resource, if it exists, has not been configured in the tenant. The application can prompt the user with instruction for installing the application and adding it to Azure AD.

Use the authorization code to request an access token

Now that you've acquired an authorization code and have been granted permission by the user, you can redeem the code for an access token to the desired resource, by sending a POST request to the /token endpoint:

```
// Line breaks for legibility only

POST /{tenant}/oauth2/token HTTP/1.1
Host: https://login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded
grant_type=authorization_code
&client_id=2d4d11a2-f814-46a7-890a-274a72a7309e
&code=AwABAAAAvPM1KaPlrEqdFSBzjqfTGBCmLdgfSTLEMPGYuNHSUYBrqqf_ZT_p5uEAEJJ_
nZ3UmphWygRNy2C3jJ239gV_DBnZ2syeg95Ki-374WHUP-i3yIhv5i-7KU2CEoPXwURQp6IVY-
Mw-DjAOzn7C3JCu5wpngXmbZKtJdWmiBzHpcO2aICJPu1KvJrDLDP20chJBXzVYJtkfjviLNN-
W7l7Y3ydcHDsBRKZc3GuMQanmcghXPyoDg41g8XbwPudVh7uCmUponBQpIhbuffFP_tbV8SN-
zsPoFz9CLpBCZagJVXeqWoYMPe2dSsPiLO9Alf_YIe5zpi-zY4C3aLw5g9at35eZTfNd0g-
BRpR5ojkMIcZZ6IgAA
&redirect_uri=https%3A%2F%2Flocalhost%3A12345
&resource=https%3A%2F%2Fservice.contoso.com%2F
&client_secret=p@ssw0rd
```

```
//NOTE: client_secret only required for web apps
```

Parameter		Description
tenant	required	The `{tenant}` value in the path of the request can be used to control who can sign into the application. The allowed values are tenant identifiers, for example, `8eaef023-2b34-4da1-9baa-8bc8c9d6a490` or `contoso.onmicrosoft.com` or `common` for tenant-independent tokens
client_id	required	The Application Id assigned to your app when you registered it with Azure AD. You can find this in the Azure portal. The Application Id is displayed in the settings of the app registration.
grant_type	required	Must be `authorization_code` for the authorization code flow.
code	required	The `authorization_code` that you acquired in the previous section
redirect_uri	required	The same `redirect_uri` value that was used to acquire the `authorization_code`.
client_secret	required for web apps, not allowed for public clients	The application secret that you created in the Azure Portal for your app under **Keys**. It cannot be used in a native app (public client), because client_secrets cannot be reliably stored on devices. It is required for web apps and web APIs (all confidential clients), which have the ability to store the `client_secret` securely on the server side. The client_secret should be URL-encoded before being sent.

Parameter		Description
resource	recommended	The App ID URI of the target web API (secured resource). To find the App ID URI, in the Azure Portal, click **Azure Active Directory**, click **Application registrations**, open the application's **Settings** page, then click **Properties**. It may also be an external resource like `https://graph.microsoft.com`. This is required in one of either the authorization or token requests. To ensure fewer authentication prompts place it in the authorization request to ensure consent is received from the user. If in both the authorization request and the token request, the resource` parameters must match.
code_verifier	optional	The same code_verifier that was used to obtain the authorization_code. Required if PKCE was used in the authorization code grant request.

To find the App ID URI, in the Azure Portal, click **Azure Active Directory**, click **Application registrations**, open the application's **Settings** page, then click **Properties**.

Successful response

Azure AD returns an access token upon a successful response. To minimize network calls from the client application and their associated latency, the client application should cache access tokens for the token lifetime that is specified in the OAuth 2.0 response. To determine the token lifetime, use either the expires_in or expires_on parameter values.

If a web API resource returns an invalid_token error code, this might indicate that the resource has determined that the token is expired. If the client and resource clock times are different (known as a "time skew"), the resource might consider the token to be expired before the token is cleared from the client cache. If this occurs, clear the token from the cache, even if it is still within its calculated lifetime.

A successful response could look like this:

```
{
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ-
2ZEstZnl0aEV1THdqcHdBSk9NOW4tQSJ9.eyJhdWQiOiJodHRwczovL3NlcnZpY2UuY29udG9z-
by5jb20vIiwiaXNzIjoiaHR0cHM6Ly9zdHMud2luZG93cy5uZXQvN2ZlODE0NDctZGE-
1Ny00Mzg1LWJlY2ItNmRlNTdmMjE0NzdlLyIsImlhdCI6MTM4ODQ0MDg2My-
wibmJmIjoxMzg4NDQwODYzLCJleHAiOjEzODg0NDQ3NjMsInZlciI6IjEuMCIsInRpZCI6Ijd-
mZTgxNDQ3LWRhNTctNDM4NS1iZWNiLTZkZTU3ZjIxNDc3ZSIsIm9pZCI6IjY4Mzg5YWUyLTYyZ-
mEtNGIxOC05MWZlLTUzZGQxMDlkNzRmNSIsInVwbiI6ImZyYW5rbUBjb250b3NvLmNvbSIsIn-
VuaXF1ZV9uYW1lIjoiZnJhbmttQGNvbnRvc28uY29tIiwic3ViIjoiZGVOcUlqOUlPRTlQV0pX-
```

```
YkhzZnRYdDJFYWJQVmwwQ2o4UUFtZWZSTFY5OCIsImZhbWlseV9uYW1lIjoiTWlsbGVyIiwi-
Z2l2ZW5fbmFtZSI6IkZyYW5rIiwiYXBwaWQiOiIyZDRkMTFhMi1mODE0LTQ2YTctODkwYS0yNz-
RhNzJhNzMwOWUiLCJhcHBpZGFjciI6IjAiLCJzY3AiOiJ1c2VyX2ltcGVyc29uYXRpb24iL-
CJhY3IiOiIxIn0.
JZw8jC0gptZxVC-7l5sFkdnJgP3_tRjeQEPgUn28XctVe3QqmheLZw7QVZDPCyGycDWBaqy-
7FLpSekET_BftDkewRhyHk9FW_KeEz0ch2c3i08NGNDbr6XYGVayNuSesYk5Aw_p3ICRlU-
V1bqEwk-Jkzs9EEkQg4hbefqJS6yS1HoV_2EsEhpd_wCQpxK89WPs3hLYZETRJtG5kvCCEO-
vSHXmDE6eTHGTnEgsIk--UlPe275Dvou4gEAwLofhLDQbMSjnlV5VLsjimNBVcSRFShoxmQwB-
JR_b2011Y5IuD6St5zPnzruBbZYkGNurQK63TJPWmRd3mbJsGM0mf3CUQ",
  "token_type": "Bearer",
  "expires_in": "3600",
  "expires_on": "1388444763",
  "resource": "https://service.contoso.com/",
  "refresh_token": "AwABAAAAvPM1KaPlrEqdFSBzjqfTGAMxZGUTdM0t4B4rTfgV29gh-
DOHRc2B-C_hHeJaJICqjZ3mY2b_YNqmf9SoAylD1PycGCB90xzZeEDg6oBzOIPfYsbDWNf621p-
Ko2Q3GGTHYlmNfwoc-OlrxK69hkha2CF12azM_NYhgO668yfcUl4VBbiSHZyd1NVZG5QTIOc-
bObu3qnLutbpadZGAxqjIbMkQ2bQS09fTrjMBtDE3D6kSMIodpCecoANon9b0LATkpitim-
VCrl-NyfN3oyG4ZCWu18M9-vEou4Sq-1oMDzExgAf61noxzkNiaTecM-Ve5cq6wHqYQjfV9DO-
z4lbceuYCAA",
  "scope": "https%3A%2F%2Fgraph.microsoft.com%2Fmail.read",
  "id_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhdWQiOiIyZDRkMTFhMi-
1mODE0LTQ2YTctODkwYS0yNzRhNzJhNzMwOWUiLCJpc3MiOiJodHRwczovL3N0cy53aW5kb3d-
zLm5ldC83ZmU4MTQ0Ny1kYTU3LTQzODUtYmVjYi02ZGU1N2YyMTQ3N2U-
vIiwiaWF0IjoxMzg4NDQwODYzLCJuYmYiOjEzODg0NDA4NjMsImV4cCI6MTM4ODQ0NDc2Mywid-
mVyIjoiMS4wIiwidGlkIjoiN2ZlODE0NDctZGE1Ny00Mzg1LWJlY2ItNmRlNTdmMjE0NzdlIi-
wib2lkIjoiNjgzODlhZTItNjJmYS00YjE4LTkxZmUtNTNkZDEwOWQ3NGY1IiwidXBuIjoiZnJh-
bmttQGNvbnRvc28uY29tIiwidW5pcXVlX25hbWUiOiJmcmFua21AY29udG9zby5jb20iLCJzd-
WIiOiJKV3ZZZENXUGhobHBTMVpzZjd5WVV4U2hVd3RVbTV5elBtd18talgzZkhZIiwiZm-
FtaWx5X25hbWUiOiJNaWxsZXIiLCJnaXZlbl9uYW1lIjoiRnJhbmsifQ."
}
```


Parameter	Description
access_token	The requested access token as a signed JSON Web Token (JWT). The app can use this token to authenticate to the secured resource, such as a web API.
token_type	Indicates the token type value. The only type that Azure AD supports is Bearer.
expires_in	How long the access token is valid (in seconds).
expires_on	The time when the access token expires. The date is represented as the number of seconds from 1970-01-01T0:0:0Z UTC until the expiration time. This value is used to determine the lifetime of cached tokens.
resource	The App ID URI of the web API (secured resource).
scope	Impersonation permissions granted to the client application. The default permission is `user_impersonation`. The owner of the secured resource can register additional values in Azure AD.

Parameter	Description
refresh_token	An OAuth 2.0 refresh token. The app can use this token to acquire additional access tokens after the current access token expires. Refresh tokens are long-lived, and can be used to retain access to resources for extended periods of time.
id_token	An unsigned JSON Web Token (JWT) representing an ID token. The app can base64Url decode the segments of this token to request information about the user who signed in. The app can cache the values and display them, but it should not rely on them for any authorization or security boundaries.

Error response

The token issuance endpoint errors are HTTP error codes, because the client calls the token issuance endpoint directly. In addition to the HTTP status code, the Azure AD token issuance endpoint also returns a JSON document with objects that describe the error.

A sample error response could look like this:

```
{
  "error": "invalid_grant",
  "error_description": "AADSTS70002: Error validating credentials.
AADSTS70008: The provided authorization code or refresh token is expired.
Send a new interactive authorization request for this user and resource.\r\
nTrace ID: 3939d04c-d7ba-42bf-9cb7-1e5854cdce9e\r\nCorrelation ID:
a8125194-2dc8-4078-90ba-7b6592a7f231\r\nTimestamp: 2016-04-11 18:00:12Z",
  "error_codes": [
    70002,
    70008
  ],
  "timestamp": "2016-04-11 18:00:12Z",
  "trace_id": "3939d04c-d7ba-42bf-9cb7-1e5854cdce9e",
  "correlation_id": "a8125194-2dc8-4078-90ba-7b6592a7f231"
}
```

Parameter	Description
`error`	An error code string that can be used to classify types of errors that occur, and can be used to react to errors.
`error_description`	A specific error message that can help a developer identify the root cause of an authentication error.
`error_codes`	A list of STS-specific error codes that can help in diagnostics.
`timestamp`	The time at which the error occurred.
`trace_id`	A unique identifier for the request that can help in diagnostics.

Parameter	Description
`correlation_id`	A unique identifier for the request that can help in diagnostics across components.

Error codes for token endpoint errors

Error Code	Description	Client Action
invalid_request	Protocol error, such as a missing required parameter.	Fix and resubmit the request
invalid_grant	The authorization code is invalid or has expired.	Try a new request to the `/authorize` endpoint
unauthorized_client	The authenticated client is not authorized to use this authorization grant type.	This usually occurs when the client application is not registered in Azure AD or is not added to the user's Azure AD tenant. The application can prompt the user with instruction for installing the application and adding it to Azure AD.
invalid_client	Client authentication failed.	The client credentials are not valid. To fix, the application administrator updates the credentials.
unsupported_grant_type	The authorization server does not support the authorization grant type.	Change the grant type in the request. This type of error should occur only during development and be detected during initial testing.
invalid_resource	The target resource is invalid because it does not exist, Azure AD cannot find it, or it is not correctly configured.	This indicates the resource, if it exists, has not been configured in the tenant. The application can prompt the user with instruction for installing the application and adding it to Azure AD.
interaction_required	The request requires user interaction. For example, an additional authentication step is required.	Instead of a non-interactive request, retry with an interactive authorization request for the same resource.
temporarily_unavailable	The server is temporarily too busy to handle the request.	Retry the request. The client application might explain to the user that its response is delayed due to a temporary condition.

Use the access token to access the resource

Now that you've successfully acquired an `access_token`, you can use the token in requests to Web APIs, by including it in the `Authorization` header.

Sample request

```
GET /data HTTP/1.1
Host: service.contoso.com
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ-
2ZEstZnl0aEV1THdqcHdBSk9NOW4tQSJ9.eyJhdWQiOiJodHRwczovL3NlcnZpY2UuY29udG9z-
by5jb20vIiwiaXNzIjoiaHR0cHM6Ly9zdHMud2luZG93cy5uZXQvN2ZlODE0NDctZGE-
1Ny00Mzg1LWJlY2ItNmRlNTdmMjE0NzdlLyIsImlhdCI6MTM4ODQ0MDg2My-
wibmJmIjoxMzg4NDQwODYzLCJleHAiOjEzODg0NDQ3NjMsInZlciI6IjEuMCIsInRpZCI6Ijd-
mZTgxNDQ3LWRhNTctNDM4NS1iZWNiLTZkZTU3ZjIxNDc3ZSIsIm9pZCI6IjY4Mzg5YWUyLTYyZ-
mEtNGIxOC05MWZlLTUzZGQxMDlkNzRmNSIsInVwbiI6ImZyYW5rbUBjb250b3NvLmNvbSIsIn-
VuaXF1ZV9uYW1lIjoiZnJhbmttQGNvbnRvc28uY29tIiwic3ViIjoiZGVOcUlqOUlPRTlQV0pX-
YkhzZnRYdDJFYWJQVmwwQ2o4UUFtZWZSTFY5OCIsImZhbWlseV9uYW1lIjoiTWlsbGVyIiwi-
Z2l2ZW5fbmFtZSI6IkZyYW5rIiwiYXBwaWQiOiIyZDRkMTFhMi1mODE0LTQ2YTctODkwYS0yNz-
RhNzJhNzMwOWUiLCJhcHBpZGFjciI6IjAiLCJzY3AiOiJ1c2VyX2ltcGVyc29uYXRpb24iL-
CJhY3IiOiIxIn0.
JZw8jC0gptZxVC-7l5sFkdnJgP3_tRjeQEPgUn28XctVe3QqmheLZw7QVZDPCyGycDWBaqy-
7FLpSekET_BftDkewRhyHk9FW_KeEz0ch2c3i08NGNDbr6XYGVayNuSesYk5Aw_p3ICRlU-
V1bqEwk-Jkzs9EEkQg4hbefqJS6yS1HoV_2EsEhpd_wCQpxK89WPs3hLYZETRJtG5kvCCEO-
vSHXmDE6eTHGTnEgsIk--UlPe275Dvou4gEAwLofhLDQbMSjnlV5VLsjimNBVcSRFShoxmQwB-
JR_b2011Y5IuD6St5zPnzruBbZYkGNurQK63TJPWmRd3mbJsGM0mf3CUQ
```

Error Response

Secured resources that implement RFC 6750 issue HTTP status codes. If the request does not include authentication credentials or is missing the token, the response includes an `WWW-Authenticate` header. When a request fails, the resource server responds with the HTTP status code and an error code.

The following is an example of an unsuccessful response when the client request does not include the bearer token:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer authorization_uri="https://login.microsoftonline.
com/contoso.com/oauth2/authorize",  error="invalid_token",  error_descrip-
tion="The access token is missing.",
```

Parameter	Description
`authorization_uri`	The URI (physical endpoint) of the authorization server. This value is also used as a lookup key to get more information about the server from a discovery endpoint. The client must validate that the authorization server is trusted. When the resource is protected by Azure AD, it is sufficient to verify that the URL begins with `https://login.microsoftonline.com` or another hostname that Azure AD supports. A tenant-specific resource should always return a tenant-specific authorization URI.

Parameter	Description
`error`	An error code value defined in Section 5.2 of the OAuth 2.0 Authorization Framework.
`error_description`	A more detailed description of the error. This message is not intended to be end-user friendly.
`resource_id`	Returns the unique identifier of the resource. The client application can use this identifier as the value of the `resource` parameter when it requests a token for the resource. It is important for the client application to verify this value, otherwise a malicious service might be able to induce an elevation-of-privileges attack. The recommended strategy for preventing an attack is to verify that the `resource_id` matches the base of the web API URL that being accessed. For example, if `https://service.contoso.com/data` is being accessed, the `resource_id` can be `htttps://service.contoso.com/`. The client application must reject a resource_id that does not begin with the base URL unless there is a reliable alternate way to verify the id.

Refreshing the access tokens

Access Tokens are short-lived and must be refreshed after they expire to continue accessing resources. You can refresh the access_token by submitting another POST request to the /token endpoint, but this time providing the refresh_token instead of the code. Refresh tokens are valid for all resources that your client has already been given consent to access - thus, a refresh token issued on a request for `resource=https://graph.microsoft.com` can be used to request a new access token for `resource=https://contoso.com/api`.

Refresh tokens do not have specified lifetimes. Typically, the lifetimes of refresh tokens are relatively long. However, in some cases, refresh tokens expire, are revoked, or lack sufficient privileges for the desired action. Your application needs to expect and handle errors returned by the token issuance endpoint correctly.

When you receive a response with a refresh token error, discard the current refresh token and request a new authorization code or access token. In particular, when using a refresh token in the Authorization Code Grant flow, if you receive a response with the interaction_required or invalid_grant error codes, discard the refresh token and request a new authorization code.

A sample request to the tenant-specific endpoint (you can also use the common endpoint) to get a new access token using a refresh token looks like this:

```
// Line breaks for legibility only

POST /{tenant}/oauth2/token HTTP/1.1
Host: https://login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded
```

```
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&refresh_token=OAAABAAAAiL9Kn2Z27UubvWFPbm0gLWQJVzCTE9UkP3pSx1aXxUjq...
&grant_type=refresh_token
&resource=https%3A%2F%2Fservice.contoso.com%2F
&client_secret=JqQX2PNo9bpM0uEihUPzyrh    // NOTE: Only required for web
apps
```

Successful response

A successful token response will look like:

```
{
  "token_type": "Bearer",
  "expires_in": "3600",
  "expires_on": "1460404526",
  "resource": "https://service.contoso.com/",
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ-
2ZEstZnl0aEV1THdqcHdBSk9NOW4tQSJ9.eyJhdWQiOiJodHRwczovL3NlcnZpY2UuY29udG9z-
by5jb20vIiwiaXNzIjoiaHR0cHM6Ly9zdHMud2luZG93cy5uZXQvN2ZlODE0NDctZGE-
1Ny00Mzg1LWJlY2ItNmRlNTdmMjE0NzdlLyIsImlhdCI6MTM4ODQ0MDg2My-
wibmJmIjoxMzg4NDQwODYzLCJleHAiOjEzODg0NDQ3NjMsInZlciI6IjEuMCIsInRpZCI6Ijd-
mZTgxNDQ3LWRhNTctNDM4NS1iZWNiLTZkZTU3ZjIxNDc3ZSIsIm9pZCI6IjY4Mzg5YWUyLTYyZ-
mEtNGIxOC05MWZlLTUzZGQxMDlkNzRmNSIsInVwbiI6ImZyYW5rbUBjb250b3NvLmNvbSIsIn-
VuaXF1ZV9uYW1lIjoiZnJhbmttQGNvbnRvc28uY29tIiwic3ViIjoiZGVOcUlqOUlPRTlQV0pX-
YkhzZnRYdDJFYWJQVmwwQ2o4UUFtZWZSTFY5OCIsImZhbWlseV9uYW1lIjoiTWlsbGVyIiwi-
Z2l2ZW5fbmFtZSI6IkZyYW5rIiwiYXBwaWQiOiIyZDRkMTFhMi1mODE0LTQ2YTctODkwYS0yNz-
RhNzJhNzMwOWUiLCJhcHBpZGFjciI6IjAiLCJzY3AiOiJ1c2VyX2ltcGVyc29uYXRpb24iL-
CJhY3IiOiIxIn0.
JZw8jC0gptZxVC-7l5sFkdnJgP3_tRjeQEPgUn28XctVe3QqmheLZw7QVZDPCyGycDWBaqy-
7FLpSekET_BftDkewRhyHk9FW_KeEz0ch2c3i08NGNDbr6XYGVayNuSesYk5Aw_p3ICRlU-
V1bqEwk-Jkzs9EEkQg4hbefqJS6yS1HoV_2EsEhpd_wCQpxK89WPs3hLYZETRJtG5kvCCEO-
vSHXmDE6eTHGTnEgsIk--UlPe275Dvou4gEAwLofhLDQbMSjnlV5VLsjimNBVcSRFShoxmQwB-
JR_b2011Y5IuD6St5zPnzruBbZYkGNurQK63TJPWmRd3mbJsGM0mf3CUQ",
  "refresh_token": "AwABAAAAv YNqmf9SoAylD1PycGCB90xzZeEDg6oBzOIPfYsbDWNf-
621pKo2Q3GGTHYlmNfwoc-OlrxK69hkha2CF12azM_NYhgO668yfcUl4VBbiSHZyd1NVZG5QTI-
OcbObu3qnLutbpadZGAxqjIbMkQ2bQS09fTrjMBtDE3D6kSMIodpCecoANon9b0LATkpitim-
VCrl
PM1KaPlrEqdFSBzjqfTGAMxZGUTdM0t4B4rTfgV29ghDOHRc2B-C_hHeJaJICqjZ3mY2b_YNqm-
f9SoAylD1PycGCB90xzZeEDg6oBzOIPfYsbDWNf621pKo2Q3GGTHYlmNfwoc-OlrxK69hkha2C-
F12azM_NYhgO668yfmVCrl-NyfN3oyG4ZCWu18M9-vEou4Sq-1oMDzExgAf61noxz-
kNiaTecM-Ve5cq6wHqYQjfV9DOz4lbceuYCAA"
}
```

Parameter	Description
`token_type`	The token type. The only supported value is bearer.
`expires_in`	The remaining lifetime of the token in seconds. A typical value is 3600 (one hour).

Parameter	Description
expires_on	The date and time on which the token expires. The date is represented as the number of seconds from 1970-01-01T0:0:0Z UTC until the expiration time.
resource	Identifies the secured resource that the access token can be used to access.
scope	Impersonation permissions granted to the native client application. The default permission is user_impersonation. The owner of the target resource can register alternate values in Azure AD.
access_token	The new access token that was requested.
refresh_token	A new OAuth 2.0 refresh_token that can be used to request new access tokens when the one in this response expires.

Error response

A sample error response could look like this:

```
{
  "error": "invalid_resource",
  "error_description": "AADSTS50001: The application named https://foo.
microsoft.com/mail.read was not found in the tenant named 295e01fc-0c56-
4ac3-ac57-5d0ed568f872. This can happen if the application has not been
installed by the administrator of the tenant or consented to by any user in
the tenant. You might have sent your authentication request to the wrong
tenant.\r\nTrace ID: ef1f89f6-a14f-49de-9868-61bd4072f0a9\r\nCorrelation
ID: b6908274-2c58-4e91-aea9-1f6b9c99347c\r\nTimestamp: 2016-04-11
18:59:01Z",
  "error_codes": [
    50001
  ],
  "timestamp": "2016-04-11 18:59:01Z",
  "trace_id": "ef1f89f6-a14f-49de-9868-61bd4072f0a9",
  "correlation_id": "b6908274-2c58-4e91-aea9-1f6b9c99347c"
}
```

Parameter	Description
error	An error code string that can be used to classify types of errors that occur, and can be used to react to errors.
error_description	A specific error message that can help a developer identify the root cause of an authentication error.
error_codes	A list of STS-specific error codes that can help in diagnostics.
timestamp	The time at which the error occurred.
trace_id	A unique identifier for the request that can help in diagnostics.

Parameter	Description
`correlation_id`	A unique identifier for the request that can help in diagnostics across components.

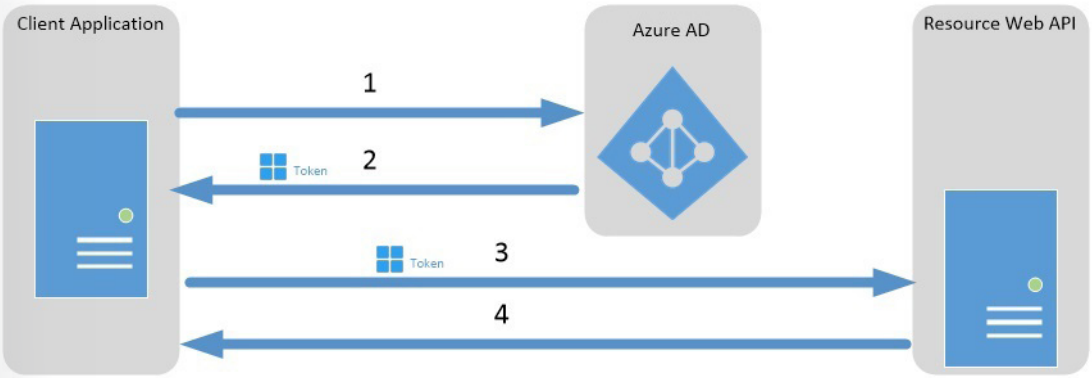
For a description of the error codes and the recommended client action, see **"Error codes for token endpoint errors"** list in the **"Use the authorization code to request an access token"** section above.

Service to service calls using client credentials

The OAuth 2.0 Client Credentials Grant Flow permits a web service (*confidential client*) to use its own credentials instead of impersonating a user, to authenticate when calling another web service. In this scenario, the client is typically a middle-tier web service, a daemon service, or web site. For a higher level of assurance, Azure AD also allows the calling service to use a certificate (instead of a shared secret) as a credential.

Client credentials grant flow diagram

The following diagram explains how the client credentials grant flow works in Azure Active Directory (Azure AD).



1. The client application authenticates to the Azure AD token issuance endpoint and requests an access token.
2. The Azure AD token issuance endpoint issues the access token.
3. The access token is used to authenticate to the secured resource.
4. Data from the secured resource is returned to the client application.

Register the Services in Azure AD

Register both the calling service and the receiving service in Azure Active Directory (Azure AD). For detailed instructions, see **Integrating applications with Azure Active Directory**[4].

Request an Access Token

To request an access token, use an HTTP POST to the tenant-specific Azure AD endpoint.

```
https://login.microsoftonline.com/<tenant id>/oauth2/token
```

4 https://docs.microsoft.com/en-us/azure/active-directory/develop/quickstart-v1-integrate-apps-with-azure-ad

Service-to-service access token request

There are two cases depending on whether the client application chooses to be secured by a shared secret, or a certificate.

First case: Access token request with a shared secret

When using a shared secret, a service-to-service access token request contains the following parameters:

Parameter		Description
grant_type	required	Specifies the requested grant type. In a Client Credentials Grant flow, the value must be **client_credentials**.
client_id	required	Specifies the Azure AD client id of the calling web service. To find the calling application's client ID, in the Azure portal, click **Azure Active Directory**, click **App registrations**, click the application. The `client_id` is the *Application ID*.
client_secret	required	Enter a key registered for the calling web service or daemon application in Azure AD. To create a key, in the Azure portal, click **Azure Active Directory**, click **App registrations**, click the application, click **Settings**, click **Keys**, and **add a Key**. URL-encode this secret when providing it.
resource	required	Enter the App ID URI of the receiving web service. To find the App ID URI, in the Azure portal, click **Azure Active Directory**, click **App registrations**, click the service application, and then click **Settings** and **Properties**.

Example

The following HTTP POST requests an access token for the `https://service.contoso.com/` web service. The `client_id` identifies the web service that requests the access token.

```
POST /contoso.com/oauth2/token HTTP/1.1
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&client_id=625bc9f6-3bf6-4b6d-94ba-e97c-
f07a22de&client_secret=qkDwDJlDfig2IpeuUZYKH1Wb8q1V0ju6sILxQQqhJ+s=&re-
```

```
source=https%3A%2F%2Fservice.contoso.com%2F
```

Second case: Access token request with a certificate

A service-to-service access token request with a certificate contains the following parameters:

Parameter		Description
grant_type	required	Specifies the requested response type. In a Client Credentials Grant flow, the value must be **client_credentials**.
client_id	required	Specifies the Azure AD client id of the calling web service. To find the calling application's client ID, in the Azure portal, click **Azure Active Directory**, click **App registrations**, click the application. The `client_id` is the *Application ID*.
client_assertion_type	required	The value must be `urn:ietf:params:oauth:client-assertion-type:jwt-bearer`
client_assertion	required	An assertion (a JSON Web Token) that you need to create and sign with the certificate you registered as credentials for your application. Read about **certificate credentials** *(https://docs.microsoft.com/en-us/azure/active-directory/develop/active-directory-certificate-credentials)* to learn how to register your certificate and the format of the assertion.
resource	required	Enter the App ID URI of the receiving web service. To find the App ID URI, in the Azure portal, click **Azure Active Directory**, click **App registration**s, click the service application, and then click **Settings** and **Properties**.

Notice that the parameters are almost the same as in the case of the request by shared secret except that the `client_secret` parameter is replaced by **two** parameters: a `client_assertion_type` and `client_assertion`.

Example

The following HTTP POST requests an access token for the `https://service.contoso.com/` web service with a certificate. The `client_id` identifies the web service that requests the access token.

```
POST /<tenant_id>/oauth2/token HTTP/1.1
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

resource=https%3A%2F%contoso.onmicrosoft.com%2Ff-
c7664b4-cdd6-43e1-9365-c2e1c4e1b3bf&client_id=97e-
0a5b7-d745-40b6-94fe-5f77d35c6e05&client_assertion_type=urn%3Aietf%3Apar-
ams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer&client_assertion=eyJhbGci-
OiJSUzI1NiIsIng1dCI6Imd4OHRHeXN5amNScUtqRlBuZDdSRnd2d1pJMCJ9.eyJ{a lot of
characters here}M8U3bSUKKJDEg&grant_type=client_credentials
```

Service-to-Service Access Token Response

A success response contains a JSON OAuth 2.0 response with the following parameters:

Parameter	Description
access_token	The requested access token. The calling web service can use this token to authenticate to the receiving web service.
token_type	Indicates the token type value. The only type that Azure AD supports is **Bearer**. For more information about bearer tokens, see **The OAuth 2.0 Authorization Framework: Bearer Token Usage (RFC 6750)** *(https://www.rfc-editor.org/rfc/rfc6750.txt)*.
expires_in	How long the access token is valid (in seconds).
expires_on	The time when the access token expires. The date is represented as the number of seconds from 1970-01-01T0:0:0Z UTC until the expiration time. This value is used to determine the lifetime of cached tokens.
not_before	The time from which the access token becomes usable. The date is represented as the number of seconds from 1970-01-01T0:0:0Z UTC until time of validity for the token.
resource	The App ID URI of the receiving web service.

Example of response

The following example shows a success response to a request for an access token to a web service.

```
{
"access_token":"eyJhbGciOiJSUzI1NiIsIng1dCI6IjdkRC1nZWNOZ1gxWmY3R0xrT3Zw-
T0IyZGNWQSIsInR5cCI6IkpXVCJ9.eyJhdWQiOiJodHRwczovL3NlcnZpY2UuY29udG9zby5jb-
20vIiwiaXNzIjoiaHR0cHM6Ly9zdHMud2luZG93cy5uZXQvN2ZlODE0NDctZGE1Ny00Mzg1LW-
```

```
JlY2ItNmRlNTdmMjE0NzdlLyIsImlhdCI6MTM4ODQ0ODI2NywibmJmIjoxMzg4NDQ4MjY3L-
CJleHAiOjEzODg0NTIxNjcsInZlciI6IjEuMCIsInRpZCI6IjdmZTgxNDQ3LWRhNTctNDM4N-
S1iZWNiLTZkZTU3ZjIxNDc3ZSIsIm9pZCI6ImE5OTE5MTYyLTkyMTctNDlkYS1hZTIyLWYxMT-
M3YzI1Y2RlYSIsInN1YiI6ImE5OTE5MTYyLTkyMTctNDlkYS1hZTIyLWYxMTM3YzI1Y2RlYSI-
sImlkcCI6Imh0dHBzOi8vc3RzLndpbmRvd3MubmV0LzdmZTgxNDQ3LWRhNTctNDM4NS1i-
ZWNiLTZkZTU3ZjIxNDc3ZS8iLCJhcHBpZCI6ImQxN2QxNWJjLWM1NzYtNDFlNS05MjdmLWRiN-
WYzMGRkNThmMSIsImFwcGlkYWNyIjoiMSJ9.
aqtfJ7G37CpKV901Vm9sGiQhde0WMg6luYJR4wuNR2ffaQsVPPpKirM5rbc6o5CmW1OtmaAIdwD-
cL6i9ZT9ooIIicSRrjCYMYWHX08ip-tj-uWUihGztI02xKdWiycItpWiHxapQm0a8T-
i1CWRjJghORC1B1-fah_yWx6Cjuf4QE8xJcu-ZHX0pVZNPX22PHYV5Km-vPTq2HtIqd-
boKyZy3Y4y3geOrRIFElZYoqjqSv5q9Jgtj5ERsNQIjefpyxW3EwPtFqMcDm4ebiAEpoEWRN-
4QYOMxnC9OUBeG9oLA0lTfmhgHLAtvJogJcYFzwngTsVo6HznsvPWy7UP3MINA",
"token_type":"Bearer",
"expires_in":"3599",
"expires_on":"1388452167",
"resource":"https://service.contoso.com/"
}
```

Implement managed identities for Azure resources

Managed identities for Azure resources overview

Note: Managed identities for Azure resources is a feature of Azure Active Directory. Each of the Azure services that support managed identities for Azure resources are subject to their own timeline. Make sure you review the **availability**[5] status of managed identities for your resource and **known issues**[6] before you begin.

A common challenge when building cloud applications is how to manage the credentials in your code for authenticating to cloud services. Keeping the credentials secure is an important task. Ideally, the credentials never appear on developer workstations and aren't checked into source control. Azure Key Vault provides a way to securely store credentials, secrets, and other keys, but your code has to authenticate to Key Vault to retrieve them.

The managed identities for Azure resources feature in Azure Active Directory (Azure AD) solves this problem. The feature provides Azure services with an automatically managed identity in Azure AD. You can use the identity to authenticate to any service that supports Azure AD authentication, including Key Vault, without any credentials in your code.

The managed identities for Azure resources feature is free with Azure AD for Azure subscriptions. There's no additional cost.

Note: Managed identities for Azure resources is the new name for the service formerly known as Managed Service Identity (MSI).

Terminology

The following terms are used throughout the managed identities for Azure resources documentation set:

- **Client id** - a unique identifier generated by Azure AD that is tied to an application and service principal during its initial provisioning.
- **Principal id** - the object id of the service principal object for your managed identity that is used to grant role based access to an Azure resource.
- **Azure Instance Metadata Service (IMDS)** - a REST Endpoint accessible to all IaaS VMs created via the Azure Resource Manager. The endpoint is available at a well-known non-routable IP address (169.254.169.254) that can be accessed only from within the VM.

How the managed identities for Azure resources works

There are two types of managed identities:

- A **system-assigned managed identity** is enabled directly on an Azure service instance. When the identity is enabled, Azure creates an identity for the instance in the Azure AD tenant that's trusted by the subscription of the instance. After the identity is created, the credentials are provisioned onto the instance. The lifecycle of a system-assigned identity is directly tied to the Azure service instance that

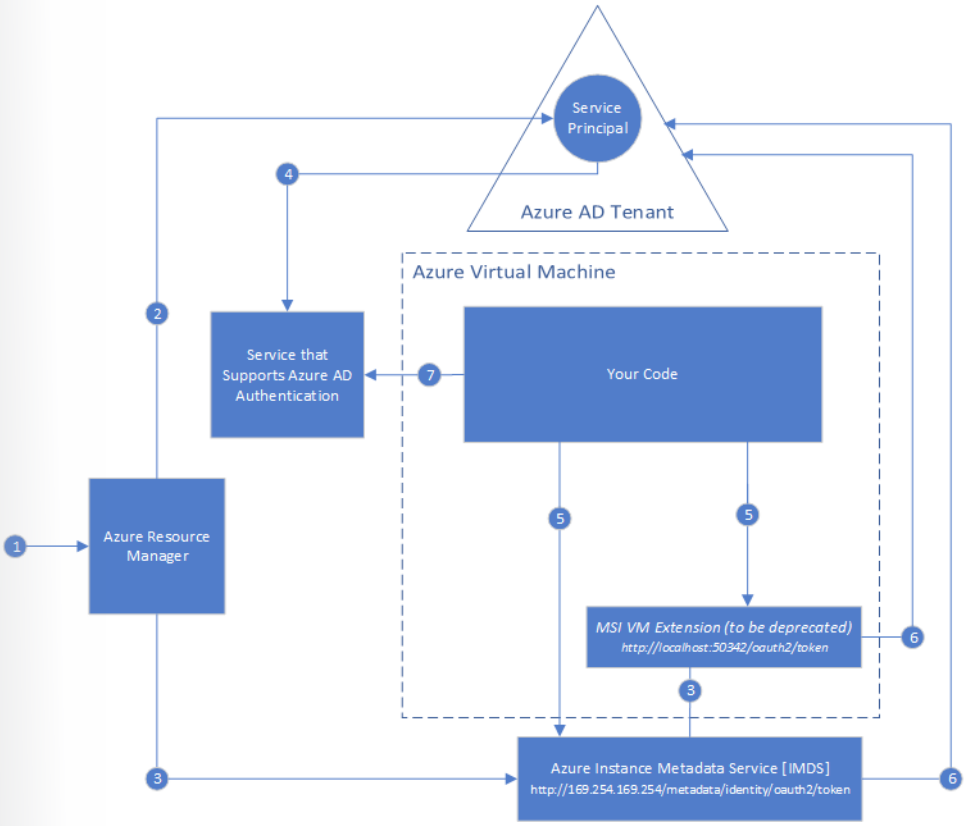
5 https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/services-support-msi
6 https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/known-issues

it's enabled on. If the instance is deleted, Azure automatically cleans up the credentials and the identity in Azure AD.

- A **user-assigned managed identity** is created as a standalone Azure resource. Through a create process, Azure creates an identity in the Azure AD tenant that's trusted by the subscription in use. After the identity is created, the identity can be assigned to one or more Azure service instances. The lifecycle of a user-assigned identity is managed separately from the lifecycle of the Azure service instances to which it's assigned.

Your code can use a managed identity to request access tokens for services that support Azure AD authentication. Azure takes care of rolling the credentials that are used by the service instance.

The following diagram shows how managed service identities work with Azure virtual machines (VMs):



How a system-assigned managed identity works with an Azure VM

1. Azure Resource Manager receives a request to enable the system-assigned managed identity on a VM.
2. Azure Resource Manager creates a service principal in Azure AD for the identity of the VM. The service principal is created in the Azure AD tenant that's trusted by the subscription.
3. Azure Resource Manager configures the identity on the VM:
   - Updates the Azure Instance Metadata Service identity endpoint with the service principal client ID and certificate.

- Provisions the VM extension (planned for deprecation in January 2019), and adds the service principal client ID and certificate. (This step is planned for deprecation.)

4. After the VM has an identity, use the service principal information to grant the VM access to Azure resources. To call Azure Resource Manager, use role-based access control (RBAC) in Azure AD to assign the appropriate role to the VM service principal. To call Key Vault, grant your code access to the specific secret or key in Key Vault.
5. Your code that's running on the VM can request a token from two endpoints that are accessible only from within the VM:
   - Azure Instance Metadata Service identity endpoint (recommended): `http://169.254.169.254/metadata/identity/oauth2/token`
     - The resource parameter specifies the service to which the token is sent. To authenticate to Azure Resource Manager, use `resource=https://management.azure.com/`.
     - API version parameter specifies the IMDS version, use `api-version=2018-02-01` or greater.
   - VM extension endpoint (planned for deprecation in January 2019): `http://localhost:50342/oauth2/token`
     - The resource parameter specifies the service to which the token is sent. To authenticate to Azure Resource Manager, use `resource=https://management.azure.com/`.
6. A call is made to Azure AD to request an access token (as specified in step 5) by using the client ID and certificate configured in step 3. Azure AD returns a JSON Web Token (JWT) access token.
7. Your code sends the access token on a call to a service that supports Azure AD authentication.

How a user-assigned managed identity works with an Azure VM

1. Azure Resource Manager receives a request to create a user-assigned managed identity.
2. Azure Resource Manager creates a service principal in Azure AD for the user-assigned managed identity. The service principal is created in the Azure AD tenant that's trusted by the subscription.
3. Azure Resource Manager receives a request to configure the user-assigned managed identity on a VM:
   - Updates the Azure Instance Metadata Service identity endpoint with the user-assigned managed identity service principal client ID and certificate.
   - Provisions the VM extension, and adds the user-assigned managed identity service principal client ID and certificate. (This step is planned for deprecation.)
4. After the user-assigned managed identity is created, use the service principal information to grant the identity access to Azure resources. To call Azure Resource Manager, use RBAC in Azure AD to assign the appropriate role to the service principal of the user-assigned identity. To call Key Vault, grant your code access to the specific secret or key in Key Vault.
5. **Note:** You can also do this step before step 3.

6. Your code that's running on the VM can request a token from two endpoints that are accessible only from within the VM:
   - Azure Instance Metadata Service identity endpoint (recommended): `http://169.254.169.254/metadata/identity/oauth2/token`
     - The resource parameter specifies the service to which the token is sent. To authenticate to Azure Resource Manager, use `resource=https://management.azure.com/`.
     - The client ID parameter specifies the identity for which the token is requested. This value is required for disambiguation when more than one user-assigned identity is on a single VM.
     - The API version parameter specifies the Azure Instance Metadata Service version. Use `api-version=2018-02-01` or higher.
   - VM extension endpoint (planned for deprecation in January 2019): `http://localhost:50342/oauth2/token`
     - The resource parameter specifies the service to which the token is sent. To authenticate to Azure Resource Manager, use `resource=https://management.azure.com/`.
     - The client ID parameter specifies the identity for which the token is requested. This value is required for disambiguation when more than one user-assigned identity is on a single VM.
7. A call is made to Azure AD to request an access token (as specified in step 5) by using the client ID and certificate configured in step 3. Azure AD returns a JSON Web Token (JWT) access token.
8. Your code sends the access token on a call to a service that supports Azure AD authentication.

Configure managed identities for Azure resources on an Azure VM using Azure CLI

In this article, you learn how to enable and disable system and user-assigned managed identities for an Azure Virtual Machine (VM), using the Azure CLI.

System-assigned managed identity

In this section, you learn how to enable and disable the system-assigned managed identity on an Azure VM using Azure CLI.

To create an Azure VM with the system-assigned managed identity enabled,your account needs the Virtual Machine Contributor role assignment. No additional Azure AD directory role assignments are required.

Enable system-assigned managed identity during creation of an Azure VM

1. If you're using the Azure CLI in a local console, first sign in to Azure using `az login`. Use an account that is associated with the Azure subscription under which you would like to deploy the VM:

   ```
   az login
   ```

2. Create a resource group for containment and deployment of your VM and its related resources, using `az group` create. You can skip this step if you already have resource group you would like to use instead:

```
az group create --name myResourceGroup --location westus
```

3. Create a VM using `az vm create`. The following example creates a VM named myVM with a system-assigned managed identity, as requested by the `--assign-identity` parameter. The `--admin-username` and `--admin-password` parameters specify the administrative user name and password account for virtual machine sign-in. Update these values as appropriate for your environment:

```
az vm create --resource-group myResourceGroup --name myVM --image win-
2016datacenter --generate-ssh-keys --assign-identity --admin-username
azureuser --admin-password myPassword12
```

Enable system-assigned managed identity on an existing Azure VM

Sign in to Azure using an account that is associated with the Azure subscription that contains the VM. Use `az vm` with the `identity assign` command to enable to the system-assigned identity to an existing VM.

```
az vm identity assign -g myResourceGroup -n myVm
```

Disable system-assigned identity from an Azure VM

To disable system-assigned managed identity on a VM, your account needs the Virtual Machine Contributor role assignment. No additional Azure AD directory role assignments are required.

If you have a Virtual Machine that no longer needs the system-assigned identity, but still needs user-assigned identities, use the following command:

```
az vm update -n myVM -g myResourceGroup --set identity.type='UserAssigned'
```

If you have a virtual machine that no longer needs system-assigned identity and it has no user-assigned identities, use the following command:

Note: The value none is case sensitive. It must be lowercase.

```
az vm update -n myVM -g myResourceGroup --set identity.type="none"
```

To remove the managed identity for Azure resources VM extension (planned for deprecation in January 2019), user `-n ManagedIdentityExtensionForWindows` or `-n ManagedIdentityExtensionForLinux` switch (depending on the type of VM):

```
az vm identity --resource-group myResourceGroup --vm-name myVm -n Manage-
dIdentityExtensionForWindows
```

User-assigned managed identity

In this section, you will learn how to add and remove a user-assigned managed identity from an Azure VM using Azure CLI.

To assign a user-assigned identity to a VM during its creation, your account needs the Virtual Machine Contributor and Managed Identity Operator role assignments. No additional Azure AD directory role assignments are required.

Important: When creating user assigned identities, only alphanumeric characters (0-9, a-z, A-Z) and the hyphen (-) are supported. Additionally, the name should be limited to 24 characters in length for the assignment to VM/VMSS to work properly. Check back for updates. For more information, see **FAQs and known issues**[7].

Assign a user-assigned managed identity during the creation of an Azure VM

1. You can skip this step if you already have a resource group you would like to use. Create a resource group for containment and deployment of your user-assigned managed identity, using `az group create`. Be sure to replace the `<RESOURCE GROUP>` and `<LOCATION>` parameter values with your own values. :

   ```
   az group create --name <RESOURCE GROUP> --location <LOCATION>
   ```

2. Create a user-assigned managed identity using `az identity create`. The `-g` parameter specifies the resource group where the user-assigned managed identity is created, and the `-n` parameter specifies its name.

   ```
   az identity create -g myResourceGroup -n myUserAssignedIdentity
   ```

3. The response contains details for the user-assigned managed identity created, similar to the following. The resource id value assigned to the user-assigned managed identity is used in the following step.

   ```
   {
       "clientId": "73444643-8088-4d70-9532-c3a0fdc190fz",
       "clientSecretUrl": "https://control-westcentralus.identity.azure.net/
   subscriptions/<SUBSCRIPTON ID>/resourcegroups/<RESOURCE GROUP>/providers/
   Microsoft.ManagedIdentity/userAssignedIdentities/<myUserAssignedIdentity>/
   credentials?tid=5678&oid=9012&aid=73444643-8088-4d70-9532-c3a0fdc190fz",
       "id": "/subscriptions/<SUBSCRIPTON ID>/resourcegroups/<RESOURCE GROUP>/
   providers/Microsoft.ManagedIdentity/userAssignedIdentities/<USER ASSIGNED
   IDENTITY NAME>",
       "location": "westcentralus",
       "name": "<USER ASSIGNED IDENTITY NAME>",
       "principalId": "e5fdfdc1-ed84-4d48-8551-fe9fb9dedfll",
       "resourceGroup": "<RESOURCE GROUP>",
       "tags": {},
       "tenantId": "733a8f0e-ec41-4e69-8ad8-971fc4b533bl",
       "type": "Microsoft.ManagedIdentity/userAssignedIdentities"
   }
   ```

4. Create a VM using `az vm create`. The following example creates a VM associated with the new user-assigned identity, as specified by the `--assign-identity` parameter. Be sure to replace the `<RESOURCE GROUP>`, `<VM NAME>`, `<USER NAME>`, `<PASSWORD>`, and `<USER ASSIGNED IDENTITY NAME>` parameter values with your own values.

7 https://docs.microsoft.com/en-us/azure/active-directory/managed-service-identity/known-issues

```
az vm create --resource-group <RESOURCE GROUP> --name <VM NAME> --image
UbuntuLTS --admin-username <USER NAME> --admin-password <PASSWORD> --as-
sign-identity <USER ASSIGNED IDENTITY NAME>
```

Assign a user-assigned managed identity to an existing Azure VM

1. Create a user-assigned identity using `az identity create`. The `-g` parameter specifies the resource group where the user-assigned identity is created, and the `-n` parameter specifies its name. Be sure to replace the `<RESOURCE GROUP>` and `<USER ASSIGNED IDENTITY NAME>` parameter values with your own values:

```
az identity create -g <RESOURCE GROUP> -n <USER ASSIGNED IDENTITY NAME>
```

1. The response contains details for the user-assigned managed identity created, similar to the following.

```
{
    "clientId": "73444643-8088-4d70-9532-c3a0fdc190fz",
    "clientSecretUrl": "https://control-westcentralus.identity.azure.net/
subscriptions/<SUBSCRIPTON ID>/resourcegroups/<RESOURCE GROUP>/providers/
Microsoft.ManagedIdentity/userAssignedIdentities/<USER ASSIGNED IDENTITY
NAME>/credentials?tid=5678&oid=9012&aid=73444643-8088-4d70-9532-c3a0fd-
c190fz",
    "id": "/subscriptions/<SUBSCRIPTON ID>/resourcegroups/<RESOURCE GROUP>/
providers/Microsoft.ManagedIdentity/userAssignedIdentities/<USER ASSIGNED
IDENTITY NAME>",
    "location": "westcentralus",
    "name": "<USER ASSIGNED IDENTITY NAME>",
    "principalId": "e5fdfdc1-ed84-4d48-8551-fe9fb9dedfll",
    "resourceGroup": "<RESOURCE GROUP>",
    "tags": {},
    "tenantId": "733a8f0e-ec41-4e69-8ad8-971fc4b533bl",
    "type": "Microsoft.ManagedIdentity/userAssignedIdentities"
}
```

2. Assign the user-assigned identity to your VM using `az vm identity assign`. Be sure to replace the `<RESOURCE GROUP>` and `<VM NAME>` parameter values with your own values. The `<USER ASSIGNED IDENTITY NAME>` is the user-assigned managed identity's resource name property, as created in the previous step:

```
az vm identity assign -g <RESOURCE GROUP> -n <VM NAME> --identities <USER
ASSIGNED IDENTITY>
```

Remove a user-assigned managed identity from an Azure VM

To remove a user-assigned identity to a VM, your account needs the Virtual Machine Contributor role assignment.

If this is the only user-assigned managed identity assigned to the virtual machine, `UserAssigned` will be removed from the identity type value. Be sure to replace the `<RESOURCE GROUP>` and `<VM NAME>` parameter values with your own values. The `<USER ASSIGNED IDENTITY>` will be the user-assigned identity's name property, which can be found in the identity section of the virtual machine using `az vm identity show`:

```
az vm identity remove -g <RESOURCE GROUP> -n <VM NAME> --identities <USER ASSIGNED IDENTITY>
```

If your VM does not have a system-assigned managed identity and you want to remove all user-assigned identities from it, use the following command:

Note: The value `none` is case sensitive. It must be lowercase.

```
az vm update -n myVM -g myResourceGroup --set identity.type="none" identity.userAssignedIdentities=null
```

If your VM has both system-assigned and user-assigned identities, you can remove all the user-assigned identities by switching to use only system-assigned by using the following command:

```
az vm update -n myVM -g myResourceGroup --set identity.type='SystemAssigned' identity.userAssignedIdentities=null
```

How to use managed identities for Azure resources on an Azure VM to acquire an access token

A client application can request managed identities for Azure resources app-only access token for accessing a given resource. The token is based on the managed identities for Azure resources service principal. As such, there is no need for the client to register itself to obtain an access token under its own service principal. The token is suitable for use as a bearer token in service-to-service calls requiring client credentials.

Get a token using HTTP

The fundamental interface for acquiring an access token is based on REST, making it accessible to any client application running on the VM that can make HTTP REST calls. This is similar to the Azure AD programming model, except the client uses an endpoint on the virtual machine (vs an Azure AD endpoint).

Sample request using the Azure Instance Metadata Service (IMDS) endpoint:

```
GET 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https://management.azure.com/' HTTP/1.1 Metadata: true
```

Element	Description
GET	The HTTP verb, indicating you want to retrieve data from the endpoint. In this case, an OAuth access token.
http://169.254.169.254/metadata/identity/oauth2/token	The managed identities for Azure resources endpoint for the Instance Metadata Service.
api-version	A query string parameter, indicating the API version for the IMDS endpoint. Please use API version 2018-02-01 or greater.
resource	A query string parameter, indicating the App ID URI of the target resource. It also appears in the aud (audience) claim of the issued token. This example requests a token to access Azure Resource Manager, which has an App ID URI of https://management.azure.com/.
Metadata	An HTTP request header field, required by managed identities for Azure resources as a mitigation against Server Side Request Forgery (SSRF) attack. This value must be set to "true", in all lower case.
object_id	(Optional) A query string parameter, indicating the object_id of the managed identity you would like the token for. Required, if your VM has multiple user-assigned managed identities.
client_id	(Optional) A query string parameter, indicating the client_id of the managed identity you would like the token for. Required, if your VM has multiple user-assigned managed identities.

Sample response:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "access_token": "eyJ0eXAi...",
  "refresh_token": "",
  "expires_in": "3599",
  "expires_on": "1506484173",
  "not_before": "1506480273",
  "resource": "https://management.azure.com/",
  "token_type": "Bearer"
}
```

Element	Description
access_token	The requested access token. When calling a secured REST API, the token is embedded in the Authorization request header field as a "bearer" token, allowing the API to authenticate the caller.

Element	Description
`refresh_token`	Not used by managed identities for Azure resources.
`expires_in`	The number of seconds the access token continues to be valid, before expiring, from time of issuance. Time of issuance can be found in the token's `iat` claim.
`expires_on`	The timespan when the access token expires. The date is represented as the number of seconds from "1970-01-01T0:0:0Z UTC" (corresponds to the token's `exp` claim).
not_before	The timespan when the access token takes effect, and can be accepted. The date is represented as the number of seconds from "1970-01-01T0:0:0Z UTC" (corresponds to the token's `nbf` claim).
`resource`	The resource the access token was requested for, which matches the `resource` query string parameter of the request.
`token_type`	The type of token, which is a "Bearer" access token, which means the resource can give access to the bearer of this token.

Token caching

While the managed identities for Azure resources subsystem being used (IMDS/managed identities for Azure resources VM Extension) does cache tokens, we also recommend to implement token caching in your code. As a result, you should prepare for scenarios where the resource indicates that the token is expired.

On-the-wire calls to Azure AD result only when:

- cache miss occurs due to no token in the managed identities for Azure resources subsystem cache
- the cached token is expired

Error handling

The managed identities for Azure resources endpoint signals errors via the status code field of the HTTP response message header, as either 4xx or 5xx errors:

Status Code	Error Reason	How To Handle
404 Not found.	IMDS endpoint is updating.	Retry with Expontential Backoff. See guidance below.
429 Too many requests.	IMDS Throttle limit reached.	Retry with Exponential Backoff. See guidance below.
4xx Error in request.	One or more of the request parameters was incorrect.	Do not retry. Examine the error details for more information. 4xx errors are design-time errors.

Status Code	Error Reason	How To Handle
5xx Transient error from service.	The managed identities for Azure resources sub-system or Azure Active Directory returned a transient error.	It is safe to retry after waiting for at least 1 second. If you retry too quickly or too often, IMDS and/or Azure AD may return a rate limit error (429).
timeout	IMDS endpoint is updating.	Retry with Expontential Backoff. See guidance below.

If an error occurs, the corresponding HTTP response body contains JSON with the error details:

Element	Description
error	Error identifier.
error_description	Verbose description of error. Error descriptions can change at any time. Do not write code that branches based on values in the error description.

HTTP response reference

This section documents the possible error responses. A "200 OK" status is a successful response, and the access token is contained in the response body JSON, in the `access_token` element.

Status code	Error	Error Description	Solution
400 Bad Request	invalid_resource	AADSTS50001: The application named *<URI>* was not found in the tenant named *<TENANT-ID>*. This can happen if the application has not been installed by the administrator of the tenant or consented to by any user in the tenant. You might have sent your authentication request to the wrong tenant.\	(Linux only)
400 Bad Request	bad_request_102	Required metadata header not specified	Either the `Metadata` request header field is missing from your request, or is formatted incorrectly. The value must be specified as `true`, in all lower case. See the "Sample request" in the preceding REST section for an example.

Status code	Error	Error Description	Solution
401 Unauthorized	unknown_source	Unknown Source <*URI*>	Verify that your HTTP GET request URI is formatted correctly. The `scheme:host/resource-path` portion must be specified as `http://localhost:50342/oauth2/token`.
	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.	
	unauthorized_client	The client is not authorized to request an access token using this method.	Caused by a request that didn't use local loopback to call the extension, or on a VM that doesn't have managed identities for Azure resources configured correctly.
	access_denied	The resource owner or authorization server denied the request.	
	unsupported_response_type	The authorization server does not support obtaining an access token using this method.	
	invalid_scope	The requested scope is invalid, unknown, or malformed.	
500 Internal server error	unknown	Failed to retrieve token from the Active directory. For details see logs in <*file path*>	Verify that managed identities for Azure resources has been enabled on the VM. Also verify that your HTTP GET request URI is formatted correctly, particularly the resource URI specified in the query string.

Retry guidance

It is recommended to retry if you receive a 404, 429, or 5xx error code.

Throttling limits apply to the number of calls made to the IMDS endpoint. When the throttling threshold is exceeded, IMDS endpoint limits any further requests while the throttle is in effect. During this period, the IMDS endpoint will return the HTTP status code 429 ("Too many requests"), and the requests fail.

For retry, we recommend the following strategy:

Retry strategy	Settings	Values	How it works
ExponentialBackoff	Retry count Min back-off Max back-off Delta back-off First fast retry	5 0 sec 60 sec 2 sec false	Attempt 1 - delay 0 sec Attempt 2 - delay ~2 sec Attempt 3 - delay ~6 sec Attempt 4 - delay ~14 sec Attempt 5 - delay ~30 sec

Assign a managed identity access to a resource using Azure CLI

Once you've configured an Azure resource with a managed identity, you can give the managed identity access to another resource, just like any security principal. This example shows you how to give an Azure virtual machine or virtual machine scale set's managed identity access to an Azure storage account using Azure CLI.

Use RBAC to assign a managed identity access to another resource

After you've enabled managed identity on an Azure resource, such as an Azure virtual machine:

1. If you're using the Azure CLI in a local console, first sign in to Azure using `az login`. Use an account that is associated with the Azure subscription under which you would like to deploy the VM:

   ```
   az login
   ```

2. In this example, we are giving an Azure virtual machine access to a storage account. First we use `az resource list` to get the service principal for the virtual machine named `myVM`:

   ```
   spID=$(az resource list -n myVM --query [*].identity.principalId --out tsv)
   ```

3. For an Azure virtual machine scale set, the command is the same except here, you get the service principal for the virtual machine scale set named `DevTestVMSS`:

   ```
   spID=$(az resource list -n DevTestVMSS --query [*].identity.principalId
   --out tsv)
   ```

4. Once you have the service principal ID, use `az role assignment create` to give the virtual machine `Reader` access to a storage account called `myStorageAcct`:

```
az role assignment create --assignee $spID --role 'Reader' --scope /sub-
scriptions/<mySubscriptionID>/resourceGroups/<myResourceGroup>/providers/
Microsoft.Storage/storageAccounts/myStorageAcct
```

Online Lab - Implementing Custom Role Based Access Control (RBAC) Roles

Lab Steps

Online Lab: Implementing Custom Role Based Access Control (RBAC) Roles

NOTE: For the most recent version of this online lab, see: **https://github.com/MicrosoftLearning/AZ-300-MicrosoftAzureArchitectTechnologies**

Scenario

Adatum Corporation wants to implement custom RBAC roles to delegate permissions to start and stop (deallocate) Azure VMs.

Objectives

After completing this lab, you will be able to:

- Define a custom RBAC role
- Assign a custom RBAC role

Lab Setup

Estimated Time: 30 minutes

User Name: **Student**

Password: **Pa55w.rd**

Exercise 1: Define a custom RBAC role

The main tasks for this exercise are as follows:

1. Deploy an Azure VM by using an Azure Resource Manager template
2. Identify actions to delegate via RBAC
3. Create a custom RBAC role in an Azure AD tenant

Task 1: Deploy an Azure VM by using an Azure Resource Manager template

1. From the lab virtual machine, start Microsoft Edge and browse to the Azure portal at **http://portal.azure.com** and sign in by using the Microsoft account that has the Owner role in the target Azure subscription.
2. In the Azure portal, in the Microsoft Edge window, start a **PowerShell** session within the **Cloud Shell**.
3. If you are presented with the **You have no storage mounted** message, configure storage using the following settings:
   - Subsciption: the name of the target Azure subscription
   - Cloud Shell region: the name of the Azure region that is available in your subscription and which is closest to the lab location
   - Resource group: the name of a new resource group **az3000900-LabRG**
   - Storage account: a name of a new storage account
   - File share: a name of a new file share
4. From the Cloud Shell pane, create a resource groups by running (replace the `<Azure region>` placeholder with the name of the Azure region that is available in your subscription and which is closest to the lab location)

   ```
   New-AzResourceGroup -Name az3000901-LabRG -Location <Azure region>
   ```

5. From the Cloud Shell pane, upload the Azure Resource Manager template **\allfiles\AZ-300T03\Module_04\azuredeploy09.json** into the home directory.
6. From the Cloud Shell pane, upload the parameter file **\allfiles\AZ-300T03\Module_04\azuredeploy09.parameters.json** into the home directory.
7. From the Cloud Shell pane, deploy an Azure VM hosting Ubuntu by running:

   ```
   New-AzResourceGroupDeployment -ResourceGroupName az3000901-LabRG -Template-
   File azuredeploy09.json -TemplateParameterFile azuredeploy09.parameters.
   json
   ```

8. **Note**: Do not wait for the deployment to complete but instead proceed to the next task.
9. In the Azure portal, close the Cloud Shell pane.

Task 2: Identify actions to delegate via RBAC

1. In the Azure portal, navigate to the **az3000901-LabRG** blade.
2. On the **az3000901-LabRG** blade, click **Access Control (IAM)**.
3. On the **az3000901-LabRG - Access Control (IAM)** blade, click **Roles**.
4. On the **Roles** blade, click **Owner**.

5. On the **Owner** blade, click **Permissions**.
6. On the **Permissions (preview)** blade, click **Microsoft Compute**.
7. On the **Microsoft Compute** blade, click **Virtual machines**.
8. On the **Virtual Machines** blade, review the list of management actions that can be delegated through RBAC. Note that they include the **Deallocate Virtual Machine** and **Start Virtual Machine** actions.

Task 3: Create a custom RBAC role in an Azure AD tenant

1. On the lab computer, open the file **\allfiles\AZ-300T03\Module_04\customRoleDefinition09.json** and review its content:

```
{
   "Name": "Virtual Machine Operator (Custom)",
   "Id": null,
   "IsCustom": true,
   "Description": "Allows to start and stop (deallocate) Azure VMs",
   "Actions": [
       "Microsoft.Compute/*/read",
       "Microsoft.Compute/virtualMachines/deallocate/action",
       "Microsoft.Compute/virtualMachines/start/action"
   ],
   "NotActions": [
   ],
   "AssignableScopes": [
       "/subscriptions/SUBSCRIPTION_ID"
   ]
}
```

2. In the Azure portal, in the Microsoft Edge window, start a **PowerShell** session within the **Cloud Shell**.
3. From the Cloud Shell pane, upload the Azure Resource Manager template **\allfiles\AZ-300T03\Module_04\customRoleDefinition09.json** into the home directory.
4. From the Cloud Shell pane, run the following to replace the **$SUBSCRIPTION_ID** placeholder with the ID value of the Azure subscription:

```
$subscription_id = (Get-AzSubscription).Id
(Get-Content -Path $HOME/customRoleDefinition09.json) -Replace 'SUBSCRIP-
TION_ID', "$subscription_id" | Set-Content -Path $HOME/customRoleDefini-
tion09.json
```

5. From the Cloud Shell pane, run the following to create the custom role definition:

```
New-AzRoleDefinition -InputFile $HOME/customRoleDefinition09.json
```

6. From the Cloud Shell pane, run the following to verify that the role was created successfully:

```
Get-AzRoleDefinition -Name 'Virtual Machine Operator (Custom)'
```

7. Close the Cloud Shell pane.

Result: After you completed this exercise, you have defined a custom RBAC role

Exercise 2: Assign and test a custom RBAC role

The main tasks for this exercise are as follows:

1. Create an Azure AD user
2. Create an RBAC role assignment
3. Test the RBAC role assignment

Task 1: Create an Azure AD user

1. In the Azure portal, in the Microsoft Edge window, start a **PowerShell** session within the **Cloud Shell**.
2. From the Cloud Shell pane, run the following to identify the Azure AD DNS domain name:

```
$domainName = ((Get-AzureAdTenantDetail).VerifiedDomains)[0].Name
```

3. From the Cloud Shell pane, run the following to create a new Azure AD user:

```
$passwordProfile = New-Object -TypeName Microsoft.Open.AzureAD.Model.Pass-
wordProfile
$passwordProfile.Password = 'Pa55w.rd1234'
$passwordProfile.ForceChangePasswordNextLogin = $false
New-AzureADUser -AccountEnabled $true -DisplayName 'lab user0901' -Pass-
wordProfile $passwordProfile -MailNickName 'labuser0901' -UserPrincipalName
"labuser0901@$domainName"
```

4. From the Cloud Shell pane, run the following to identify the user principal name of the newly created Azure AD user:

```
(Get-AzureADUser -Filter "MailNickName eq 'labuser0901'").UserPrincipalName
```

5. Close the Cloud Shell pane.

Task 2: Create an RBAC role assignment

1. In the Azure portal, navigate to the **az3000901-LabRG** blade.
2. On the **az3000901-LabRG** blade, click **Access Control (IAM)**.
3. On the **az3000901-LabRG - Access Control (IAM)** blade, click **+ Add** and select the **Add role assignment** option.

4. On the **Add role assignment** blade, specify the following settings and click **Save**:
   - Role: **Virtual Machine Operator (Custom)**
   - Assign access to: **Azure AD user, group, or application**
   - Select: **lab user0901**

Task 3: Test the RBAC role assignment

1. Start a new in-private Microsoft Edge window, browse to the Azure portal at **http://portal.azure.com** and sign in by using the newly created user account:
   - Username: the user principal name you identified in the first task of this exercise
   - Password: **Pa55w.rd1234**
2. In the Azure portal, navigate to the **Resource groups** blade. Note that you are not able to see any resource groups.
3. In the Azure portal, navigate to the **All resources** blade. Note that you are able to see only the **az3000901-vm** and its managed disk.
4. In the Azure portal, navigate to the **az3000901-vm** blade. Try restarting the virtual machine. Review the error message in the notification area and note that this action failed because the current user is not authorized to carry it out.
5. Stop the virtual machine and verify that the action completed successfully.

Result: After you completed this exercise, you have assigned and tested a custom RBAC role

Review Questions

Module 1 Review Questions

Token-based authentication

You develop a message board for students to collaborate on courses they take at a college.

The message board should allow students to use their existing social media accounts to register and authenticate.

You decide to implement ASP.NET Identity to allow social media accounts to be used in the application.

What are the benefits of using ASP.NET Identity for the application? What functionality can be added to the application in the future by using ASP.NET Identity?

Suggested Answer ↓

ASP.NET Identity is a unified identity platform for ASP.NET applications that can be used across all flavors of ASP.NET and that can be used in web, phone, store, or hybrid applications. ASP.NET Identity implements two core features that makes it ideal for token-based authentication:

ASP.NET Identity implements a provider model for logins. Today you may want to log in using a local Active Directory server, but tomorrow you may want to migrate to Azure AD. In ASP.NET Identity, you can simply add, remove, or replace providers. If your company decides to implement social network logins, you can keep adding providers or write your own providers without changing any other code in your application.

App Service Authentication

You develop a game that will use the players social media account for authentication and access to the app.

You decide to implement ASP.NET Identity to allow social media accounts to be used in your application.

You need to ensure that users can post high scores from the application to timelines on social media platforms.

How can you use Azure App Service to enable this functionality? How does it work?

Suggested Answer ↓

Azure App Service provides built-in authentication and authorization support, so you can sign in users and access data by writing minimal or no code in your app instance. The authentication and authorization module runs in the same sandbox as your application code.

Identity information flows directly into the application code. For all language frameworks, App Service makes the user's claims available to your code by injecting them into the request headers.

App Service provides a built-in token store, which is a repository of tokens that are associated with the users of your web apps, APIs, or native mobile apps. When you enable authentication with any provider, this token store is immediately available to your app. The token information can be used in your application code to perform tasks such as posting to the authenticated user's social media timeline.

Security Best Practices

Your company has experienced several instances of data loss. The losses are a combination of weak passwords, loss of hardware, and brute force attacks.

You decide to implement multi-factor authentication (MFA).

What tenets of secure authentication should you consider before you deploy MFA?

Suggested Answer ↓

In security best practices, it is recommended to use two or more factors when authenticating users. This practice is referred to as multi-factor authentication. Using an enterprise as an example, the company could require employees to scan their badges and then enter their passwords as two factors of authentication. In the world of security, it is often recommended to have two of the following factors:

- **Knowledge** – Something that only the user knows (security questions, password, or PIN).
- **Possession** – Something that only the user has (corporate badge, mobile device, or security token).
- **Inherence** – Something that only the user is (fingerprint, face, voice, or iris).

The security of two-step verification lies in its layered approach. Compromising multiple authentication factors presents a significant challenge for attackers. Even if an attacker manages to learn the user's password, it is useless without possession of the additional authentication method.

Module 5 Module Implementing Secure Data

Encryption options

Encryption

Encryption is the process of translating plain text data (**plaintext**) into something that appears to be random and meaningless (ciphertext). Decryption is the process of converting ciphertext back to plaintext. To encrypt more than a small amount of data, **symmetric encryption** is used. A **symmetric key** is used during both the encryption and the decryption process. To decrypt a particular piece of ciphertext, the key that was used to encrypt the data must be used.

The goal of every encryption algorithm is to make it as difficult as possible to decrypt the generated ciphertext without using the key. If a really good encryption algorithm is used, there is no technique significantly better than methodically trying every possible key. For such an algorithm, the longer the key, the more difficult it is to decrypt a piece of ciphertext without possessing the key. It is difficult to determine the quality of an encryption algorithm. Algorithms that look promising sometimes turn out to be very easy to break, given the proper attack. When selecting an encryption algorithm, it is a good idea to choose one that has been in use for several years and has successfully resisted all attacks.

Encryption at rest

Encryption at rest is the encoding (encryption) of data when it is persisted. It is a common security requirement that data that is persisted on disk be encrypted with a secret encryption key. Encryption at rest helps provide data protection for stored data (at rest). Attacks against data at rest include attempts to obtain physical access to the hardware on which the data is stored and to then compromise the contained data. In such an attack, a server's hard drive may have been mishandled during maintenance, allowing an attacker to remove the hard drive. Later, the attacker puts the hard drive into a computer under their control to attempt to access the data.

Encryption at rest is designed to prevent the attacker from accessing the unencrypted data by ensuring that the data is encrypted when on disk. If an attacker were to obtain a hard drive with such encrypted data but no access to the encryption keys, the attacker would not compromise the data without great difficulty. In such a scenario, an attacker would have to attempt attacks against encrypted data, which are much more complex and resource consuming than accessing unencrypted data on a hard drive. For this

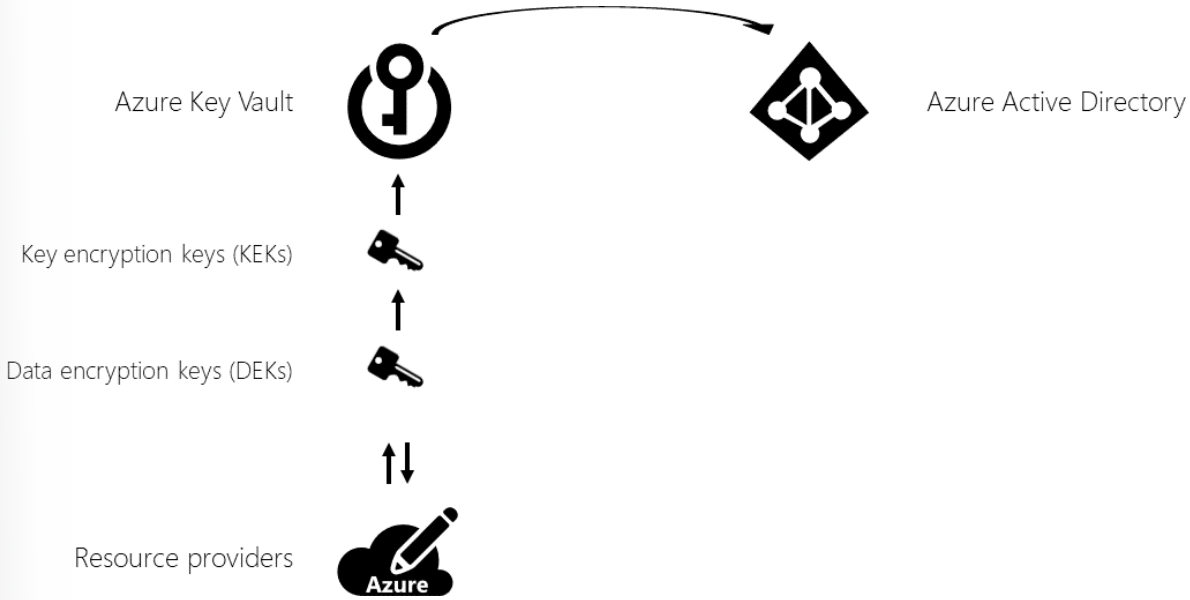
reason, encryption at rest is highly recommended and is a high-priority requirement for many organizations.

Encryption at rest may also be required by an organization's need for data governance and compliance efforts. Industry and government regulations, such as the Health Insurance Portability and Accountability Act (HIPAA), PCI DSS, and Federal Risk and Authorization Management Program (FedRAMP), lay out specific safeguards regarding data protection and encryption requirements. Encryption at rest is a mandatory measure required for compliance with some of those regulations. In addition to meeting compliance and regulatory requirements, encryption at rest should be perceived as a defense-in-depth platform capability.

In Microsoft Azure, organizations can achieve encryption at rest without having the cost of implementation and management and the risk of a custom key management solution. While Microsoft provides a compliant platform for services, applications, data, comprehensive facility and physical security enhancement, data access control, and auditing, it is important to provide additional, overlapping security measures in case one of the other security measures fails. Encryption at rest provides such an additional defense mechanism.

The encryption at rest designs in Azure use symmetric encryption to encrypt and decrypt large amounts of data quickly according to a simple conceptual model:

- A symmetric encryption key is used to encrypt data as it is written to storage.
- The same encryption key is used to decrypt that data as it is readied for use in memory.
- Data may be partitioned, and different keys may be used for each partition.
- Keys must be stored in a security-enhanced location with access control policies limiting access to certain identities and logging key usage. Data encryption keys are often encrypted with asymmetric encryption to further limit access.



Azure Storage encryption

All Azure Storage services (Blob storage, Queue storage, Table storage, and Azure Files) support server-side encryption at rest, with some services supporting customer-managed keys and client-side

encryption. All Azure Storage services enable server-side encryption by default using service-managed keys, which is transparent to the application.

Storage Service Encryption is enabled for all new and existing storage accounts and cannot be disabled. Because your data is security enhanced by default, you don't need to modify your code or applications to take advantage of Storage Service Encryption.

Azure SQL Database encryption

Azure SQL Database supports encryption at rest for Microsoft-managed server-side and client-side encryption scenarios. Support for server encryption is currently provided through the unified SQL feature called Transparent Data Encryption (TDE). Once an Azure SQL Database customer enables TDE, keys are automatically created and managed for them. Encryption at rest can be enabled at the database and server levels. TDE is enabled by default on newly created databases. Azure SQL Database also supports RSA 2048-bit customer-managed keys in Azure Key Vault.

Azure Cosmos DB encryption

Cosmos DB stores its primary databases on solid-state drives (SSDs). Its media attachments and backups are stored in Azure Blob storage, which is generally backed up by hard disk drives (HDDs). Cosmos DB automatically encrypts all databases, media attachments and backups.

End-to-end encryption

Encrypt data with Transparent Data Encryption (TDE)

You can take several precautions to help secure the database, such as designing a security-enhanced system, encrypting confidential assets, and building a firewall around the database servers. However, in a scenario where the physical media (such as drives or backup tapes) are stolen, a malicious party can just restore or attach the database and browse the data. One solution is to encrypt the sensitive data in the database and help to protect the keys that are used to encrypt the data with a certificate. This helps prevent anyone without the keys from using the data, but this kind of protection must be planned in advance.

TDE encrypts SQL Server, Azure SQL Database, and Azure SQL Data Warehouse data files. TDE performs real-time I/O encryption and decryption of the data and log files. The encryption of the database file is performed at the page level. The pages in an encrypted database are encrypted before they are written to disk and decrypted when read into memory. TDE does not increase the size of the encrypted database.

The encryption uses a database encryption key (DEK), which is stored in the database boot record for availability during recovery. The DEK is either a symmetric key secured by using a certificate stored in the master database of the server or an asymmetric key protected by an Extensible Key Management (EKM) module. TDE protects data at rest, meaning the data and log files. It provides the ability to comply with many laws, regulations, and guidelines established in various industries. This enables software developers to encrypt data by using the AES and 3DES encryption algorithms without changing existing applications.

Encrypt data with Always Encrypted

Always Encrypted is a new data encryption technology in Azure SQL Database and SQL Server that helps protect sensitive data at rest on the server, during movement between client and server, and while the data is in use, helping to ensure that sensitive data never appears as plaintext inside the database system.

Always Encrypted is a feature designed to protect sensitive data, such as credit card numbers or national identification numbers (for example, United States social security numbers), stored in Azure SQL Database or SQL Server databases. Always Encrypted allows clients to encrypt sensitive data inside client applications and never reveal the encryption keys to the database engine (SQL Database or SQL Server). As a result, Always Encrypted provides a separation between those who own the data (and can view it) and those who manage the data (but should have no access). After you encrypt the data, only client applications or app servers that have access to the keys can access the plaintext data.

By helping ensure that on-premises database administrators, cloud database operators, or other highly privileged but unauthorized users cannot access the encrypted data, Always Encrypted allows organizations to encrypt data at rest and in use for storage in Azure, to enable the delegation of on-premises database administration to third parties, or to reduce security clearance requirements for database administrators.

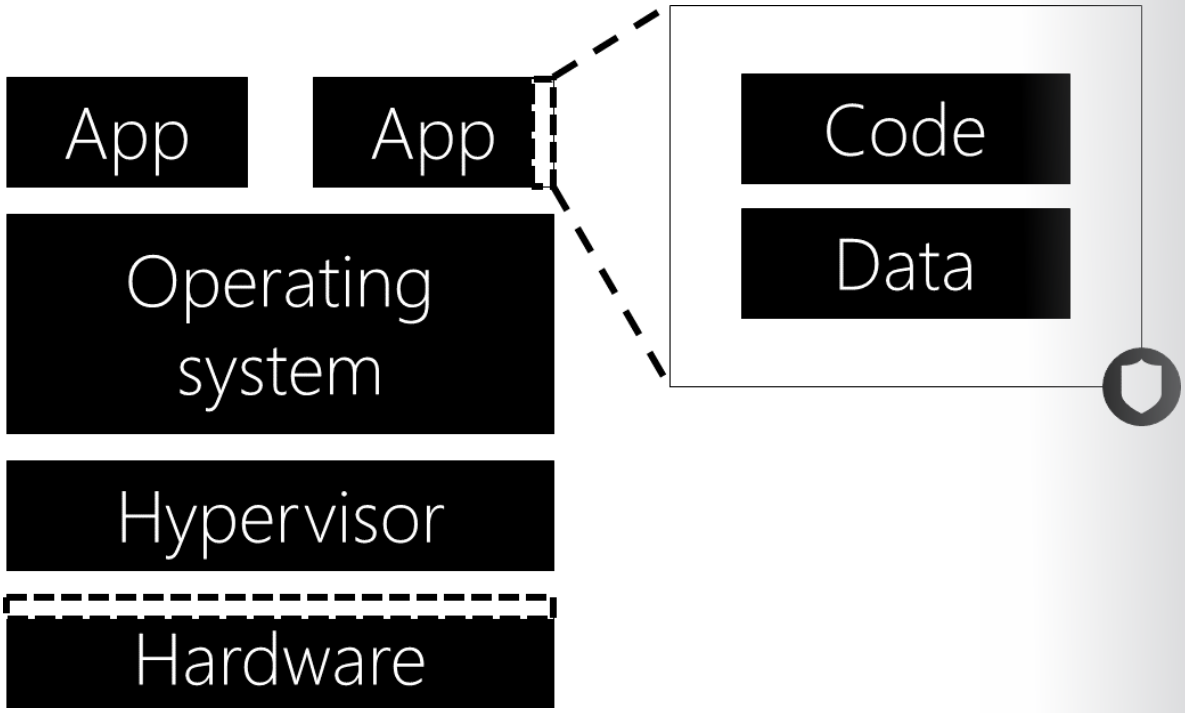
Note: Always Encrypted requires a specialized driver installed on the client computer to automatically encrypt and decrypt sensitive data in the client application. For many applications, this does require some code changes. This is in contrast to TDE, which only requires a change to the application's connection string.

Implement Azure confidential computing

Azure confidential computing

Azure confidential computing refers to features available in many Azure services that encrypt data in use. Confidential computing is designed for scenarios where data needs to be processed in the cloud while still maintaining a level of encryption that helps protect the data from being viewed in a plaintext manner. Confidential computing is a collaborative project between hardware vendors like Intel and software vendors like Microsoft.

Confidential computing helps to ensure that when data is "in the clear," which is required for efficient processing, the data is protected inside a Trusted Execution Environment (TEE). TEEs help to ensure that there is no way to view data or operations inside from the outside, even with a debugger. They also help to ensure that only authorized code is permitted to access data. If the code is altered or tampered with, the operations are denied and the environment disabled. The TEE enforces these protections throughout the execution of the code within it.



Note: In some online articles, TEEs are commonly referred to as **enclaves**.

The goal of confidential computing is to build a platform where developers can take advantage of both hardware and software TEEs without being required to change their code. TEEs are exposed in multiple ways:

- **Hardware** – Intel Xeon processors with Intel SGX technology are available for Azure Virtual Machines.
- **Software** – The Intel SGX software development kit (SDK) and third-party enclave APIs can be used with compute instances and Virtual Machines in Azure.
- **Services** – Many Azure services, such as Azure SQL Database, already execute code in TEEs.
- **Frameworks** – The Microsoft Research team has developer frameworks, such as the Confidential Consortium Blockchain Framework, to help jumpstart new projects that need to run in TEEs.

Implement SSL and TLS communications

SSL and TLS overview

Transport Layer Security (TLS) and Secure Sockets Layer (SSL) are cryptographic protocols that help provide communications security over a computer network. SSL encryption is the most commonly used method of helping secure data sent across the internet. Many Azure services, including (but not limited to) the following, support SSL encryption:

- Azure SQL Database
- Azure Database for MySQL
- Azure Storage
- Azure Application Gateway
- Azure App Service

TLS in Azure Storage

SSL 1.0, 2.0 and 3.0 have been found to be vulnerable, and they have been prohibited by an Internet Engineering Task Force (IETF) Request For Comments (RFC). Many services and clients have moved forward to TLS 1.0. Unfortunately, TLS 1.0 became insecure for using insecure block ciphers (Data Encryption Standard [DES] CBC and RC2 CBC) and an insecure stream cipher (RC4). The Payment Card Industry's (PCI) Standards Council has recommended moving on to newer versions of TLS.

For these reasons, the Azure Storage team has determined that TLS 1.2 is the best protocol to use when connecting to Azure Storage accounts. To help ensure a secure and compliant connection to Azure Storage, you need to enable TLS 1.2 or newer on the client side before sending requests to operate the Azure Storage service.

To enable TLS 1.2 in Microsoft .NET, you should use the `ServicePointManager` class in the `System.Net` namespace:

```
System.Net.ServicePointManager.SecurityProtocol = System.Net.SecurityProtocolType.Tls12;
```

To enable TLS 1.2 in PowerShell, you can use the same class:

```
[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.SecurityProtocolType]::Tls12;
```

Note: While not recommend, TLS 1.0 and 1.1 are still supported by Azure Storage for older client applications.

Manage cryptographic keys in Azure Key Vault

Azure Key Vault

You have passwords, connection strings, and other pieces of information that are needed to keep your applications working. You want to make sure that this information is available but that it is security enhanced. Azure Key Vault is a cloud service that works as a security-enhanced secrets store.

Key Vault allows you to create multiple security-enhanced containers, called **vaults**. These vaults are backed by **hardware security modules (HSMs)**. Vaults help to reduce the chance of accidentally losing security information by centralizing the storage of application secrets. Vaults also control and log the access to anything stored in them. Azure Key Vault is designed to support any type of **secret**, such as a password, database credential, API key, or certificate. Software or HSMs can help to protect these secrets. Azure Key Vault can handle requesting and renewing TLS certificates, providing the features required for a robust certificate lifecycle management solution.

Key Vault streamlines the key management process and enables you to maintain control of keys that access and encrypt your data. Developers can create keys for development and testing in minutes and then seamlessly migrate them to production keys. Security administrators can grant (and revoke) permission to keys as needed.

Accessing Key Vault in Azure CLI

To create a vault using the Azure Command-Line Interface, you need to provide some information:

- A unique name. For this example, we will use `contosovault`.
- A resource group. Here, we are using `SecurityGroup`.
- A location. We will use `West US`.

```
az keyvault create --name contosovault --resource-group SecurityGroup
--location westus
```

The output of this cmdlet shows properties of the newly created vault. Take note of the two properties listed below:

- **Vault Name:** In the example, this is `contosovault`. You will use this name for other Key Vault commands.
- **Vault URI:** In the example, this is `https://contosovault.vault.azure.net/`. Applications that use your vault through its REST API must use this URI.

At this point, your Azure account is the only one authorized to perform any operations on this new vault.

To add a secret to the vault, you just need to take a couple of additional steps. This password could be used by an application. The password will be called DatabasePassword and will store the value of Pa5w.rd in it:

```
az keyvault secret set --vault-name contosovault --name DatabasePassword
--value 'Pa5w.rd'
```

To view the value contained in the secret as plain text:

```
az keyvault secret show --vault-name contosovault --name DatabasePassword
```

Review Questions

Module 5 Review Questions

Azure SQL Database encryption

You manage several SQL Server instance for your organization.

You must encrypt all data at rest.

What should you implement? How does encryption of SQL databases affect the amount of storage space that is used?

Suggested Answer ↓

TDE encrypts SQL Server, Azure SQL Database, and Azure SQL Data Warehouse data files. TDE performs real-time I/O encryption and decryption of the data and log files.

Always Encrypted is a new data encryption technology in Azure SQL Database and SQL Server that helps protect sensitive data at rest on the server, during movement between client and server, and while the data is in use, helping to ensure that sensitive data never appears as plaintext inside the database system.

SSL and TLS overview

You manage an application in Azure.

The application must communicate securely with users inside the corporate network.

You have hired an outside security consultant to perform a vulnerability analysis of your application, and the results show a concern regarding secure communications.

What should you do Which cryptographic protocols should be enabled?

Suggested Answer ↓

SSL 1.0, 2.0 and 3.0 have been found to be vulnerable, and they have been prohibited by an Internet Engineering Task Force (IETF) Request For Comments (RFC).

For these reasons, the Azure Storage team has determined that TLS 1.2 is the best protocol to use when connecting to Azure Storage accounts.

Azure Key Vault

You manage several applications in Azure. Each application has unique credentials to access content and enable communication with internal resources.

You need to ensure that all authentication information is securely stored.

What should you use to secure the information?

Suggested Answer ↓

Azure Key Vault is a cloud service that works as a security-enhanced secrets store.
Key Vault allows you to create multiple security-enhanced containers, called vaults. These vaults are

backed by hardware security modules (HSMs). Vaults help to reduce the chance of accidentally losing security information by centralizing the storage of application secrets. Vaults also control and log the access to anything stored in them. Azure Key Vault is designed to support any type of secret, such as a password, database credential, API key, or certificate.

Module 6 Module Business Continuity and Resiliency in Azure

Business Continuity and Resiliency

Business Continuity and Resilience in Azure

NOTE: The content in this module serves as an informal checklist of considerations for sustaining business continuity and resilience in Azure and is the result of real-world implementations.

Business continuity represents the ability to perform essential business functions during and after adverse conditions such as a natural disaster or a failure of a service. It covers the entire operation of the business including physical facilities, people, communications, transportation, and technology.

This module covers technical aspects of business continuity and it's important to remember that technology must be considered in the context of overall business continuity strategy.

A technical strategy for business continuity helps you ensure that your internal and external applications, workloads, and services are resilient by remaining operational during planned downtime and unplanned outages. Such resiliency must also ensure that business-critical data is backed up and stored in a secure location, and that the data can be recovered within a reasonable amount of time when an unexpected incident or a disaster occurs.

Architecting for resiliency in a cloud environment focuses on failure recovery, rather than on avoiding failures. Its goal is to respond to failures in a way that avoids downtime or data loss and returns applications to a fully functioning state following a failure.

High Availability and Disaster Recovery

High Availability and Disaster Recovery

Two essential aspects of resiliency are high availability and disaster recovery:

- **High availability (HA)** is the ability of the application to continue running in a healthy state despite localized or transient failures. Typically, high availability relies on redundancy of application components and automatic failover.
- **Disaster recovery (DR)** is the ability to recover from major incidents, such as service disruption that affects an entire region. Disaster recovery provisions include data backup and archiving, and may require manual intervention, such as restoring a database from backup.

When designing for resiliency, you must understand the availability requirements. This is partially a function of cost incurred due to potential downtime, which, in turn, impacts the budget allocated to implementing HA and DR provisions. You also have to identify what constitutes application availability. For example, an order processing application might be considered operational if a customer is able to submit an order, even if such order cannot be immediately processed. In addition, you should consider the frequency with which a particular type of failure might occur.

Data Backup

Data backup is a critical part of DR. If the stateless components of an application fail, you can always redeploy them. But if data is lost, the system can't return to a previous stable state. Data must be backed up and, whenever possible, stored in a remote location in order to protect against regional disasters.

Backup is distinct from data replication. Data replication involves copying data in near-real-time (either synchronously or asynchronously), so that the system can fail over quickly to a replica. Data replication can reduce the time it takes to recover from an outage, by ensuring that a replica of the data is readily available. However, data replication should not be considered as a substitute to backups. For example, any data corruption will be automatically copied to the replicas, rendering their content unusable. Effectively, even with data replication in place, you still need to include backup in your DR strategy

Resiliency

Identifying Requirements

Resiliency Checklist

Use the following checklist to incorporate resiliency requirements into your application throughout its lifecycle.

Identifying Requirements
Identify the expected recovery time objective and recovery point objective:

- **Recovery time objective (RTO)** is the maximum acceptable time that an application can be unavailable after an incident. If your RTO is 90 minutes, you must be able to restore the application to a running state within 90 minutes from the start of a disaster. If you have a very low RTO, you might keep a second deployment running in the standby mode.
- **Recovery point objective (RPO)** is the maximum duration of data loss that is acceptable during a disaster. For example, if you store data in a single database, with no replication to other databases, and perform hourly backups, you could lose up to an hour worth of data.

RTO and RPO are business requirements. Conducting a risk assessment can help you define the application's RTO and RPO. Another common metric is mean time to recover (MTTR), which is the average time that it takes to restore the application after a failure. MTTR represents an empirical fact about a system. If MTTR exceeds the RTO, then a failure of the system represents an unacceptable business disruption, because the system restore time exceeds the defined RTO.

Identify the expected and the actual Service Level Agreements. In Azure, a Service Level Agreement (SLA) describes Microsoft's commitments to maintain uptime and connectivity. If the SLA for a particular service is 99.9%, it means you should expect the service to be available 99.9% of the time. The Azure SLAs also include provisions for obtaining a service credit if the SLA is not met, along with specific definitions of "availability" for each service. That aspect of the SLA acts as an enforcement policy.

You should identify the expected target SLAs for each workload in your solution. An SLA makes it possible to evaluate whether the architecture meets the business requirements. For example, if a workload requires 99.99% uptime, but depends on a service with a 99.9% SLA, that service cannot be a single-point of failure in the system. One remedy is to have a fallback path in case the service fails, or take other measures to recover from a failure of that service.

See the next topic for estimating SLA downtime.

Estimating SLA Downtime

The following table shows the maximum cumulative downtime for various SLA levels.

SLA	Downtime per week	Downtime per month	Downtime per year
99%	1.68 hours	7.2 hours	3.65 days
99.9%	10.1 minutes	43.2 minutes	8.76 hours
99.95%	5 minutes	21.6 minutes	4.38 hours
99.99%	1.01 minutes	4.32 minutes	52.56 minutes
99.999%	6 seconds	25.9 seconds	5.26 minutes

Whenever applicable, implement composite SLAs. For example, consider an App Service web app that writes to Azure SQL Database. As of December 2018, these Azure services have the following SLAs:

- **App Service Web Apps** = 99.95%
- **SQL Database** = 99.99%

If either service fails, the whole application fails. In general, the probability of each service failing is independent, so the composite SLA for this application is 99.95% × 99.99% = 99.94%. That's lower than the individual SLAs, which isn't surprising, because an application that relies on multiple services has more potential failure points.

Alternatively, you can improve the composite SLA by creating independent fallback paths. For example, if SQL Database is unavailable, you can store transactions into a queue, to be processed later. As of December 2018, Azure Service Bus Queues have the 99.9% availability SLA.

With this design, the application is still available even if it can't connect to the database. However, it fails if the database and the queue both fail at the same time. The expected percentage of time for a simultaneous failure is 0.0001 × 0.001, so the composite SLA for this combined path is:

- **Database OR queue** = 1.0 − (0.0001 × 0.001) = 99.99999%

Effectively, the total composite SLA is:

- **Web app AND (database OR queue)** = 99.95% × 99.99999% = ~99.95%

NOTE: There are tradeoffs to this approach. The application logic is more complex, you are paying for the queue, and there may be data consistency issues to consider.

Take into account SLAs for multi-region deployments. Another HA technique is to deploy the application in more than one region, and use Azure Traffic Manager to fail over if the application fails in one region.

For a two-region deployment, the composite SLA is calculated as follows:

- Let N be the composite SLA for the application deployed in one region. The expected chance that the application will fail in both regions at the same time is (1 − N) × (1 − N).
- Combined SLA for both regions = 1 − (1 − N)(1 − N) = N + (1 − N)N
- To calculate the composite SLA, you must factor in the SLA for Traffic Manager. As of December 2018, the SLA for Traffic Manager SLA is 99.99%:
- Composite SLA = 99.99% × (combined SLA for both regions)

It is important to remember that Traffic Manager-based failover is not instantaneous and can result in some additional downtime.

NOTE: The calculated SLA number is a useful baseline, but it doesn't tell the whole story about availability. Often, an application can degrade gracefully when a non-critical path fails. Consider an application that shows a catalog of books. If the application can't retrieve the thumbnail image for the cover, it might show a placeholder image. In that case, failing to get the image does not reduce the application's uptime, although it affects the user experience.

Identify the intended usage patterns. For example, a tax-filing service needs to be available before the filing deadline and a video streaming service must stay up during major sports events. During these critical periods, you might have redundant deployments across several regions, so the application could fail over following a failure in the primary location. However, a multi-region deployment is more expensive, so during less critical times, you might run the application in a single region.

Application Design

Failure Mode Analysis (FMA)

Perform a failure mode analysis (FMA) for your application. FMA is a process for building resiliency into an application early in the design stage. The goals of an FMA include:

- Identify what types of failures an application might experience.
- Capture the potential effects and impact of each type of failure on the application.
- Identify recovery strategies.

For example, for calls to an external web service / API, you could consider the following points of failure:

Failure mode	Detection strategy
Service is unavailable	HTTP 5xx
Throttling	HTTP 429 (Too Many Requests)
Authentication	HTTP 401 (Unauthorized)
Slow response	Request times out

Avoiding a Single Point of Failure

Avoiding a Single Point of Failure for Applications

Avoid any single point of failure. All components, services, resources, and compute instances should be deployed as multiple instances to prevent a single point of failure from affecting availability. This includes authentication mechanisms. Design the application to be configurable to use multiple instances, and to automatically detect failures and redirect requests to non-failed instances where the platform does not do this automatically.

Azure has a number of features to make an application redundant at every level of failure, from an individual VM to an entire region.

- **Single VM.** Azure provides an uptime SLA for individual VMs, as long as all disks of these VMs are configured to use Premium Storage. Although you can get a higher SLA by running two or more VMs, a single VM may be reliable enough for some workloads. For production workloads, we recommend using two or more VMs for redundancy.
- **Availability sets.** To protect against localized hardware failures, such as a power unit or a network switch failing, deploy two or more VMs in an availability set. An availability set consists of two or more fault domains, each of which uses a separate power source and network switch. VMs in an availability set are distributed across the fault domains, so localized a hardware failure affects only one fault domain.
- **Availability zones.** An Availability Zone is a separate physical datacenter within an Azure region. Each Availability Zone has a distinct power source, network, and cooling. Deploying VMs across availability zones helps to protect an application against datacenter-wide failures.

- **Azure Site Recovery.** Replicate Azure virtual machines to another Azure region for business continuity and disaster recovery needs. You can conduct periodic DR drills to ensure you meet the compliance needs. The VM will be replicated with the specified settings to the selected region so that you can recover your applications in the event of outages in the source region.
- **Paired regions**. Each Azure region is paired with another region. With the exception of Brazil South, regional pairs are located within the same geography in order to meet data residency requirements for tax and law enforcement jurisdiction purposes.

Auto Scaling and Load Balancing

Azure Autoscaling and Load Balancing

Use autoscaling to respond to increases in load. If your application is not configured to scale out automatically as load increases, it's possible that your application's services will fail if they become saturated with user requests. When implementing Azure App Service, use the Standard, Premium, or Isolated tier.

Load balance across instances. For scalability, a cloud application should be able to scale out by adding more instances. This approach also improves resiliency, because unhealthy instances can be removed from rotation. Some of the more common examples of this approach include:

- **Placing two or more VMs behind a load balancer**. The load balancer distributes traffic to all the VMs. If you choose Azure Application Gateway, remember that you need to provision two or more Application Gateway instances to qualify for the availability SLA.
- **Scaling out an Azure App Service app to multiple instances**. App Service automatically balances load across instances.
- **Using Azure Traffic Manager** to distribute traffic across a set of endpoints.

Multi-Region Deployment

Consider deploying your application across multiple regions. A multi-region deployment can use an active-active pattern (distributing requests across multiple active instances) or an active-passive pattern (keeping a "warm" instance in reserve, in case the primary instance fails).

Use Azure Traffic Manager to route your application's traffic to different regions. Azure Traffic Manager performs load balancing at the DNS level and will route traffic to different regions based on the traffic routing method you specify and the health of your application's endpoints.

Configure and test health probes for your load balancers and traffic managers. Ensure that your health logic checks the critical parts of the system and responds appropriately to health probes. The health probes for Azure Traffic Manager and Azure Load Balancer serve a specific function. For Traffic Manager, the health probe determines whether to fail over to another region. For a load balancer, it determines whether to remove a VM from rotation.

	Availability Set	Availability Zone	Azure Site Recovery/ Paired region
Scope of failure	Rack	Datacenter	Region
Request routing	Load Balancer	Cross-zone Load Balancer	Traffic Manager, Azure Front Door
Network latency	Very low	Low	Mid to high
Virtual network	VNet	VNet	Cross-region VNet peering

Workload Service-Level Objectives

Configure workloads by service-level objectives. If a service is composed of critical and less-critical workloads, manage them differently and specify the service features and number of instances to meet their availability requirements. The term "workload" means a discrete capability or computing task, which can be logically separated from other tasks, in terms of business logic and data storage requirements.

For example, an e-commerce app might include the following workloads:

- Browse and search a product catalog.
- Create and track orders.
- View recommendations.

These workloads might have different requirements for availability, scalability, data consistency, and disaster recovery. These requirements should be driven based on their business relevance.

For Azure App Service, separate web apps from web APIs. If your solution has both a web front-end and a web API, consider decomposing them into separate App Service apps. This way you can run the web app and the API in separate App Service plans, so they can be scaled independently.

Minimize and understand service dependencies. Minimize the number of different services used where possible, and ensure you understand all of the feature and service dependencies that exist in the system. This includes the nature of these dependencies, and the impact of failure or reduced performance in each one on the overall application.

Design tasks and messages to be idempotent where possible. An operation is idempotent if it can be repeated multiple times and produce the same result. For example, in messaging scenarios, consumers and the operations they carry out should be idempotent so that repeating a previously executed operation does not render the results invalid. This may mean detecting duplicated messages, or ensuring consistency by using an optimistic approach to handling conflicts.

Enhancing Security

Enhancing security

Ensure application-level protection against distributed denial of service (DDoS) attacks. Azure services are protected against DDoS attacks at the network layer. However, Azure cannot protect against application-layer attacks, because it is difficult to distinguish between true user requests from malicious user requests.

Adhere to the principle of least privilege for access to the application's resources. The default for access to the application's resources should be as restrictive as possible. Grant higher level permissions on an approval basis. Granting overly permissive access to your application's resources by default can result in someone purposely or accidentally deleting resources. Azure provides role-based access control to manage user privileges, but it's important to verify least privilege permissions for other resources that have their own permissions systems such as SQL Server.

Additional Resiliency Tips

Use a message broker that implements high availability for critical transactions. Many cloud applications use messaging to initiate tasks that are performed asynchronously. To guarantee delivery of messages, the messaging system should provide high availability. Azure Service Bus Messaging implements at least once semantics. This means that a message posted to a queue will not be lost, although

duplicate copies may be delivered under certain circumstances. To account for duplicates, ensure that message processing is idempotent.

Design applications to gracefully degrade. The load on an application may exceed the capacity of one or more of its parts, causing reduced availability and failed connections. Scaling can help to alleviate this, but it may reach a limit imposed by other factors, such as resource availability or cost. When an application reaches a resource limit, it should take appropriate action to minimize the impact for the user. For example, in an ecommerce system, if the order-processing subsystem is under strain or fails, it can be temporarily disabled while allowing other functionality, such as browsing the product catalog. It might be appropriate to postpone requests to a failing subsystem, for example still enabling customers to submit orders but saving them for later processing, when the orders subsystem is available again.

Throttle high-volume users. Sometimes a small number of users create excessive load. That can have an impact on other users, reducing the overall availability of your application. When a single client makes an excessive number of requests, the application might throttle the client for a certain period of time. During the throttling period, the application refuses some or all of the requests from that client (depending on the exact throttling strategy). The threshold for throttling might depend on the customer's service tier. Throttling does not necessarily imply that the client was acting maliciously, only that it exceeded its service quota. If consumers consistently exceed their quota or otherwise behave badly, you might consider blocking access by applying an API key-based protection or IP address range filtering.

Use load leveling to smooth spikes in traffic. Applications may experience sudden spikes in traffic, which can overwhelm services on the backend. If a backend service cannot respond to requests quickly enough, it may cause requests to queue or cause the service to throttle the application. To avoid this, you can use a queue as a buffer that smooths out peaks in the load. When there is a new work item, instead of calling the backend service immediately, the application queues a work item to run asynchronously.

Monitor third-party services. If your application has dependencies on third-party services, identify where and how these third-party services can fail and what effect those failures will have on your application. A third-party service may not include monitoring and diagnostics, so it's important to log your invocations of them and correlate them with your application's health and diagnostic logging using a unique identifier.

Implement resiliency patterns for remote operations where appropriate. If your application depends on communication between remote services, follow design patterns for dealing with transient failures, such as the Retry pattern and the Circuit Breaker pattern.

- **Retry pattern.** Transient failures can be caused by momentary loss of network connectivity, a dropped database connection, or a timeout when a service is busy. Often, a transient failure can be resolved simply by retrying the request. For many Azure services, the client SDK implements automatic retries, in a way that is transparent to the caller. Each retry attempt increases the total latency. Additionally, too many failed requests can cause a bottleneck, as pending requests accumulate in the queue. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on, which can cause cascading failures. To avoid this, increase the delay between each retry attempt, and limit the total number of failed requests.
- **Circuit Breaker pattern.** The Circuit Breaker pattern can prevent an application from repeatedly trying an operation that is likely to fail. The circuit breaker wraps calls to a service and tracks the number of recent failures. If the failure count exceeds a threshold, the circuit breaker starts returning an error code without calling the service. This gives the service time to recover.

Implement asynchronous operations whenever possible. Synchronous operations can monopolize resources and block other operations while the caller waits for the process to complete. Design each part of your application to allow for asynchronous operations whenever possible.

Apply compensating transactions. A compensating transaction is a transaction that undoes the effects of another completed transaction. In a distributed system, it can be very difficult to achieve strong transactional consistency. Compensating transactions are a way to achieve consistency by using a series of smaller, individual transactions that can be undone at each step. For example, to book a trip, a customer might reserve a car, a hotel room, and a flight. If any of these steps fails, the entire operation fails. Instead of trying to use a single distributed transaction for the entire operation, you can define a compensating transaction for each step. For example, to undo a car reservation, you cancel the reservation. In order to complete the whole operation, a coordinator executes each step. If any step fails, the coordinator applies compensating transactions to undo any steps that were completed.

Testing, Deployment, and Maintenance

Deployment and Maintenance Tasks

Automate and test deployment and maintenance tasks. Distributed applications consist of multiple parts that must work together. The deployment process should be predictable and repeatable. In Azure, this process might include provisioning Azure resources, deploying application code, and applying configuration settings:

- Use Azure Resource Manager templates to automate provisioning of Azure resources.
- Use Azure Automation Desired State Configuration (DSC) to configure VMs.
- Use an automated deployment process for application code.

For App Service deployments, store configuration as app settings. Define the settings in your Resource Manager templates, or by using PowerShell, so that you can apply them as part of an automated deployment / update process, which is more reliable.

Give resources meaningful names. Giving resources meaningful names makes it easier to locate a specific resource and understand its role.

Organize resource groups by function and lifecycle. In general, a resource group should contain resources that share the same lifecycle. This makes it easier to manage deployments, delete test deployments, and assign access rights, reducing the chance that a production deployment is accidentally deleted or modified. Create separate resource groups for production, development, and test environments. In a multi-region deployment, put resources for each region into separate resource groups. This makes it easier to redeploy one region without affecting the other region(s).

Infrastructure as Code and Immutable Infrastructure

Below are summaries forthe principles of infrastructure as code and immutable infrastructure:

- **Infrastructure as code is the practice of using code to provision and configure infrastructure.** Infrastructure as code may use a declarative approach or an imperative approach (or a combination of both). Resource Manager templates constitute an example of a declarative approach. PowerShell scripts constitute an example of an imperative approach.
- **Immutable infrastructure is the principle that you shouldn't modify infrastructure after it's deployed to production**. Otherwise, you can get into a state where ad hoc changes have been applied, so it's hard to know exactly what changed, and hard to reason about the system.

Use staging and production features of the platform. For example, Standard, Premium, and Isolated tiers of Azure App Service support deployment slots, which you can use to stage a deployment before swapping it to production. Azure Service Fabric supports rolling upgrades to application services.

Maximize Application Availability

Design your release process to maximize application availability. If your release process requires services to go offline during deployment, your application will be unavailable until they come back online. Use the blue/green or canary release deployment technique to deploy your application to production. Both of

these techniques involve deploying your release code alongside production code so users of release code can be redirected to production code in the event of a failure:

- **Blue-green deployment** is a technique where an update is deployed into a production environment separate from the live application. After you validate the deployment, switch the traffic routing to the updated version. For example, Azure App Service Web Apps enables this with staging slots.
- **Canary releases are similar to blue-green deployments.** Instead of switching all traffic to the updated version, you roll out the update to a small percentage of users, by routing a portion of the traffic to the new deployment. If there is a problem, back off and revert to the old deployment. Otherwise, route more of the traffic to the new version, until it gets 100% of the traffic.

Additional Considerations for Testing, Deployment, and Maintenance

Have a rollback plan for deployment. It's possible that your application deployment could fail and cause your application to become unavailable. Design a rollback process to go back to a last known good version and minimize downtime.

Ensure that your application does not run up against Azure subscription limits. Azure subscriptions have limits on certain resource types, such as number of resource groups, number of cores, and number of storage accounts. If your application requirements exceed Azure subscription limits, create another Azure subscription and provision sufficient resources there.

Ensure that your application does not run up against per-service limits. Individual Azure services have consumption limits — for example, limits on storage, throughput, number of connections, requests per second, and other metrics. Your application will fail if it attempts to use resources beyond these limits. This will result in service throttling and possible downtime for affected users. Depending on the specific service and your application requirements, you can often avoid these limits by scaling up (for example, choosing another pricing tier) or scaling out (adding new instances).

Perform fault injection testing of your applications. Test the resiliency of the system during failures, either by triggering actual failures or by simulating them. Your application can fail for many different reasons, such as certificate expiration, exhaustion of system resources in a VM, or storage failures. Test your application in an environment as close as possible to production, by simulating or triggering real failures. For example, delete certificates, artificially consume system resources, or delete a storage source. Verify your application's ability to recover from all types of faults, alone and in combination. Check that failures are not propagating or cascading through your system.

Perform load testing of your applications. Load testing is crucial for identifying failures that only happen under load, such as the backend database being overwhelmed or service throttling. Test for peak load, using production data or synthetic data that is as close to production data as possible. The goal is to see how the application behaves under real-world conditions.

Run tests in production using both synthetic and real user data. Test and production are rarely identical, so it's important to use blue/green or a canary deployment and test your application in production. This allows you to test your application in production under real load and ensure it will function as expected when fully deployed.

Establish a process for interacting with Azure support. If the process for contacting Azure support is not set before the need to contact support arises, downtime will be prolonged as the support process is navigated for the first time. Include the process for contacting support and escalating issues as part of your application's resiliency from the outset.

Use resource locks for critical resources, such as VMs. Resource locks prevent an operator from accidentally deleting a resource.

Data Management

Replicating Data

Data Management

Replicating data is a general strategy for handling non-transient failures in a data store. Many storage technologies provide built-in replication. It's important to consider both the read and write paths. Depending on the storage technology, you might have multiple writable replicas, or a single writable replica and multiple read-only replicas. To maximize availability, replicas can be placed in multiple regions. However, this increases the latency when replicating the data. Typically, replicating across regions is done asynchronously, which implies an eventual consistency model and potential data loss if a replica fails.

- **Geo-replicate databases.** Azure SQL Database and Azure Cosmos DB both support geo-replication, which enables you to configure secondary database replicas in other regions. Secondary databases are available for querying and for failover in the case of a data center outage or the inability to connect to the primary database. With Azure SQL Database, you can create auto-failover groups, which facilitate automatic failover. Azure Cosmos DB additionally supports multi-master configuration, with multiple write regions and customizable conflict resolution mechanism.
- **Geo-replicate data in Azure Storage.** Data in Azure Storage is automatically replicated within a datacenter. For higher availability, use Read-access geo-redundant storage (RA-GRS), which replicates your data to a secondary region and provides read-only access to the data in that region. The data is durable even in the case of a complete regional outage or a disaster.
- **For VMs, do not rely on RA-GRS replication to restore the VM disks (VHD files).** Instead, use Azure Backup. In addition, consider using managed disks. Managed disks provide enhanced resiliency for VMs in an availability set, because the disks are sufficiently isolated from each other to avoid single points of failure. In addition, managed disks eliminate the need to account for the storage account-level IOPS limits.

Additional Data Management Considerations

Below are additional considerations for managing data.

- **Sharding.** Consider using sharding to partition a database horizontally. Sharding can provide fault isolation and eliminate constraints imposed by database size limits.
- **Optimistic concurrency and eventual consistency.** Transactions that block access to resources through locking (pessimistic concurrency) can cause poor performance and considerably reduce availability. These problems can become especially acute in distributed systems. In many cases, careful design and techniques such as partitioning can minimize the chances of conflicting updates. Where data is replicated, or is read from a separately updated store, the data will only be eventually consistent. But the advantages usually far outweigh the impact on availability of using transactions to ensure immediate consistency.
- **Document data source fail over and fail back processes, and then test it.** In the case where your data source fails catastrophically, a human operator will have to follow a set of documented instructions to fail over to a new data source. Regularly test the instruction steps to verify that an operator following them is able to successfully fail over and fail back the data source.

- **Periodic backup and point-in-time restore.** Regularly and automatically back up data and verify you can reliably restore both the data and the application. Ensure that backups meet your Recovery Point Objective (RPO). The backup process must be secure to protect the data in transit and at rest.
- **Ensure that no single user account has access to both production and backup data.** Your data backups are compromised if one single user account has permission to write to both production and backup sources. A malicious user could purposely delete all your data, while a regular user could accidentally delete it. Design your application to limit the permissions of each user account so that only the users that require write access have write access and it's only to either production or backup, but not both.
- **Validate your data backups.** Regularly verify that your backup data is what you expect by running a script to validate data integrity, schema, and queries. There's no point having a backup if it's not useful to restore your data sources. Log and report any inconsistencies so the backup service can be repaired.

Monitoring and Disaster Recovery

Best Practices for Monitoring and Alerting Applications

Without proper monitoring, diagnostics, and alerting, there is no way to detect a failure in your application or alert an operator to resolve the failure. Below is a list best practices for monitoring and alerting applications:

Implement best practices for monitoring and alerting in your application. Without proper monitoring, diagnostics, and alerting, there is no way to detect failures in your application and alert an operator to fix them.

Measure remote call statistics and make the information available to the application team. If you don't track and report remote call statistics in real time and provide an easy way to review this information, the operations team will not have an instantaneous view into the health of your application. And if you only measure average remote call time, you will not have enough information to reveal issues in the services. Summarize remote call metrics such as latency, throughput, and errors in the 99 and 95 percentiles. Perform statistical analysis on the metrics to uncover errors that occur within each percentile.

Track the number of transient exceptions and retries over an appropriate timeframe. If you don't track and monitor transient exceptions and retry attempts over time, it's possible that an issue or failure could be hidden by your application's retry logic.

Track the progress of long-running workflows and retry on failure. Long-running workflows are often composed of multiple steps. Ensure that each step is independent and can be retried to minimize the chance that the entire workflow will need to be rolled back, or that multiple compensating transactions need to be executed. Monitor and manage the progress of long-running workflows by implementing a pattern such as Scheduler Agent Supervisor pattern.

Implement an early warning system that alerts an operator. Identify the key performance indicators of your application's health, such as transient exceptions and remote call latency, and set appropriate threshold values for each of them. Send an alert to operations when the threshold value is reached. Set these thresholds at levels that identify issues before they become critical and require a recovery response.

Implement application logging. Application logs are an important source of diagnostics data. The recommended practices for application logging include:

- Log in production.
- Log events at service boundaries. Include a correlation ID that flows across service boundaries. If a transaction flows through multiple services and one of them fails, the correlation ID will help you pinpoint why the transaction failed.
- Use semantic logging, also known as structured logging. Unstructured logs make it hard to automate the consumption and analysis of the log data, which is needed at cloud scale.
- Use asynchronous logging. With synchronous logging, the logging system might cause the application to fail, as incoming requests are blocked while waiting for log writes.

Implement logging using an asynchronous pattern. If logging operations are synchronous, they might block your application code. Ensure that your logging operations are implemented as asynchronous operations.

Test the Monitoring Systems

Automated failover and fallback systems, and manual visualization of system health and performance by using dashboards, all depend on monitoring and instrumentation functioning correctly. If these elements fail, miss critical information, or report inaccurate data, an operator might not realize that the system is unhealthy or failing.

Plan for and test disaster recovery. Create an accepted, fully-tested plan for recovery from any type of failure that may affect system availability. Choose a multi-site disaster recovery architecture for any mission-critical applications. Identify a specific owner of the disaster recovery plan, including automation and testing. Ensure the plan is well-documented, and automate the process as much as possible. Establish a backup strategy for all reference and transactional data, and test the restoration of these backups regularly. Train operations staff to execute the plan, and perform regular disaster simulations to validate and improve the plan. If you are using Azure Site Recovery to replicate VMs, create a fully automated recovery plan to failover the entire application within minutes.

Implement operational readiness testing. If your application fails over to a secondary region, you should perform an operational readiness test before you fail back to the primary region. The test should verify that the primary region is healthy and ready to receive traffic again.

Perform data consistency checks. If a failure happens in a data store, there may be data inconsistencies when the store becomes available again, especially if the data was replicated.