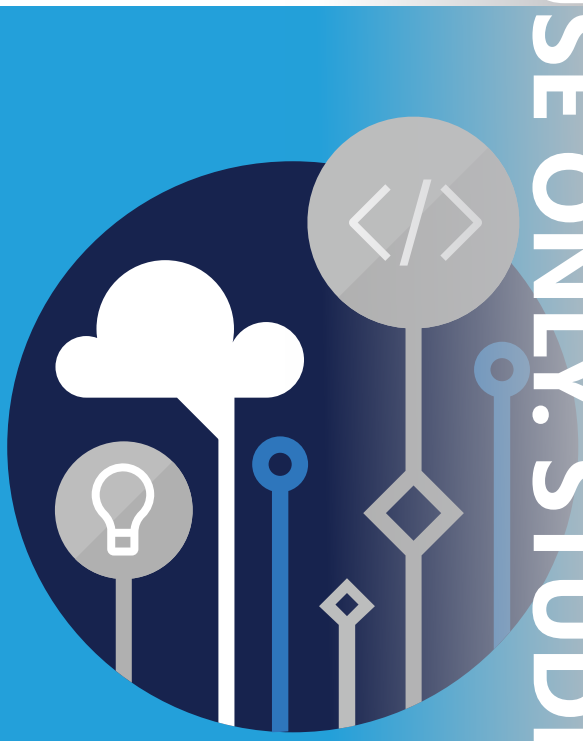
Microsoft

Microsoft Official Course



AZ-203T04

Implement Azure Security

AZ-203T04
Implement Azure Security

Contents

Module 0 Welcome to the course

Start Here

Welcome

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

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

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

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

Exam study areas

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

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

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

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

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

Course description

In this course students will gain the knowledge and skills needed to include Azure authentication and authorization services in their development solutions. Students will learn how identity is managed and utilized in Azure solutions by using the Microsoft identity platform. Students will also learn about access control (claims-based authorization and role-based access control) and how to implement secure data solutions. Throughout the course students learn how to create and integrate these resources by using the Azure CLI, REST, and application code.

Level: Intermediate

Audience:

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

Course Syllabus

Module 1: Implement authentication

- Microsoft identity platform
- Implement OAuth2 authentication
- Implement managed identities for Azure resources
- Implement authentication by using certificates, forms-based authentication, or tokens
- Implement multi-factor authentication

Module 2: Implement access control

- Claims-based authorization
- Role-based access control (RBAC) authorization

Module 3: Implement secure data solutions

- Encryption options
- End-to-end encryption
- Implement Azure confidential computing
- Manage cryptographic keys in Azure Key Vault

Module 1 Implement authentication

Microsoft identity platform

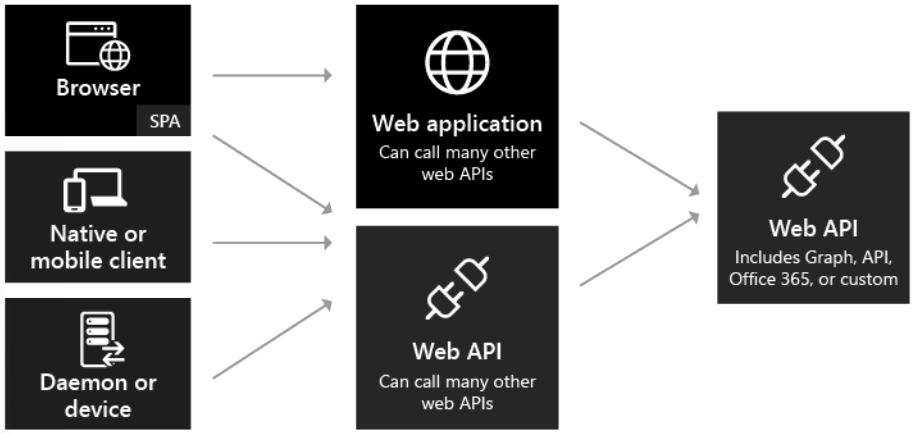
Microsoft identity platform overview

Microsoft identity platform is an evolution of the Azure Active Directory (Azure AD) identity service and developer platform. It allows developers to build applications that sign in all Microsoft identities, get tokens to call Microsoft Graph, other Microsoft APIs, or APIs that developers have built. It's a full-featured platform that consists of an authentication service, open-source libraries, application registration and configuration (through a developer portal and application API), full developer documentation, code samples, and other developer content. The Microsoft identity platform supports industry standard protocols such as OAuth 2.0 and OpenID Connect.

Application types in Azure AD

Azure Active Directory (Azure AD) supports authentication for a variety of modern app architectures, all of them based on industry-standard protocols OAuth 2.0 or OpenID Connect.

The following diagram illustrates the scenarios and application types, and how different components can be added:



These are the five primary application scenarios supported by Azure AD:

- **Single-page application (SPA):** A user needs to sign in to a single-page application that is secured by Azure AD.
- **Web browser to web application:** A user needs to sign in to a web application that is secured by Azure AD.
- **Native application to web API:** A native application that runs on a phone, tablet, or PC needs to authenticate a user to get resources from a web API that is secured by Azure AD.
- **Web application to web API:** A web application needs to get resources from a web API secured by Azure AD.
- **Daemon or server application to web API:** A daemon application or a server application with no web user interface needs to get resources from a web API secured by Azure AD.

App registration

Any application that outsources authentication to Azure AD must be registered in a directory. This step involves telling Azure AD about your application, including the URL where it's located, the URL to send replies after authentication, the URI to identify your application, and more. This information is required for a few key reasons:

- Azure AD needs to communicate with the application when handling sign-on or exchanging tokens. The information passed between Azure AD and the application includes the following:

    - **Application ID URI** - The identifier for an application. This value is sent to Azure AD during authentication to indicate which application the caller wants a token for. Additionally, this value is included in the token so that the application knows it was the intended target.
    - **Reply URL and Redirect URI** - For a web API or web application, the Reply URL is the location where Azure AD will send the authentication response, including a token if authentication was successful. For a native application, the Redirect URI is a unique identifier to which Azure AD will redirect the user-agent in an OAuth 2.0 request.
    - **Application ID** - The ID for an application, which is generated by Azure AD when the application is registered. When requesting an authorization code or token, the Application ID and Key are sent to Azure AD during authentication.

- **Key** - The key that is sent along with an Application ID when authenticating to Azure AD to call a web API.

- Azure AD needs to ensure the application has the required permissions to access your directory data, other applications in your organization, and so on.

Single-tenant and multi-tenant apps

Provisioning becomes clearer when you understand that there are two categories of applications that can be developed and integrated with Azure AD:

- **Single tenant application** - A single tenant application is intended for use in one organization. These are typically line-of-business (LoB) applications written by an enterprise developer. A single tenant application only needs to be accessed by users in one directory, and as a result, it only needs to be provisioned in one directory. These applications are typically registered by a developer in the organization.

- **Multi-tenant application** - A multi-tenant application is intended for use in many organizations, not just one organization. These are typically software-as-a-service (SaaS) applications written by an independent software vendor (ISV). Multi-tenant applications need to be provisioned in each directory where they will be used, which requires user or administrator consent to register them. This consent process starts when an application has been registered in the directory and is given access to the Graph API or perhaps another web API. When a user or administrator from a different organization signs up to use the application, they are presented with a dialog that displays the permissions the application requires. The user or administrator can then consent to the application, which gives the application access to the stated data, and finally registers the application in their directory.

Additional considerations when developing single tenant or multi-tenant apps

Some additional considerations arise when developing a multi-tenant application instead of a single tenant application. For example, if you are making your application available to users in multiple directories, you need a mechanism to determine which tenant they're in. A single tenant application only needs to look in its own directory for a user, while a multi-tenant application needs to identify a specific user from all the directories in Azure AD. To accomplish this task, Azure AD provides a common authentication endpoint where any multi-tenant application can direct sign-in requests, instead of a tenant-specific endpoint. This endpoint is `https://login.microsoftonline.com/common` for all directories in Azure AD, whereas a tenant-specific endpoint might be `https://login.microsoftonline.com/contoso.onmicrosoft.com`. The common endpoint is especially important to consider when developing your application because you'll need the necessary logic to handle multiple tenants during sign-in, sign-out, and token validation.

If you are currently developing a single tenant application but want to make it available to many organizations, you can easily make changes to the application and its configuration in Azure AD to make it multi-tenant capable. In addition, Azure AD uses the same signing key for all tokens in all directories, whether you are providing authentication in a single tenant or multi-tenant application.

Application and service principal objects in Azure Active Directory

Sometimes, the meaning of the term “application” can be misunderstood when used in the context of Azure Active Directory (Azure AD). This article clarifies the conceptual and concrete aspects of Azure AD application integration, with an illustration of registration and consent for a multi-tenant application.

Overview

An application that has been integrated with Azure AD has implications that go beyond the software aspect. “Application” is frequently used as a conceptual term, referring to not only the application software, but also its Azure AD registration and role in authentication/authorization "conversations" at runtime.

By definition, an application can function in these roles:

- Client role (consuming a resource)
- Resource server role (exposing APIs to clients)
- Both client role and resource server role

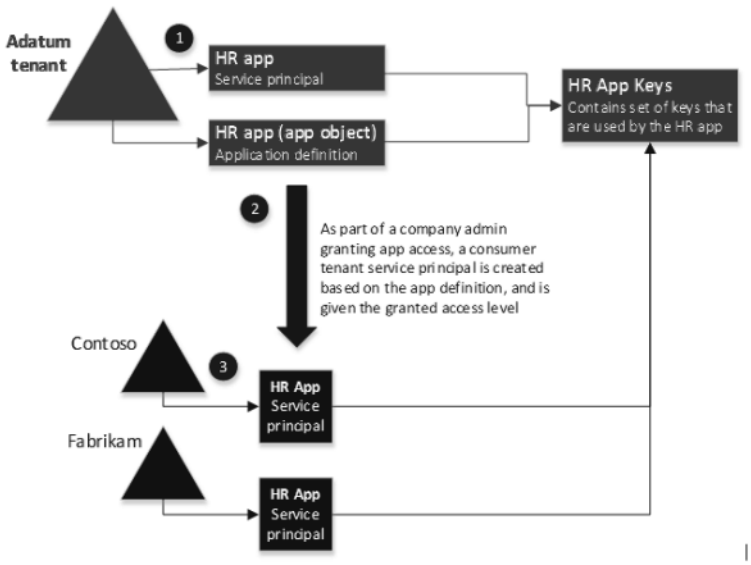
An OAuth 2.0 Authorization Grant flow defines the conversation protocol, which allows the client/resource to access/protect a resource's data, respectively.

In the following sections, you'll see how the Azure AD application model represents an application at design-time and run-time.

Example

The following diagram illustrates the relationship between an application's application object and corresponding service principal objects, in the context of a sample multi-tenant application called HR app. There are three Azure AD tenants in this example scenario:

- **Adatum** - The tenant used by the company that developed the HR app
- **Contoso** - The tenant used by the Contoso organization, which is a consumer of the HR app
- **Fabrikam** - The tenant used by the Fabrikam organization, which also consumes the HR app



In this example scenario:

Step	Description
1	Is the process of creating the application and service principal objects in the application's home tenant.
2	When Contoso and Fabrikam administrators complete consent, a service principal object is created in their company's Azure AD tenant and assigned the permissions that the administrator granted. Also note that the HR app could be configured/ designed to allow consent by users for individual use.
3	The consumer tenants of the HR application (Contoso and Fabrikam) each have their own service principal object. Each represents their use of an instance of the application at runtime, governed by the permissions consented by the respective administrator.

Permissions and consent in the Azure Active Directory v1.0 and v2.0 endpoints

Azure Active Directory (Azure AD) makes extensive use of permissions for both OAuth and OpenID Connect (OIDC) flows. When your app receives an access token from Azure AD, the access token will include claims that describe the permissions that your app has in respect to a particular resource.

Permissions, also known as scopes, make authorization easy for the resource because the resource only needs to check that the token contains the appropriate permission for whatever API the app is calling.

Types of permissions

Azure AD defines two kinds of permissions:

- **Delegated permissions** - Are used by apps that have a signed-in user present. For these apps, either the user or an administrator consents to the permissions that the app requests and the app is delegated permission to act as the signed-in user when making calls to an API. Depending on the API, the user may not be able to consent to the API directly and would instead require an administrator to provide "admin consent".
- **Application permissions** - Are used by apps that run without a signed-in user present; for example, apps that run as background services or daemons. Application permissions can only be consented by an administrator because they are typically powerful and allow access to data across user-boundaries, or data that would otherwise be restricted to administrators.

Effective permissions are the permissions that your app will have when making requests to an API.

- For delegated permissions, the effective permissions of your app will be the least privileged intersection of the delegated permissions the app has been granted (through consent) and the privileges of the currently signed-in user. Your app can never have more privileges than the signed-in user. Within organizations, the privileges of the signed-in user may be determined by policy or by membership in one or more administrator roles. To learn which administrator roles can consent to delegated permissions, see Administrator role permissions in Azure AD. For example, assume your app has been granted the User.ReadWrite.All delegated permission in Microsoft Graph. This permission nominally grants your app permission to read and update the profile of every user in an organization. If the signed-in user is a global administrator, your app will be able to update the profile of every user in the organization. However, if the signed-in user is not in an administrator role, your app will be able to update only the profile of the signed-in user. It will not be able to update the profiles of other users in the organization because the user that it has permission to act on behalf of does not have those privileges.
- For application permissions, the effective permissions of your app are the full level of privileges implied by the permission. For example, an app that has the User.ReadWrite.All application permission can update the profile of every user in the organization.

Permission attributes

Permissions in Azure AD have a number of properties that help users, administrators, or app developers make informed decisions about what the permission grants access to.

Property name	Description	Example
`ID`	Is a GUID value that uniquely identifies this permission.	570282fd-fa5c-430d-a7fd-fc8d-c98a9dca
`IsEnabled`	Indicates whether this permission is available for use.	true
`Type`	Indicates whether this permission requires user consent or admin consent.	User

Property name	Description	Example
`AdminConsentDescription`	Is a description that's shown to administrators during the admin consent experiences	Allows the app to read email in user mailboxes.
`AdminConsentDisplayName`	Is the friendly name that's shown to administrators during the admin consent experience.	Read user mail
`UserConsentDescription`	Is a description that's shown to users during a user consent experience.	Allows the app to read email in your mailbox.
`UserConsentDisplayName`	Is the friendly name that's shown to users during a user consent experience.	Read your mail
`Value`	Is the string that's used to identify the permission during OAuth 2.0 authorize flows. `Value` may also be combined with the App ID URI string in order to form a fully qualified permission name.	`Mail.Read`

Types of consent

Applications in Azure AD rely on consent in order to gain access to necessary resources or APIs. There are a number of kinds of consent that your app may need to know about in order to be successful. If you are defining permissions, you will also need to understand how your users will gain access to your app or API.

- **Static user consent** - Occurs automatically during the OAuth 2.0 authorize flow when you specify the resource that your app wants to interact with. In the static user consent scenario, your app must have already specified all the permissions it needs in the app's configuration in the Azure portal. If the user (or administrator, as appropriate) has not granted consent for this app, then Azure AD will prompt the user to provide consent at this time.
- **Dynamic user consent** - Is a feature of the v2 Azure AD app model. In this scenario, your app requests a set of permissions that it needs in the OAuth 2.0 authorize flow for v2 apps. If the user has not consented already, they will be prompted to consent at this time. Learn more about dynamic consent.
- **Important:** Dynamic consent can be convenient, but presents a big challenge for permissions that require admin consent, since the admin consent experience doesn't know about those permissions at consent time. If you require admin privileged permissions or if your app uses dynamic consent, you must register all of the permissions in the Azure portal (not just the subset of permissions that require admin consent). This enables tenant admins to consent on behalf of all their users.
- **Admin consent** - Is required when your app needs access to certain high-privilege permissions. Admin consent ensures that administrators have some additional controls before authorizing apps or users to access highly privileged data from the organization.

Best practices

Client best practices

- Only request for permissions that your app needs. Apps with too many permissions are at risk of exposing user data if they are compromised.
- Choose between delegated permissions and application permissions based on the scenario that your app supports.
    - Always use delegated permissions if the call is being made on behalf of a user.
    - Only use application permissions if the app is non-interactive and not making calls on behalf of any specific user. Application permissions are highly privileged and should only be used when absolutely necessary.
- When using an app based on the v2.0 endpoint, always set the static permissions (those specified in your application registration) to be the superset of the dynamic permissions you request at runtime (those specified in code and sent as query parameters in your authorize request) so that scenarios like admin consent works correctly.

Resource/API best practices

- Resources that expose APIs should define permissions that are specific to the data or actions that they are protecting. Following this best practice helps to ensure that clients do not end up with permission to access data that they do not need and that users are well informed about what data they are consenting to.
- Resources should explicitly define `Read` and `ReadWrite` permissions separately.
- Resources should mark any permissions that allow access to data across user boundaries as `Admin` permissions.
- Resources should follow the naming pattern `Subject.Permission[.Modifier]`, where:
    - `Subject` corresponds with the type of data that is available
    - `Permission` corresponds to the action that a user may take upon that data
    - `Modifier` is used optionally to describe specializations of another permission

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.

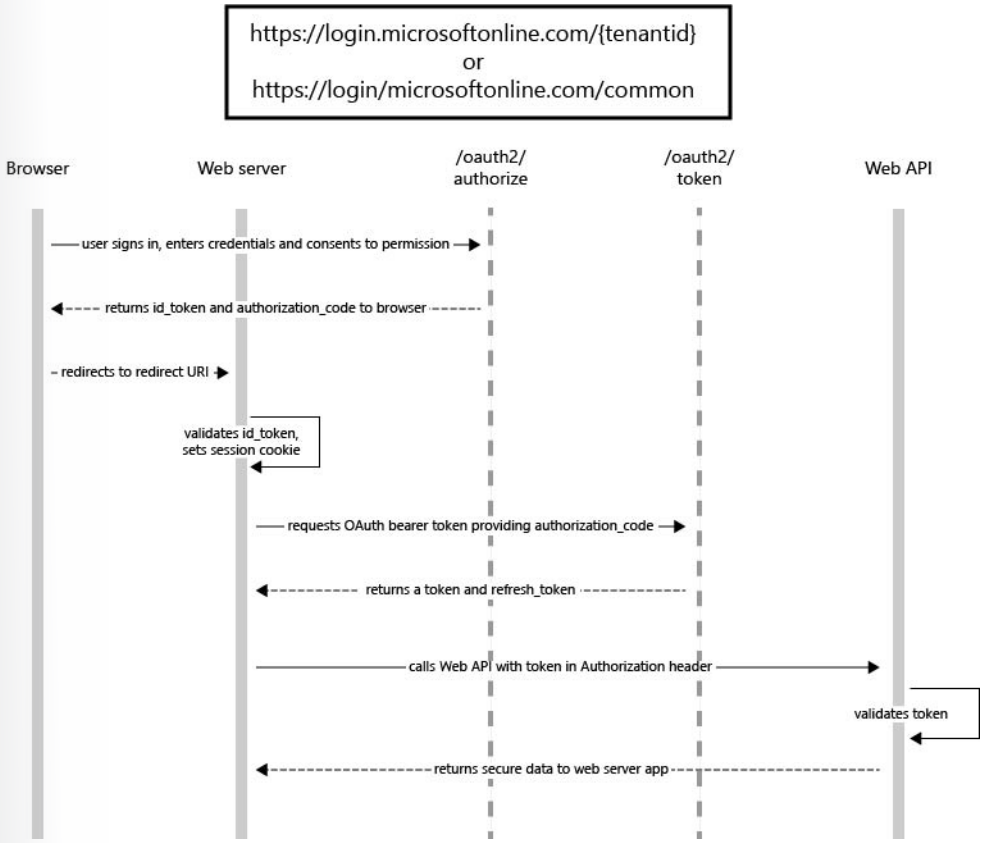
- Sign in to the Azure portal.
- Choose your Azure AD tenant by clicking on your account in the top right corner of the page, followed by clicking on the Switch Directory navigation and then select the appropriate tenant.

    - 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.

Parameter		Description
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.
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.

Error Code	Description	Client Action
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.

Parameter	Description
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 authorization 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.

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

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
```

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`.

Parameter	Need	Description
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.
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.

Parameter	Need	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.
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 plain-text 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.

Parameter	Description
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.
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.

Error Code	Description	Client Action
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.

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`.

Parameter		Description
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.
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.
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.
`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.

Error Code	Description	Client Action
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.
`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).
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.
correlation_id	A unique identifier for the request that can help in diagnostics across components.

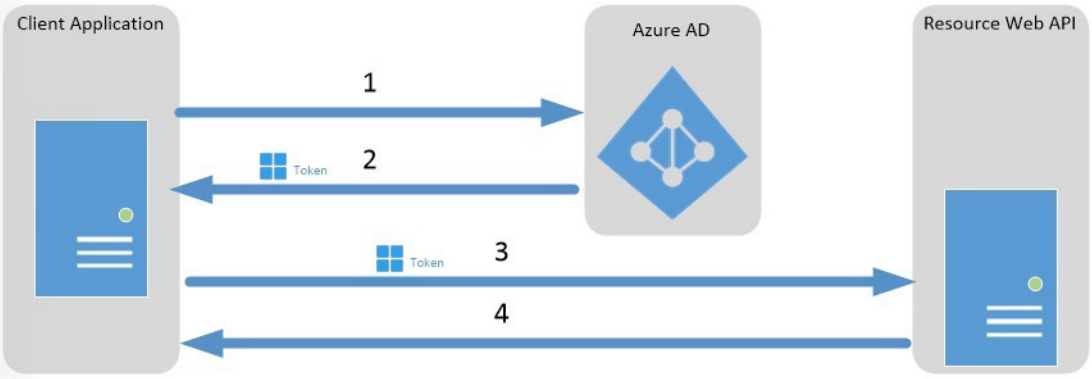
For a description of the error codes and the recommended client action, see **"Error codes for token endpoint errors"** list in the **"Use the authorization code to request an access token"** section above.

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
```

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**.

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

Parameter		Description
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**.

Parameter		Description
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

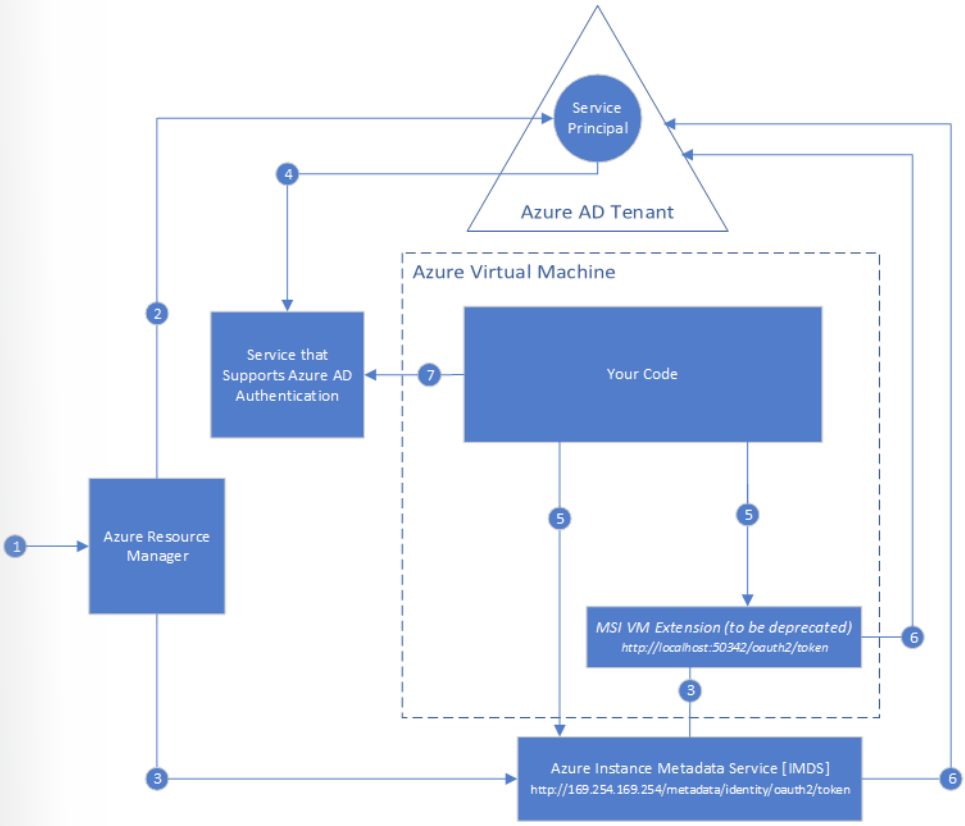
5 https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/services-support-msi
6 https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/known-issues

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.
`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.

Status Code	Error Reason	How To Handle
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.
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)

Status code	Error	Error Description	Solution
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.
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://local-host: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.	

Status code	Error	Error Description	Solution
	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
   ```

Implement authentication by using certificates, forms-based authentication, or tokens

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.

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

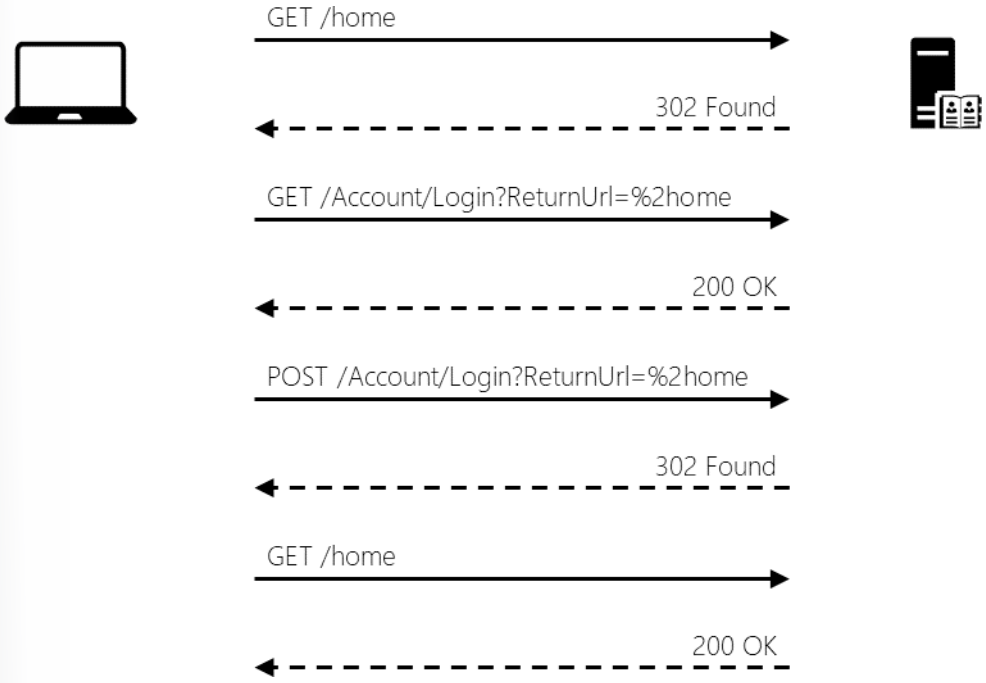
Forms authentication uses an HTML form to send the user's credentials to the server. It is not an internet standard. Forms authentication is appropriate only for web APIs that are called from a web application so that the user can interact with the HTML form. Forms authentication does have a few disadvantages,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

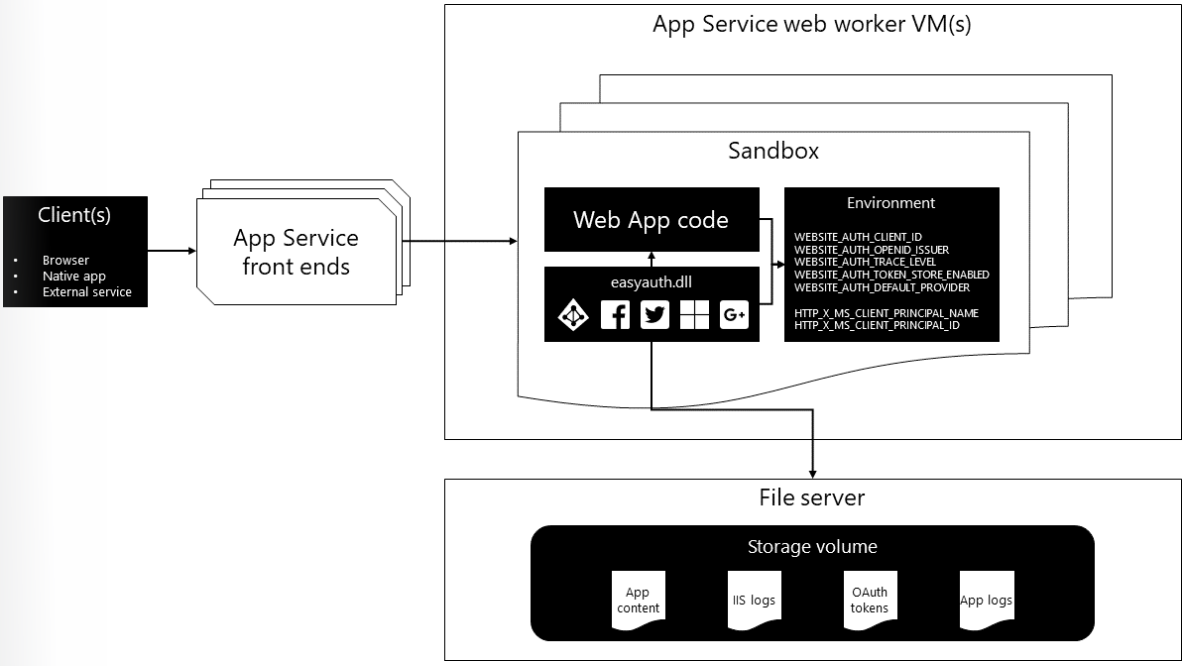
Historically, ASP.NET applications used forms authentication to solve member requirements that were common in the early 2000s. These requirements revolved mostly around authoring login forms and managing a SQL Server database for user names, passwords, and profile data. Today, there is a much broader 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.

Review questions

Module 1 review questions

Microsoft identity platform

Azure Active Directory (Azure AD) supports authentication for a variety of modern app architectures, all of them based on industry-standard protocols OAuth 2.0 or OpenID Connect. What are the two categories of applications that can be developed and integrated with Azure AD?

> Click to see suggested answer

- **Single tenant application** - A single tenant application is intended for use in one organization. These are typically line-of-business (LoB) applications written by an enterprise developer. A single tenant application only needs to be accessed by users in one directory, and as a result, it only needs to be provisioned in one directory. These applications are typically registered by a developer in the organization.
- **Multi-tenant application** - A multi-tenant application is intended for use in many organizations, not just one organization. These are typically software-as-a-service (SaaS) applications written by an independent software vendor (ISV). Multi-tenant applications need to be provisioned in each directory where they will be used, which requires user or administrator consent to register them. This consent process starts when an application has been registered in the directory and is given access to the Graph API or perhaps another web API. When a user or administrator from a different organization signs up to use the application, they are presented with a dialog that displays the permissions the application requires. The user or administrator can then consent to the application, which gives the application access to the stated data, and finally registers the application in their directory.

OAuth2 grant flows

The OAuth2 implicit grant is notorious for being the grant with the longest list of security concerns in the OAuth2 specification. For what type of application scenarios is it actually the recommended approach?

> Click to see suggested answer

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.

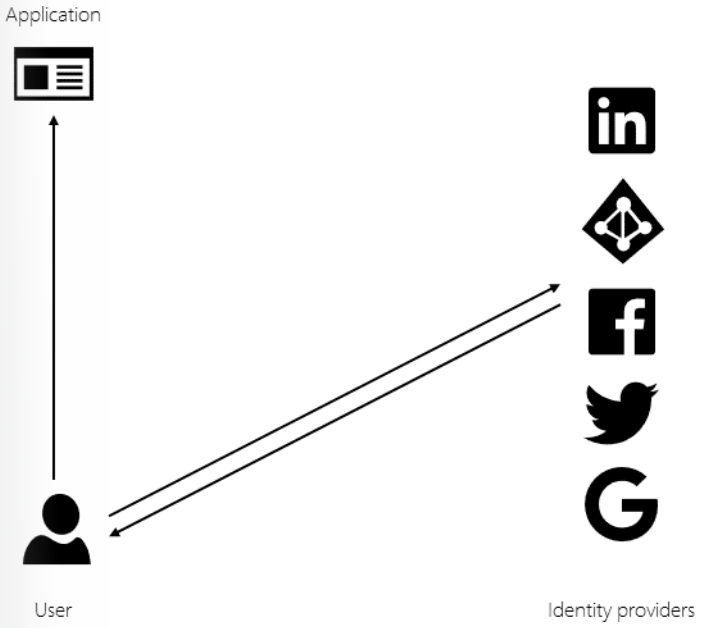
Module 2 Implement access control

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 authorization, 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 access control overview

Access management for cloud resources is a critical function for any organization that is using the cloud. Role-based access control (RBAC) helps you manage who has access to Azure resources, what they can do with those resources, and what areas they have access to.

RBAC is an authorization system built on Azure Resource Manager that provides fine-grained access management of resources in Azure.

What can I do with RBAC?

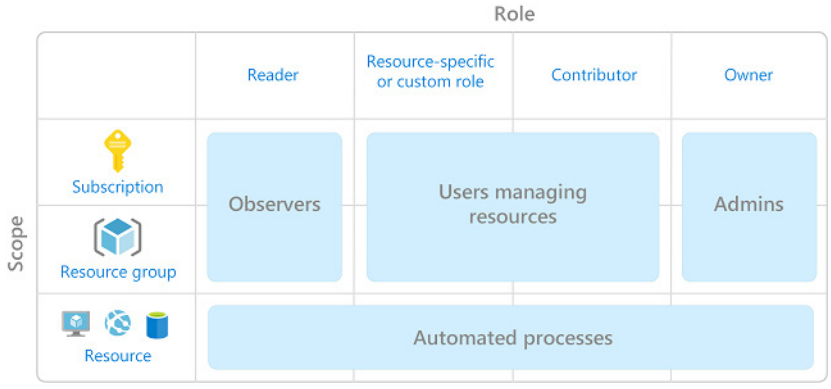
Here are some examples of what you can do with RBAC:

- Allow an application to access all resources in a resource group
- Allow one user to manage virtual machines in a subscription and another user to manage virtual networks
- Allow a DBA group to manage SQL databases in a subscription
- Allow a user to manage all resources in a resource group, such as virtual machines, websites, and subnets

Best practice for using RBAC

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 privilege to get their work done. The following diagram shows a suggested pattern for using RBAC.



How RBAC works

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: security principal, role definition, and scope.

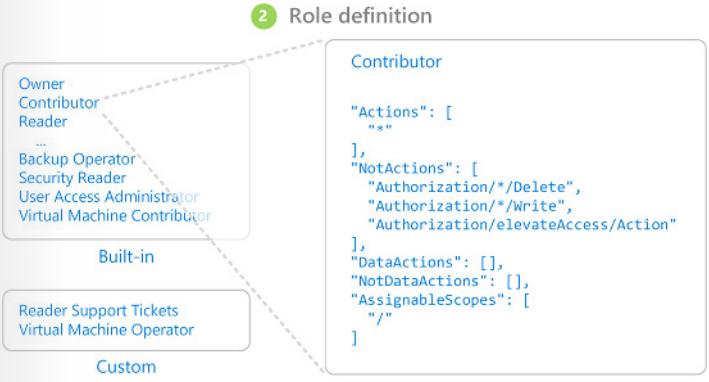
Security principal

A security principal is an object that represents a user, group, service principal, or managed identity that is requesting access to Azure resources.

- User - An individual who has a profile in Azure Active Directory. You can also assign roles to users in other tenants.
- Group - 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.
- Service principal - 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.
- Managed identity - An identity in Azure Active Directory that is automatically managed by Azure. You typically use managed identities when developing cloud applications to manage the credentials for authenticating to Azure services.

Role definition

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.



Azure includes several built-in roles that you can use. The following lists four fundamental built-in roles. The first three apply to all resource types.

- **Owner** - Has full access to all resources including the right to delegate access to others.
- **Contributor** - Can create and manage all types of Azure resources but can't grant access to others.
- **Reader** - Can view existing Azure resources.
- **User Access Administrator** - Lets you manage user access to Azure resources.

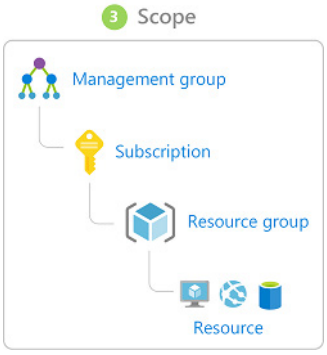
The rest of the built-in roles allow management of specific Azure resources. For example, the **Virtual Machine Contributor** role allows a user to create and manage virtual machines. If the built-in roles don't meet the specific needs of your organization, you can create your own custom roles.

Azure has introduced data operations (currently in preview) that enable you to grant access to data within an object. For example, if a user has read data access to a storage account, then they can read the blobs or messages within that storage account.

Scope

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.



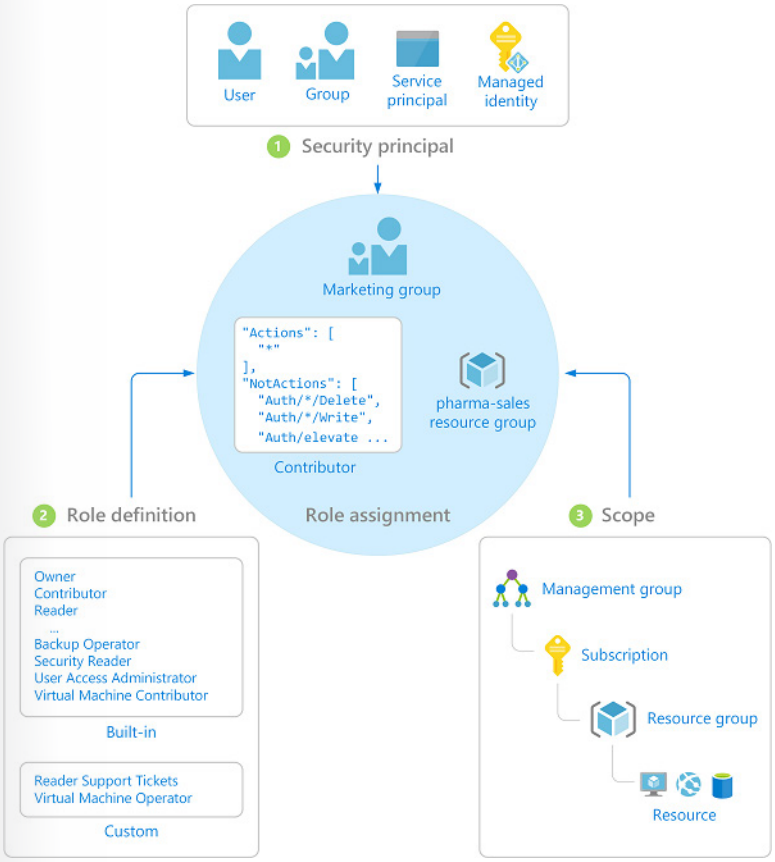
When you grant access at a parent scope, those permissions are inherited to the child scopes. For example:

- If you assign the **Owner** role to a user at the management group scope, that user can manage everything in all subscriptions in the management group.
- If you assign the **Reader** role to a group at the subscription scope, the members of that group can view every resource group and resource in the subscription.
- If you assign the **Contributor** role to an application at the resource group scope, it can manage resources of all types in that resource group, but not other resource groups in the subscription.

Role assignments

A *role assignment* is the process of attaching a role definition to a user, group, service principal, or managed identity 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.



You can create role assignments using the Azure portal, Azure CLI, Azure PowerShell, Azure SDKs, or REST APIs. You can have up to 2000 role assignments in each subscription. To create and remove role assignments, you must have Microsoft.Authorization/roleAssignments/* permission. This permission is granted through the Owner or User Access Administrator roles.

Deny assignments

Previously, RBAC was an allow-only model with no deny, but now RBAC supports deny assignments in a limited way. Similar to a role assignment, a deny assignment attaches a set of deny actions to a user, group, service principal, or managed identity at a particular scope for the purpose of denying access. A role assignment defines a set of actions that are allowed, while a deny assignment defines a set of actions that are not allowed. In other words, deny assignments block users from performing specified actions even if a role assignment grants them access. Deny assignments take precedence over role assignments. Currently, deny assignments are read-only and can only be set by Azure.

How RBAC determines if a user has access to a resource

The following are the high-level steps that RBAC uses to determine if you have access to a resource on the management plane. This is helpful to understand if you are trying to troubleshoot an access issue.

1. A user (or service principal) acquires a token for Azure Resource Manager.
2. The token includes the user's group memberships (including transitive group memberships).
3. The user makes a REST API call to Azure Resource Manager with the token attached.

4. Azure Resource Manager retrieves all the role assignments and deny assignments that apply to the resource upon which the action is being taken.
5. Azure Resource Manager narrows the role assignments that apply to this user or their group and determines what roles the user has for this resource.
6. Azure Resource Manager determines if the action in the API call is included in the roles the user has for this resource.
7. If the user doesn't have a role with the action at the requested scope, access is not granted. Otherwise, Azure Resource Manager checks if a deny assignment applies.
8. If a deny assignment applies, access is blocked. Otherwise access is granted.

Classic subscription administrator roles, Azure RBAC roles, and Azure AD administrator roles

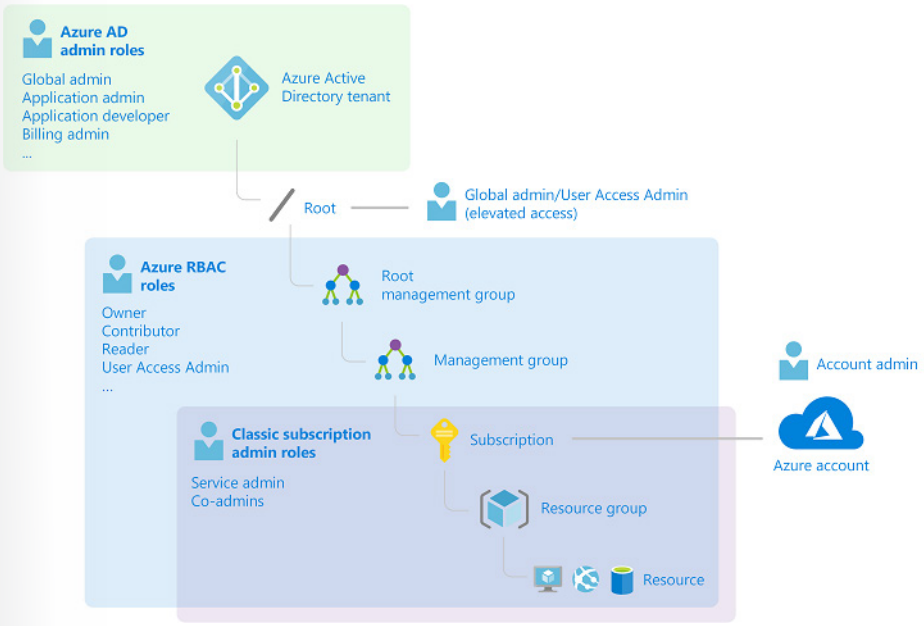
If you are new to Azure, you may find it a little challenging to understand all the different roles in Azure. This article helps explain the following roles and when you would use each:

- Classic subscription administrator roles
- Azure role-based access control (RBAC) roles
- Azure Active Directory (Azure AD) administrator roles

How the roles are related

To better understand roles in Azure, it helps to know some of the history. When Azure was initially released, access to resources was managed with just three administrator roles: Account Administrator, Service Administrator, and Co-Administrator. Later, role-based access control (RBAC) for Azure resources was added. Azure RBAC is a newer authorization system that provides fine-grained access management to Azure resources. RBAC includes many built-in roles, can be assigned at different scopes, and allows you to create your own custom roles. To manage resources in Azure AD, such as users, groups, and domains, there are several Azure AD administrator roles.

The following diagram is a high-level view of how the classic subscription administrator roles, Azure RBAC roles, and Azure AD administrator roles are related.



Classic subscription administrator roles

Account Administrator, Service Administrator, and Co-Administrator are the three classic subscription administrator roles in Azure. Classic subscription administrators have full access to the Azure subscription. They can manage resources using the Azure portal, Azure Resource Manager APIs, and the classic deployment model APIs. The account that is used to sign up for Azure is automatically set as both the Account Administrator and Service Administrator. Then, additional Co-Administrators can be added. The Service Administrator and the Co-Administrators have the equivalent access of users who have been assigned the Owner role (an Azure RBAC role) at the subscription scope. The following table describes the differences between these three classic subscription administrative roles.

Classic subscription administrator	Limit	Permissions	Notes
Account Administrator	1 per Azure account	• Access the Azure Account Center • Manage all subscriptions in an account • Create new subscriptions • Cancel subscriptions • Change the billing for a subscription • Change the Service Administrator	Conceptually, the billing owner of the subscription. The Account Administrator has no access to the Azure portal.
Service Administrator	1 per Azure subscription	• Manage services in the Azure portal • Assign users to the Co-Administrator role	By default, for a new subscription, the Account Administrator is also the Service Administrator. The Service Administrator has the equivalent access of a user who is assigned the Owner role at the subscription scope. The Service Administrator has full access to the Azure portal.

Classic subscription administrator	Limit	Permissions	Notes
Co-Administrator	200 per subscription	• Same access privileges as the Service Administrator, but can't change the association of subscriptions to Azure directories • Assign users to the Co-Administrator role, but cannot change the Service Administrator	The Co-Administrator has the equivalent access of a user who is assigned the Owner role at the subscription scope.

Azure RBAC roles

Azure RBAC is an authorization system built on Azure Resource Manager that provides fine-grained access management to Azure resources, such as compute and storage. Azure RBAC includes over 70 built-in roles. There are four fundamental RBAC roles. The first three apply to all resource types:

Azure RBAC role	Permissions	Notes
Owner	• Full access to all resources • Delegate access to others	The Service Administrator and Co-Administrators are assigned the Owner role at the subscription scope Applies to all resource types.
Contributor	• Create and manage all of types of Azure resources • Cannot grant access to others	Applies to all resource types.
Reader	• View Azure resources	Applies to all resource types.
User Access Administrator	• Manage user access to Azure resources	

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

Differences between Azure RBAC roles and Azure AD administrator roles

At a high level, Azure RBAC roles control permissions to manage Azure resources, while Azure AD administrator roles control permissions to manage Azure Active Directory resources. The following table compares some of the differences.

Azure RBAC roles	Azure AD administrator roles
Manage access to Azure resources	Manage access to Azure Active Directory resources
Supports custom roles	Cannot create your own roles
Scope can be specified at multiple levels (management group, subscription, resource group, resource)	Scope is at the tenant level
Role information can be accessed in Azure portal, Azure CLI, Azure PowerShell, Azure Resource Manager templates, REST API	Role information can be accessed in Azure admin portal, Office 365 admin portal, Microsoft Graph, AzureAD PowerShell

Manage access using RBAC and the REST API

Role-based access control (RBAC) is the way that you manage access to resources in Azure. This lesson describes how you manage access for users, groups, and applications using RBAC and the REST API.

List access

In RBAC, to list access, you list the role assignments. To list role assignments, use one of the **Role Assignments - List**[1] REST APIs. To refine your results, you specify a scope and an optional filter. To call the API, you must have access to the `Microsoft.Authorization/roleAssignments/read` operation at the specified scope. Several **built-in roles**[2] are granted access to this operation.

1. Start with the following request:

```
GET https://management.azure.com/{scope}/providers/Microsoft.Authorization/
roleAssignments?api-version=2015-07-01&$filter={filter}
```

2. Within the URI, replace {scope} with the scope for which you want to list the role assignments.

Scope	Type
`subscriptions/{subscriptionId}`	Subscription
`subscriptions/{subscriptionId}/resourceGroups/myresourcegroup1`	Resource group
`subscriptions/{subscriptionId}/resourceGroups/myresourcegroup1/ providers/Microsoft.Web/sites/mysite1`	Resource

3. Replace *{filter}* with the condition that you want to apply to filter the role assignment list.

1 https://docs.microsoft.com/en-us/rest/api/authorization/roleassignments/list
2 https://docs.microsoft.com/en-us/azure/role-based-access-control/built-in-roles

Filter	Description
`$filter=atScope()`	List role assignments for only the specified scope, not including the role assignments at subscopes.
`$filter=principalId%20eq%20'{objectId}'`	List role assignments for a specified user, group, or service principal.
$filter=assignedTo('{objectId}')	List role assignments for a specified user, including ones inherited from groups.

Grant access

In RBAC, to grant access, you create a role assignment. To create a role assignment, use the **Role Assignments - Create**[3] REST API and specify the security principal, role definition, and scope. To call this API, you must have access to the `Microsoft.Authorization/roleAssignments/write` operation. Of the built-in roles, only Owner and User Access Administrator are granted access to this operation.

1. Use the [Role Definitions - List(https://docs.microsoft.com/en-us/rest/api/authorization/roledefinitions/list)] REST API or see **Built-in**[4] roles to get the identifier for the role definition you want to assign.
2. Use a GUID tool to generate a unique identifier that will be used for the role assignment identifier. The identifier has the format:
3. `00000000-0000-0000-0000-000000000000`
4. Start with the following request and body:

```
PUT https://management.azure.com/{scope}/providers/Microsoft.Authorization/
roleAssignments/{roleAssignmentName}?api-version=2015-07-01


{
"properties": {
    "roleDefinitionId": "/subscriptions/{subscriptionId}/providers/Micro-
soft.Authorization/roleDefinitions/{roleDefinitionId}",
    "principalId": "{principalId}"
}
}
```

5. Within the URI, replace {scope} with the scope for the role assignment.

Scope	Type
`subscriptions/{subscriptionId}`	Subscription
`subscriptions/{subscriptionId}/resourceGroups/myresourcegroup1`	Resource group
`subscriptions/{subscriptionId}/resourceGroups/myresourcegroup1/ providers/Microsoft.Web/sites/mysite1`	Resource

6. Replace `{roleAssignmentName}` with the GUID identifier of the role assignment.
7. Within the request body, replace `{subscriptionId}` with your subscription identifier.

3 https://docs.microsoft.com/en-us/rest/api/authorization/roleassignments/create
4 https://docs.microsoft.com/en-us/azure/role-based-access-control/built-in-roles

8. Replace `{roleDefinitionId}` with the role definition identifier.
9. Replace `{principalId}` with the object identifier of the user, group, or service principal that will be assigned the role.

Remove access

In RBAC, to remove access, you remove a role assignment. To remove a role assignment, use the **Role Assignments - Delete**[5] REST API. To call this API, you must have access to the `Microsoft.Authorization/roleAssignments/delete` operation. Of the built-in roles, only Owner and User Access Administrator are granted access to this operation.

1. Get the role assignment identifier (GUID). This identifier is returned when you first create the role assignment or you can get it by listing the role assignments.
2. Start with the following request:

```
DELETE https://management.azure.com/{scope}/providers/Microsoft.Authorization/roleAssignments/{roleAssignmentName}?api-version=2015-07-01
```

1. Within the URI, replace {scope} with the scope for removing the role assignment.

Scope	Type
`subscriptions/{subscriptionId}`	Subscription
`subscriptions/{subscriptionId}/resourceGroups/myresourcegroup1`	Resource group
`subscriptions/{subscriptionId}/resourceGroups/myresourcegroup1/ providers/Microsoft.Web/sites/mysite1`	Resource

2. Replace `{roleAssignmentName}` with the GUID identifier of the role assignment.

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
{
```

5 https://docs.microsoft.com/en-us/rest/api/authorization/roleassignments/delete

```
}
```

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`.

Review questions

Module 2 review questions

Authentication vs. authorization

The terms authentication and authorization can be confusing, can you describe what role each plays?

> Click to see suggested answer

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.

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. Can you come up with some code snippets that illustrate how to do that?

> Click to see suggested answer

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
{
}
```

Module 3 Implement secure data solutions

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

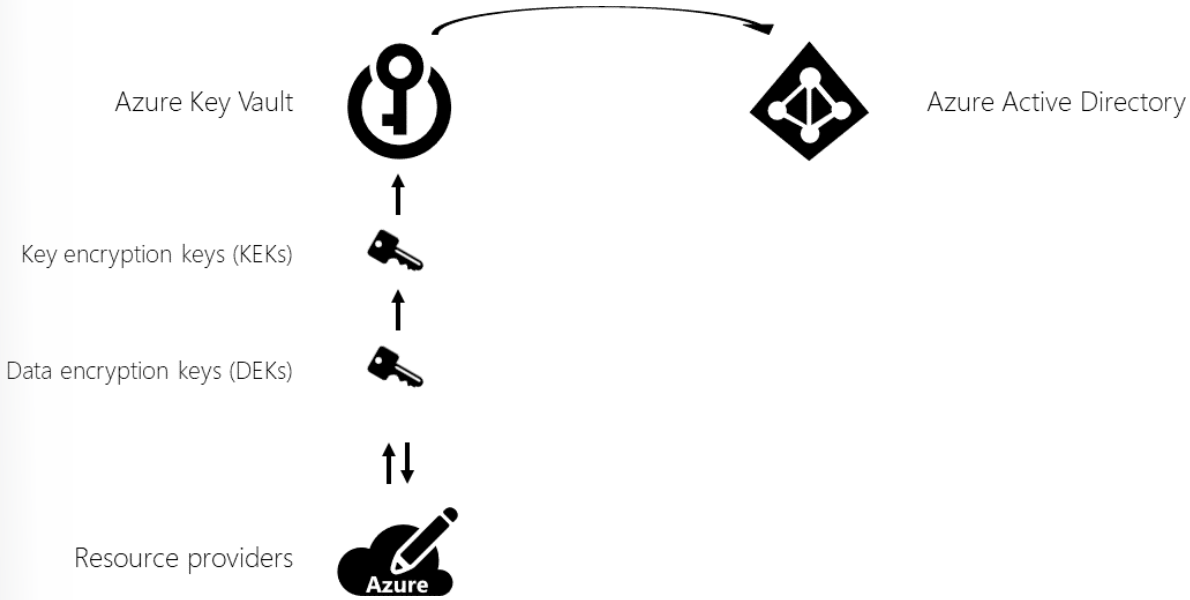
reason, encryption at rest is highly recommended and is a high-priority requirement for many organizations.

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.

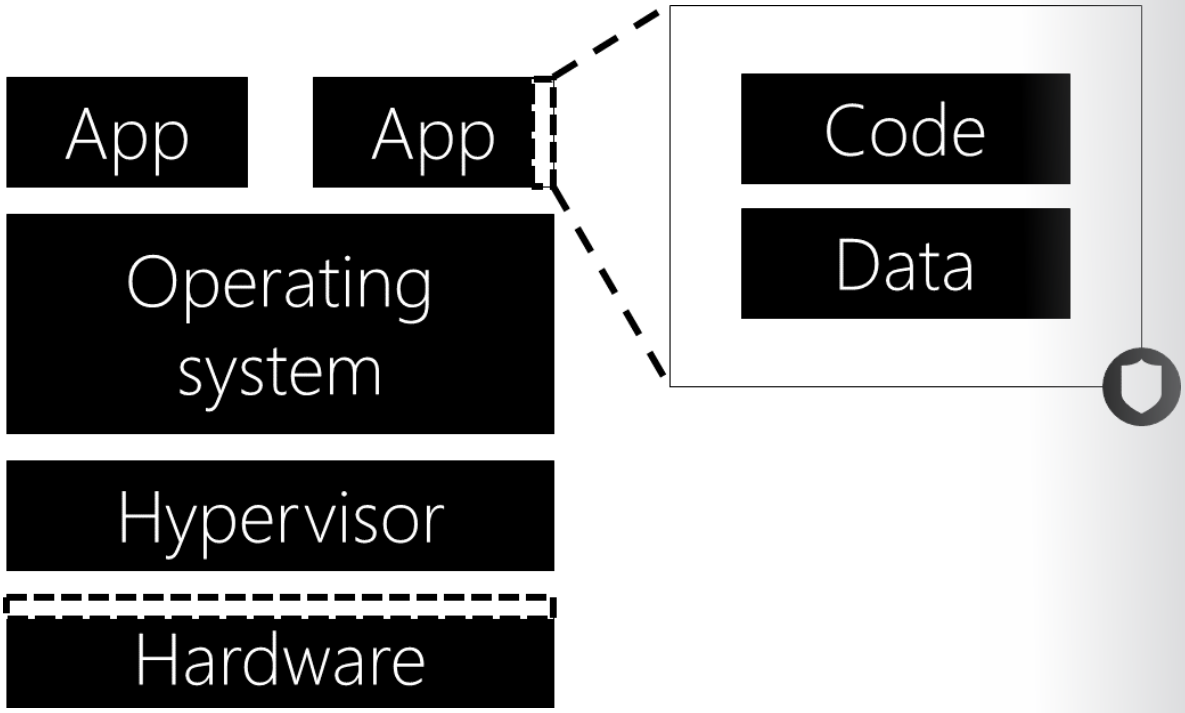
Note: Always Encrypted requires a specialized driver installed on the client computer to automatically encrypt and decrypt sensitive data in the client application. For many applications, this does require some code changes. This is in contrast to TDE, which only requires a change to the application's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

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.

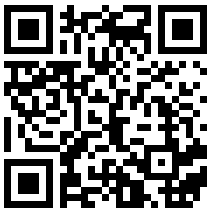
To add a secret to the vault, you just need to take a couple of additional steps. This password could be used by an application. The password will be called DatabasePassword and will store the value of Pa5w.rd in it:

```
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
```

Accessing Key Vault using the portal



Review questions

Module 3 review questions

Encryption at rest

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. What type of encryption does Azure storage use and what are some of the design concepts?

> Click to see suggested answer

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 confidential computing

What does confidential computing help to ensure when protecting data?

> Click to see suggested answer

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.