# Microsoft identity platform

## Microsoft identity platform

The Microsoft identity platform helps you build applications your users and customers can sign in to using their Microsoft identities or social accounts, and provide authorized access to your own APIs or Microsoft APIs like Microsoft Graph.

There are several components that make up the Microsoft identity platform:

OAuth 2.0 and OpenID Connect standard-compliant authentication service enabling developers to authenticate several identity types, including:

* Work or school accounts, provisioned through Azure Active Directory
* Personal Microsoft account, like Skype, Xbox, and Outlook.com
* Social or local accounts, by using Azure Active Directory B2C
* Open-source libraries: Microsoft Authentication Libraries (MSAL) and support for other standards-compliant libraries

Application management portal: A registration and configuration experience in the Azure portal, along with the other Azure management capabilities.

Application configuration API and PowerShell: Programmatic configuration of your applications through the Microsoft Graph API and PowerShell so you can automate your DevOps tasks.

## Service principals

To delegate Identity and Access Management functions to Azure Active Directory, an application must be registered with an Azure Active Directory tenant.

When you register an app in the Azure portal, you choose whether it is:

* Single tenant: only accessible in your tenant
* Multi-tenant: accessible in other tenants Availability zones

If you register an application in the portal, an application object (the globally unique instance of the app) as well as a service principal object are automatically created in your home tenant. You also have a globally unique ID for your app (the app or client ID).

### Application object

An Azure Active Directory application is defined by its one and only application object, which resides in the Azure Active Directory tenant where the application was registered (known as the application's "home" tenant). An application object is used as a template or blueprint to create one or more service principal objects. A service principal is created in every tenant where the application is used. Similar to a class in object-oriented programming, the application object has some static properties that are applied to all the created service principals (or application instances).

The application object describes three aspects of an application: how the service can issue tokens in order to access the application, resources that the application might need to access, and the actions that the application can take.

The Microsoft Graph Application entity defines the schema for an application object's properties.

### Service principal object

To access resources that are secured by an Azure Active Directory tenant, the entity that requires access must be represented by a security principal. This is true for both users (user principal) and applications (service principal).

The security principal defines the access policy and permissions for the user/application in the Azure Active Directory tenant. This enables core features such as authentication of the user/application during sign-in, and authorization during resource access.

There are three types of service principal:

* **Application** - This type of service principal is the local representation, or application instance, of a global application object in a single tenant or directory. A service principal is created in each tenant where the application is used and references the globally unique app object. The service principal object defines what the app can actually do in the specific tenant, who can access the app, and what resources the app can access.
* **Managed identity** - This type of service principal is used to represent a [managed identity](https://learn.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/overview). Managed identities provide an identity for applications to use when connecting to resources that support Azure Active Directory authentication. When a managed identity is enabled, a service principal representing that managed identity is created in your tenant. Service principals representing managed identities can be granted access and permissions, but cannot be updated or modified directly.
* **Legacy** - This type of service principal represents a legacy app, which is an app created before app registrations were introduced or an app created through legacy experiences. A legacy service principal can have credentials, service principal names, reply URLs, and other properties that an authorized user can edit, but does not have an associated app registration. The service principal can only be used in the tenant where it was created.

## Relationship between application objects and service principals

The application object is the *global* representation of your application for use across all tenants, and the service principal is the *local* representation for use in a specific tenant. The application object serves as the template from which common and default properties are *derived* for use in creating corresponding service principal objects.

An application object has:

* A 1:1 relationship with the software application, and
* A 1:many relationship with its corresponding service principal object(s).

## Permissions and consent

The Microsoft identity platform implements the [OAuth 2.0](https://learn.microsoft.com/en-us/azure/active-directory/develop/active-directory-v2-protocols) authorization protocol. OAuth 2.0 is a method through which a third-party app can access web-hosted resources on behalf of a user. Any web-hosted resource that integrates with the Microsoft identity platform has a resource identifier, or *application ID URI*.

Here are some examples of Microsoft web-hosted resources:

* Microsoft Graph: https://graph.microsoft.com
* Microsoft 365 Mail API: https://outlook.office.com
* Azure Key Vault: <https://vault.azure.net>

## Permission types

The Microsoft identity platform supports two types of permissions: delegated permissions and app-only access.

1. 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. The app is delegated with the permission to act as a signed-in user when it makes calls to the target resource.

2. App-only access permissions are used by apps that run without a signed-in user present, for example, apps that run as background services or daemons. Only an administrator can consent to app-only access permissions.

## Consent types

Applications in Microsoft identity platform 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.

There are three consent types: *static user consent*, *incremental and dynamic user consent*, and *admin consent*.

### 1. Static user consent

In the static user consent scenario, you must specify 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 Microsoft identity platform will prompt the user to provide consent at this time. Static permissions also enables administrators to consent on behalf of all users in the organization.

While static permissions of the app defined in the Azure portal keep the code nice and simple, it presents some possible issues for developers:

* The app needs to request all the permissions it would ever need upon the user's first sign-in. This can lead to a long list of permissions that discourages end users from approving the app's access on initial sign-in.
* The app needs to know all of the resources it would ever access ahead of time. It is difficult to create apps that could access an arbitrary number of resources.

### 2. Incremental and dynamic user consent

With the Microsoft identity platform endpoint, you can ignore the static permissions defined in the app registration information in the Azure portal and request permissions incrementally instead. You can ask for a minimum set of permissions upfront and request more over time as the customer uses additional app features.

To do so, you can specify the scopes your app needs at any time by including the new scopes in the scope parameter when requesting an access token - without the need to pre-define them in the application registration information. If the user hasn't yet consented to new scopes added to the request, they'll be prompted to consent only to the new permissions. Incremental, or dynamic consent, only applies to delegated permissions and not to app-only access permissions.

**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

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.

Admin consent done on behalf of an organization still requires the static permissions registered for the app. Set those permissions for apps in the app registration portal if you need an admin to give consent on behalf of the entire organization. This reduces the cycles required by the organization admin to set up the application.

## Requesting individual user consent

In an OpenID Connect or OAuth 2.0 authorization request, an app can request the permissions it needs by using the scope query parameter. For example, when a user signs into an app, the app sends a request like the following example. Line breaks are added for legibility.

GET https://login.microsoftonline.com/common/oauth2/v2.0/authorize?

client\_id=6731de76-14a6-49ae-97bc-6eba6914391e

&response\_type=code

&redirect\_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F

&response\_mode=query

&scope=

https%3A%2F%2Fgraph.microsoft.com%2Fcalendars.read%20

https%3A%2F%2Fgraph.microsoft.com%2Fmail.send

&state=12345

The scope parameter is a space-separated list of delegated permissions that the app is requesting. Each permission is indicated by appending the permission value to the resource's identifier (the application ID URI). In the request example, the app needs permission to read the user's calendar and send mail as the user.

After the user enters their credentials, the Microsoft identity platform checks for a matching record of *user consent*. If the user hasn't consented to any of the requested permissions in the past, and if the administrator hasn't consented to these permissions on behalf of the entire organization, the Microsoft identity platform asks the user to grant the requested permissions.

## Conditional access

The Conditional Access feature in Azure Active Directory offers one of several ways that you can use to secure your app and protect a service. Conditional Access enables developers and enterprise customers to protect services in a multitude of ways including:

* [Multifactor authentication](https://learn.microsoft.com/en-us/azure/active-directory/authentication/concept-mfa-howitworks)
* Allowing only Intune enrolled devices to access specific services
* Restricting user locations and IP ranges

**How does Conditional Access impact an app?**

In most common cases, Conditional Access does not change an app's behavior or require any changes from the developer. Only in certain cases when an app indirectly or silently requests a token for a service does an app require code changes to handle Conditional Access challenges. It may be as simple as performing an interactive sign-in request.

Specifically, the following scenarios require code to handle Conditional Access challenges:

* Apps performing the on-behalf-of flow
* Apps accessing multiple services/resources
* Single-page apps using MSAL.js
* Web apps calling a resource

For your app to continue functioning when a new policy is applied, implement challenge handling.

### Conditional Access examples

Some scenarios require code changes to handle Conditional Access whereas others work as is. Here are a few scenarios using Conditional Access to do multifactor authentication that gives some insight into the difference.

* You are building a single-tenant iOS app and apply a Conditional Access policy. The app signs in a user and doesn't request access to an API. When the user signs in, the policy is automatically invoked and the user needs to perform multifactor authentication.
* You are building an app that uses a middle tier service to access a downstream API. An enterprise customer at the company using this app applies a policy to the downstream API. When an end user signs in, the app requests access to the middle tier and sends the token. The middle tier performs on-behalf-of flow to request access to the downstream API. At this point, a claims "challenge" is presented to the middle tier. The middle tier sends the challenge back to the app, which needs to comply with the Conditional Access policy.

# Microsoft Authentication Library

## Microsoft Authentication Library

The Microsoft Authentication Library (MSAL) can be used to provide secure access to Microsoft Graph, other Microsoft APIs, third-party web APIs, or your own web API. MSAL supports many different application architectures and platforms including .NET, JavaScript, Java, Python, Android, and iOS.

MSAL gives you many ways to get tokens, with a consistent API for a number of platforms. Using MSAL provides the following benefits:

* No need to directly use the OAuth libraries or code against the protocol in your application.
* Acquires tokens on behalf of a user or on behalf of an application (when applicable to the platform).
* Maintains a token cache and refreshes tokens for you when they are close to expire. You don't need to handle token expiration on your own.
* Helps you specify which audience you want your application to sign in.
* Helps you set up your application from configuration files.
* Helps you troubleshoot your app by exposing actionable exceptions, logging, and telemetry.

### Application types

Using MSAL, a token can be acquired from a number of application types:

* web applications
* web APIs
* single-page apps (JavaScript)
* mobile and native applications
* daemons and server-side applications

### Authentication flows

Below are some of the different authentication flows provided by Microsoft Authentication Library (MSAL). These flows can be used in a variety of different application scenarios.

| **Flow** | **Description** |
| --- | --- |
| Authorization code | Native and web apps securely obtain tokens in the name of the user |
| Client credentials | Service applications run without user interaction |
| On-behalf-of | The application calls a service/web API, which in turns calls Microsoft Graph |
| Implicit | Used in browser-based applications |
| Device code | Enables sign-in to a device by using another device that has a browser |
| Integrated Windows | Windows computers silently acquire an access token when they are domain joined |
| Interactive | Mobile and desktops applications call Microsoft Graph in the name of a user |
| Username/password | The application signs in a user by using their username and password |

## Public client and Confidential client applications

Security tokens can be acquired by multiple types of applications. These applications tend to be separated into the following two categories. Each is used with different libraries and objects.

### 1. Public client applications:

Are apps that run on devices or desktop computers or in a web browser. They're not trusted to safely keep application secrets, so they only access web APIs on behalf of the user. (They support only public client flows.) Public clients can't hold configuration-time secrets, so they don't have client secrets.

### 2. Confidential client applications:

Are apps that run on servers (web apps, web API apps, or even service/daemon apps). They're considered difficult to access, and for that reason capable of keeping an application secret. Confidential clients can hold configuration-time secrets. Each instance of the client has a distinct configuration (including client ID and client secret).

## Initialize client applications

With MSAL.NET 3.x, the recommended way to instantiate an application is by using the application builders: PublicClientApplicationBuilder and ConfidentialClientApplicationBuilder. They offer a powerful mechanism to configure the application either from the code, or from a configuration file, or even by mixing both approaches.

Before initializing an application, you first need to register it so that your app can be integrated with the Microsoft identity platform.

### Initializing public and confidential client applications from code

The following code instantiates a public client application, signing-in users in the Microsoft Azure public cloud, with their work and school accounts, or their personal Microsoft accounts.

IPublicClientApplication app = PublicClientApplicationBuilder.Create(clientId).Build();

In the same way, the following code instantiates a confidential application (a Web app located at https://myapp.azurewebsites.net) handling tokens from users in the Microsoft Azure public cloud, with their work and school accounts, or their personal Microsoft accounts. The application is identified with the identity provider by sharing a client secret:

string redirectUri = "https://myapp.azurewebsites.net";

IConfidentialClientApplication app = ConfidentialClientApplicationBuilder.Create(clientId)

.WithClientSecret(clientSecret)

.WithRedirectUri(redirectUri )

.Build();

## Builder modifiers

In the code snippets using application builders, a number of .With methods can be applied as modifiers (for example, .WithAuthority and .WithRedirectUri).

### Modifiers common to public and confidential client applications

The table below lists some of the modifiers you can set on a public, or confidential client.

| **Modifier** | **Description** |
| --- | --- |
| .WithAuthority() | Sets the application default authority to an Azure Active Directory authority, with the possibility of choosing the Azure Cloud, the audience, the tenant (tenant ID or domain name), or providing directly the authority URI. |
| .WithTenantId(string tenantId) | Overrides the tenant ID, or the tenant description. |
| .WithClientId(string) | Overrides the client ID. |
| .WithRedirectUri(string redirectUri) | Overrides the default redirect URI. In the case of public client applications, this will be useful for scenarios requiring a broker. |
| .WithComponent(string) | Sets the name of the library using MSAL.NET (for telemetry reasons). |
| .WithDebugLoggingCallback() | If called, the application will call Debug.Write simply enabling debugging traces. |
| .WithLogging() | If called, the application will call a callback with debugging traces. |
| .WithTelemetry(TelemetryCallback telemetryCallback) | Sets the delegate used to send telemetry. |

### Modifiers specific to confidential client applications

The modifiers you can set on a confidential client application builder are:

| **Modifier** | **Description** |
| --- | --- |
| .WithCertificate(X509Certificate2 certificate) | Sets the certificate identifying the application with Azure Active Directory. |
| .WithClientSecret(string clientSecret) | Sets the client secret (app password) identifying the application with Azure Active Directory. |

## Implement interactive authentication by using MSAL.NET

Perform the following actions:

* Register an application with the Microsoft identity platform
* Use the PublicClientApplicationBuilder class in MSAL.NET
* Acquire a token interactively in a console application

## Register a new application

Sign in to the portal: [https://portal.azure.com](https://portal.azure.com/) -> Azure Active Directory

-> Manage -> App registrations > New registration ->

| **Field** | **Value** |
| --- | --- |
| **Name** | az204appreg |
| **Supported account types** | Select **Accounts in this organizational directory only** |
| **Redirect URI (optional)** | Select **Public client/native (mobile & desktop)** and enter http://localhost in the box to the right. |

-> Register.

Azure Active Directory assigns a unique application (client) ID to your app, and you're taken to your application's **Overview** page.

## Set up the console application

1. Launch Visual Studio Code and open a terminal by selecting **Terminal** and then **New Terminal**.
2. Create a folder for the project and change into the folder.

md az204-auth

cd az204-auth

1. Create the .NET console app.

dotnet new console

1. Open the *az204-auth* folder in Visual Studio Code.

code . -r

## Build the console app

### 1. Add packages and using statements

**Microsoft.Identity.Client**

### 2. Add code for the interactive authentication

### 3. Acquire a token

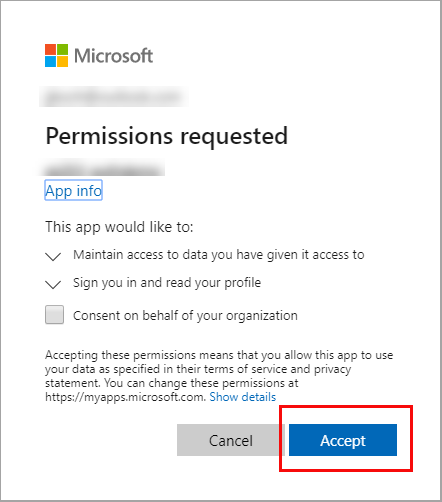
## Run the application

dotnet build

dotnet run

The app will open the default browser prompting you to select the account you want to authenticate with. If there are multiple accounts listed select the one associated with the tenant used in the app.

If this is the first time you've authenticated to the registered app you will receive a **Permissions requested** notification asking you to approve the app to read data associated with your account. Select **Accept**.



You should see the results similar to the example below in the console.

Token: eyJ0eXAiOiJKV1QiLCJub25jZSI6IlVhU.....