

Contents

[Microsoft identity platform \(Azure Active Directory for developers\)](#)

[Evolution of Microsoft identity platform](#)

[v2.0](#)

[Overview](#)

[Quickstarts](#)

[Set up a tenant](#)

[Configure an application](#)

[Register an app](#)

[Configure app to access web APIs](#)

[Configure app to expose web APIs](#)

[Modify accounts supported by an app](#)

[Remove an app](#)

[Single-page apps](#)

[JavaScript](#)

[Web apps](#)

[ASP .NET](#)

[ASP .NET Core](#)

[NodeJS](#)

[Java](#)

[Python](#)

[Web APIs](#)

[ASP .NET](#)

[ASP .NET Core](#)

[Mobile and desktop apps](#)

[Android](#)

[iOS and macOS](#)

[Universal Windows Platform](#)

[Windows Desktop .NET](#)

[Daemon apps](#)

[.NET Core console \(daemon\)](#)

[Python console daemon](#)

[ASP.NET daemon web app](#)

Tutorials

[Single-page apps](#)

[JavaScript](#)

[Web apps](#)

[ASP .NET](#)

[Mobile and desktop apps](#)

[Android](#)

[iOS and macOS](#)

[Universal Windows Platform](#)

[Windows Desktop .NET](#)

Samples

Concepts

[Authentication basics](#)

[Authentication flows and app scenarios](#)

[Identity platform best practices](#)

[Application scenarios](#)

[Sign-in audience](#)

[Single-page app](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Sign in and sign out](#)

[Acquire token for an API](#)

[Call a web API](#)

[Move to production](#)

[Web app that signs in users](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Sign-in and sign-out](#)

[Move to production](#)

[Web app that calls web APIs](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Sign-in](#)

[Acquire token for the app](#)

[Call a web API](#)

[Move to production](#)

[Protected web API](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Verification of scopes or app roles](#)

[Move to production](#)

[Web API that calls web APIs](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Acquire a token](#)

[Call a web API](#)

[Move to production](#)

[Desktop app that calls web APIs](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Acquire token](#)

[Call a web API](#)

[Move to production](#)

[Daemon app](#)

[Overview](#)

- [App registration](#)
- [Code configuration](#)
- [Acquire token](#)
- [Call a web API](#)
- [Move to production](#)
- [Mobile app that calls web APIs](#)
 - [Overview](#)
 - [App registration](#)
 - [Code configuration](#)
 - [Mobile platforms specific config](#)
 - [Xamarin Android](#)
 - [System browser on Android](#)
 - [Xamarin iOS](#)
 - [Use brokers with Xamarin iOS and Android Applications](#)
 - [Universal Windows Platform](#)
 - [Acquiring a token for the app](#)
 - [Calling a web API](#)
 - [Move to production](#)
- [Microsoft Authentication Library \(MSAL\)](#)
 - [Overview](#)
 - [Accounts & tenant profiles \(Android\)](#)
 - [Authorization agents \(Android\)](#)
 - [Brokered auth \(Android\)](#)
 - [Migration](#)
 - [Migrate to MSAL.NET](#)
 - [Migrate to MSAL.js](#)
 - [Migrate to MSAL.Android](#)
 - [Migrate to MSAL.iOS and MacOS](#)
 - [Migrate to MSAL Java](#)
 - [Migrate Xamarin apps using brokers from ADAL.NET to MSAL.NET](#)
 - [Supported authentication flows](#)
 - [Single and multi account public client apps \(Android\)](#)

[Acquire and cache tokens](#)

[Acquire and cache tokens](#)

[Scopes for v1.0 apps](#)

[Client applications](#)

[Client applications](#)

[Application configuration \(.NET\)](#)

[Initialize applications \(.NET\)](#)

[Client assertions \(.NET\)](#)

[Initialize applications \(JS\)](#)

[Handle errors and exceptions](#)

[Logging](#)

[Single sign-on](#)

[Single sign-on with MSAL.js](#)

[Single sign-on with MSAL for iOS and macOS](#)

[SSO between MSAL apps](#)

[SSO between ADAL and MSAL apps](#)

[Integrate with ADFS](#)

[ADFS support in MSAL.NET](#)

[Integrate with Azure AD B2C](#)

[Android](#)

[JavaScript](#)

[.NET](#)

[iOS and macOS](#)

[Considerations and known issues](#)

[MSAL.NET](#)

[Web browsers](#)

[MSAL.js](#)

[Considerations using IE](#)

[Known issues- IE and Microsoft Edge](#)

[Known issues- Safari](#)

[MSAL for iOS and macOS SSL issues](#)

[Authentication protocol](#)

- [Application types and OAuth2.0](#)
 - [OAuth 2.0 and OpenID Connect protocols](#)
 - [OpenID Connect](#)
 - [OAuth 2.0 implicit grant flow](#)
 - [OAuth 2.0 auth code grant](#)
 - [OAuth 2.0 on-behalf-of flow](#)
 - [OAuth 2.0 client credentials grant](#)
 - [OAuth 2.0 device code flow](#)
 - [OAuth 2.0 resource owner password credentials grant](#)
 - [OAuth 2.0 SAML bearer assertion flow](#)
 - [Signing key rollover](#)
 - [ID tokens](#)
 - [Access tokens](#)
 - [Certificate credentials](#)
- [Application configuration](#)
 - [Applications and service principals](#)
 - [How and why apps are added to Azure AD](#)
 - [Redirect URI/reply URL restrictions and limitations](#)
 - [Validation differences by supported account types](#)
 - [Single tenant and multi-tenant apps](#)
- [Permissions and consent](#)
 - [Types of permissions and consent](#)
 - [Consent framework](#)
 - [Admin consent](#)
 - [Application consent experiences](#)
- [National Clouds](#)
 - [Overview](#)
 - [Authentication](#)
- [Automatic user provisioning \(SCIM\)](#)
 - [What is automatic user provisioning?](#)
 - [Building and integrating a SCIM endpoint](#)
- [How-to guides](#)
 - [Authentication](#)

[Configure role claims](#)

[Customize claims - Portal](#)

[Customize claims - PowerShell](#)

[Configure optional claims](#)

[Configure token lifetimes](#)

[Application configuration](#)

[Transitioning from App registrations \(Legacy\) to the new App registrations experience in the Azure portal](#)

[Transitioning from Application Registration Portal to the new App registrations experience in the Azure portal](#)

[Convert a single-tenant app to a multi-tenant app](#)

[Create service principal](#)

[Using Azure PowerShell](#)

[Using Azure portal](#)

[Restrict your app to a set of users](#)

[Add app roles in your application](#)

[Branding guidelines](#)

[Publisher domain](#)

[Terms of Service and Privacy Statement](#)

[Work with Microsoft Authentication Library](#)

[MSAL.NET](#)

[Acquire a token from the cache](#)

[Token cache serialization](#)

[Clear the token cache](#)

[Instantiate a public client with options](#)

[Instantiate a confidential client with options](#)

[Get consent for several resources](#)

[Provide your own HttpClient](#)

[MSAL.js](#)

[Avoid page reloads](#)

[Pass custom state in authentication requests](#)

[Prompt behavior](#)

[MSAL for iOS and macOS](#)

[MSAL for iOS and macOS differences](#)

[Configure keychain](#)

[Customize browsers and WebViews](#)

[Request custom claims](#)

[Redirect URI configuration](#)

[MSAL Java](#)

[Token cache serialization](#)

[Add and remove accounts from the token cache](#)

[Work with Visual Studio](#)

[Use the Active Directory connected service](#)

[Get started with .NET MVC projects](#)

[What happened to my .NET MVC project?](#)

[Get started with WebAPI projects](#)

[What happened to my WebAPI project?](#)

[Error during authentication detection](#)

[References](#)

[Authentication libraries](#)

[Identity and access APIs in Microsoft Graph](#)

[Application manifest](#)

[Android MSAL configuration file](#)

[Authentication and authorization error codes](#)

[Breaking changes](#)

[v1.0](#)

[Overview](#)

[Quickstarts](#)

[Set up a tenant](#)

[Configure an application](#)

[Register an app](#)

[Configure app to access web APIs](#)

[Configure app to expose web APIs](#)

[Modify accounts supported by an app](#)

[Remove an app](#)

[Single-page apps](#)

[AngularJS](#)

[JavaScript](#)

[Web apps](#)

[ASP .NET](#)

[Java](#)

[NodeJS](#)

[Python](#)

[Web APIs](#)

[ASP .NET](#)

[NodeJS](#)

[Mobile and desktop apps](#)

[Android](#)

[iOS](#)

[Windows Desktop .NET](#)

[Xamarin](#)

[Service-to-service](#)

[Daemon apps](#)

[.NET Core console \(daemon\)](#)

[Samples](#)

[Concepts](#)

[Why update to v2.0](#)

[Application types](#)

[Types of applications](#)

[Single-page application](#)

[Web app](#)

[Web API](#)

[Mobile & desktop app](#)

[Daemon or server application](#)

[Authentication](#)

[Authentication basics](#)

[OAuth 2.0 and OpenID Connect protocols](#)

- [OpenID Connect](#)
- [OAuth 2.0 implicit grant flow](#)
- [OAuth 2.0 auth code grant](#)
- [OAuth 2.0 on-behalf-of flow](#)
- [OAuth 2.0 client credentials grant](#)
- [SAML 2.0 protocol](#)
 - [Single sign-on](#)
 - [Sign-out](#)
 - [How Azure AD uses the SAML protocol](#)
- [WS-Federation](#)
 - [Federation metadata](#)
 - [Signing key rollover](#)
 - [ID tokens](#)
 - [Access tokens](#)
 - [Certificate credentials](#)
 - [SAML tokens](#)
- [Application configuration](#)
 - [Applications and service principals](#)
 - [How and why apps are added to Azure AD](#)
 - [Single tenant and multi-tenant apps](#)
- [Permissions and consent](#)
 - [Types of permissions and consent](#)
 - [Consent framework](#)
 - [Application consent experiences](#)
- [Conditional Access](#)
- [National Clouds](#)
- [How-to guides](#)
 - [Authentication](#)
 - [Configure single sign-on](#)
 - [Enable SSO on Android](#)
 - [Enable SSO on iOS](#)
 - [Debug SAML-based SSO](#)

[Configure claims](#)

[Configure role claims](#)

[Customize claims - Portal](#)

[Customize claims - PowerShell](#)

[Configure optional claims](#)

[Configure token lifetimes](#)

[ADAL error handling best practices](#)

[Application configuration](#)

[Convert a single-tenant app to a multi-tenant app](#)

[Create a service principal](#)

[Using Azure PowerShell](#)

[Using Azure portal](#)

[Restrict your app to a set of users](#)

[Add app roles in your application](#)

[Branding guidelines](#)

[Terms of Service and Privacy Statement](#)

[Work with Visual Studio](#)

[Use the Active Directory connected service](#)

[Get started with .NET MVC projects](#)

[What happened to my .NET MVC project?](#)

[Get started with WebAPI projects](#)

[What happened to my WebAPI project?](#)

[Error during authentication detection](#)

[Bring an app to market](#)

[Publish to App Source](#)

[Publish to Azure AD App Gallery](#)

[Publish to the Office 365 Seller Dashboard](#)

[Azure Access Control Service](#)

[Migrate from the Azure Access Control Service](#)

[Reactivate disabled Access Control Service namespaces](#)

[Use the Azure AD Graph API](#)

[AD FS for developers](#)

References

- [Authentication libraries](#)
- [Azure AD Graph API](#)
- [Application manifest](#)
- [Authentication and authorization error codes](#)
- [Breaking changes](#)

Resources

- [Glossary](#)
- [Azure roadmap](#)
- [Azure AD blog](#)
- [Microsoft identity platform developer blog](#)
- [Try Sign in with Microsoft](#)
- [Getting help](#)
- [Managed identities for Azure resources](#)

Evolution of Microsoft identity platform

7/8/2019 • 3 minutes to read • [Edit Online](#)

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

Up until now, most developers have worked with the Azure AD v1.0 platform to authenticate work and school accounts (provisioned by Azure AD) by requesting tokens from the Azure AD v1.0 endpoint, using Azure AD Authentication Library (ADAL), Azure portal for application registration and configuration, and Azure AD Graph API for programmatic application configuration.

With Microsoft identity platform (v2.0), expand your reach to these kinds of users:

- Work and school accounts (Azure AD provisioned accounts)
- Personal accounts (such as Outlook.com or Hotmail.com)
- Your customers who bring their own email or social identity (such as LinkedIn, Facebook, Google) via the Azure AD B2C offering

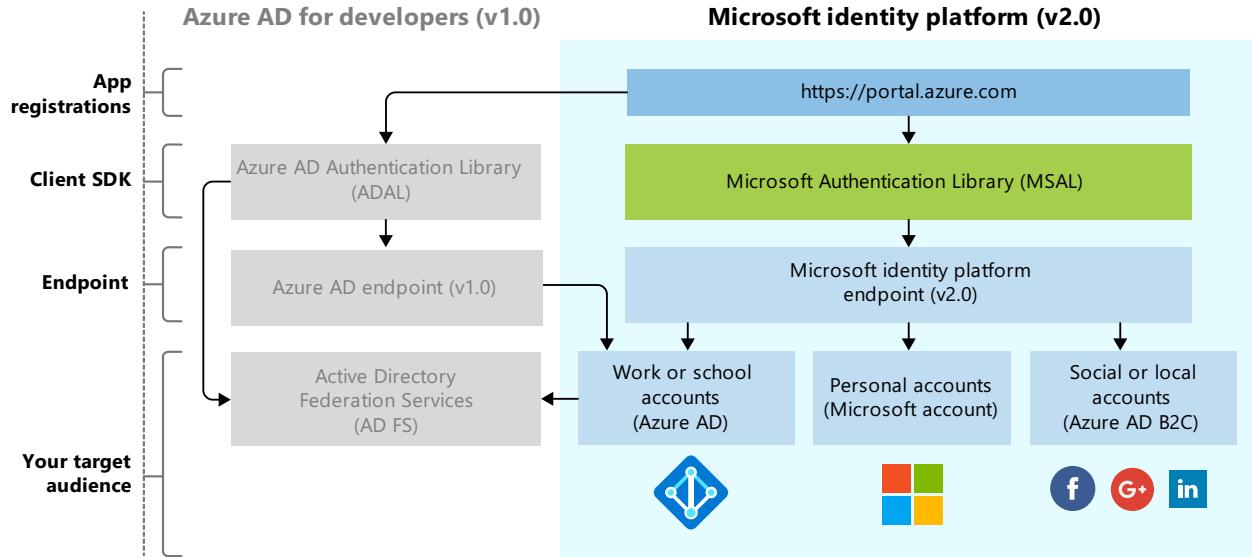
With the unified Microsoft identity platform, you can write code once and authenticate any Microsoft identity into your application. For several platforms, there's a fully supported open-source library called Microsoft Authentication Library (MSAL). MSAL is simple to use, provides great single sign-on (SSO) experiences for your users, helps you achieve high reliability and performance, and is developed using Microsoft Secure Development Lifecycle (SDL). When calling APIs, you can configure your application to take advantage of incremental consent, which allows you to delay the request for consent for more invasive scopes until the application's usage warrants this at runtime.

You can use the Azure portal to register and configure your application, and use the Microsoft Graph API for programmatic application configuration.

Update your application at your own pace. Applications built with ADAL libraries continue to be supported. Mixed application portfolios, that consist of applications built with ADAL and applications built with MSAL libraries, are also supported. This means that applications using the latest ADAL and the latest MSAL will deliver SSO across the portfolio, provided by the shared token cache between these libraries. Applications updated from ADAL to MSAL will maintain user sign-in state upon upgrade.

Microsoft identity platform experience

The following diagram shows the Microsoft identity experience at a high level, including the app registration experience, SDKs, endpoints, and supported identities.



App registration experience

The Azure portal **App registrations** experience is the one portal experience for managing all applications you've integrated with Microsoft identity platform. If you have been using the Application Registration Portal, start using the Azure portal app registration experience instead.

For integration with Azure AD B2C (when authenticating social or local identities), you'll need to register your application in a B2C tenant. This experience is also part of the Azure portal.

The **application API in Microsoft Graph** is currently in preview. Use this API to programmatically configure your applications integrated with Microsoft identity platform for authenticating any Microsoft identity. However, until this API reaches general availability, you should use the Azure AD Graph 1.6 API and the application manifest.

MSAL libraries

You can use the MSAL library to build applications that authenticate all Microsoft identities. The MSAL libraries in .NET and JavaScript are generally available. MSAL libraries for iOS and Android are in preview and suitable for use in a production environment. We provide the same production level support for MSAL libraries in preview as we do for versions of MSAL and ADAL that are generally available.

You can also use the MSAL libraries to integrate your application with Azure AD B2C.

Server-side libraries for building web apps and web APIs are generally available: [ASP.NET](#) and [ASP.NET Core](#)

Microsoft identity platform endpoint

Microsoft identity platform (v2.0) endpoint is now OIDC certified. It works with the Microsoft Authentication Libraries (MSAL) or any other standards-compliant library. It implements human readable scopes, in accordance with industry standards.

Next steps

Learn more about v1.0 and v2.0.

- [Microsoft identity platform \(v2.0\) overview](#)
- [Azure Active Directory for developers \(v1.0\) overview](#)

Microsoft identity platform (v2.0) overview

8/7/2019 • 2 minutes to read • [Edit Online](#)

Microsoft identity platform is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs, such as Microsoft Graph, or APIs that developers have built. The Microsoft identity platform consists of:

- **OAuth 2.0 and OpenID Connect standard-compliant authentication service** that enables developers to authenticate any Microsoft identity, including:
 - Work or school accounts (provisioned through Azure AD)
 - Personal Microsoft accounts (such as Skype, Xbox, and Outlook.com)
 - Social or local accounts (via Azure AD B2C)
- **Open-source libraries:** Microsoft Authentication Libraries (MSAL) and support for other standards-compliant libraries
- **Application management portal:** A registration and configuration experience built in the Azure portal, along with all your other Azure management capabilities.
- **Application configuration API and PowerShell:** which allows programmatic configuration of your applications through REST API (Microsoft Graph and Azure Active Directory Graph 1.6) and PowerShell, so you can automate your DevOps tasks.
- **Developer content:** conceptual and reference documentation, quickstart samples, code samples, tutorials, and how-to guides.

For developers, Microsoft identity platform offers seamless integration into innovations in the identity and security space, such as passwordless authentication, step-up authentication, and Conditional Access. You don't need to implement such functionality yourself: applications integrated with the Microsoft identity platform natively take advantage of such innovations.

With Microsoft identity platform, you can write code once and reach any user. You can build an app once and have it work across many platforms, or build an app that functions as a client as well as a resource application (API).

Getting started

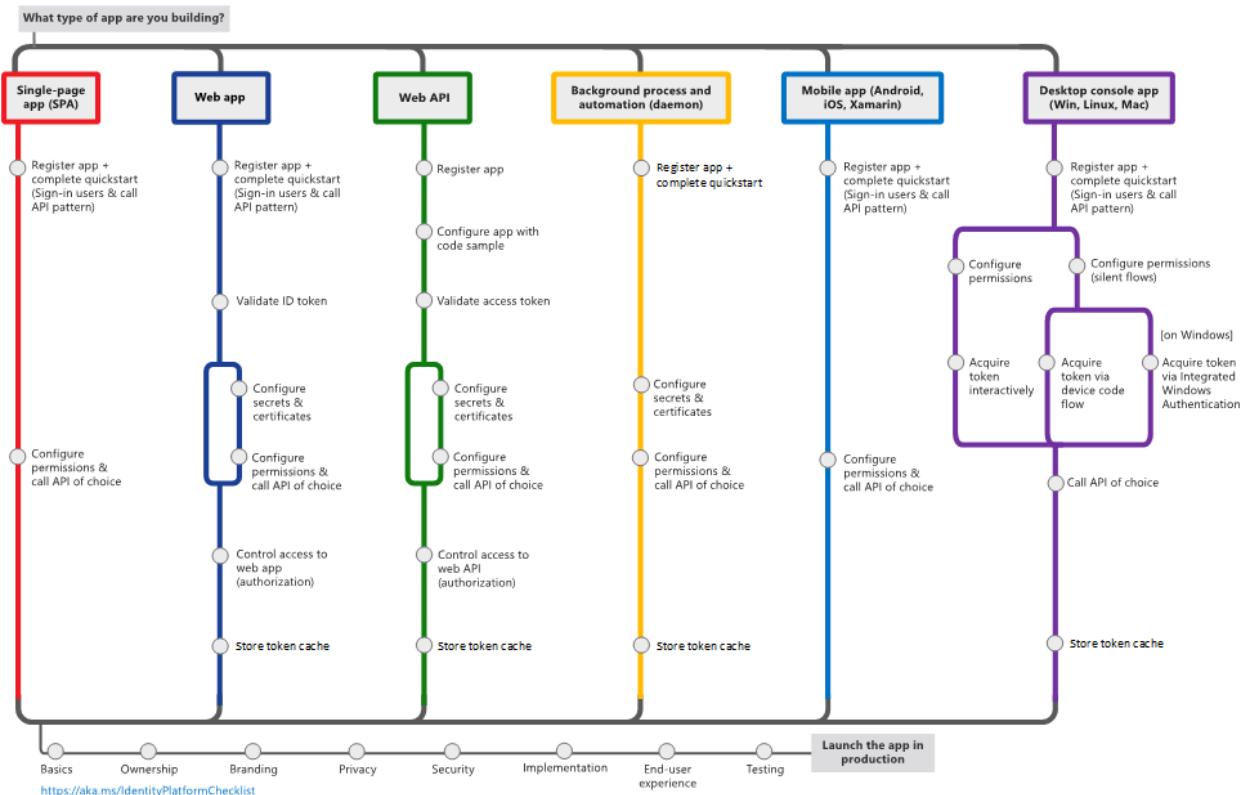
Working with identity doesn't have to be hard. Choose a [scenario](#) that applies to you— each scenario path has a quickstart and an overview page to get you up and running in minutes:

- [Build a single-page app](#)
- [Build a web app that signs in users](#)
- [Build a web app that calls web APIs](#)
- [Build a protected web API](#)
- [Build a web API that calls web APIs](#)
- [Build a desktop app](#)
- [Build a daemon app](#)
- [Build a mobile app](#)

The following chart outlines common authentication app scenarios – use it as a reference when integrating the Microsoft identity platform with your app.

Microsoft identity platform

<http://aka.ms/identityPlatform>



Next steps

If you'd like to learn more about core authentication concepts, we recommend you start with these topics:

- [Authentication flows and application scenarios](#)
- [Authentication basics](#)
- [Application and service principals](#)
- [Audiences](#)
- [Permissions and consent](#)
- [ID tokens and access tokens](#)

Build a data-rich application that calls [Microsoft Graph](#).

When you're ready to launch your app into a **production environment**, review these best practices:

- [Enable logging](#) in your application.
- Enable telemetry in your application.
- Enable [proxies and customize HTTP clients](#).
- Test your integration by following the [Microsoft identity platform integration checklist](#).

Learn more

If you're planning to build a customer-facing application that signs in social and local identities, see the [Azure AD B2C overview](#).

Quickstart: Set up a tenant

9/25/2019 • 3 minutes to read • [Edit Online](#)

The Microsoft identity platform allows developers to build apps targeting a wide variety of custom Microsoft 365 environments and identities. To get started using Microsoft identity platform, you will need access to an environment, also called an Azure AD tenant, that can register and manage apps, have access to Microsoft 365 data, and deploy custom Conditional Access and tenant restrictions.

A tenant is a representation of an organization. It's a dedicated instance of Azure AD that an organization or app developer receives when the organization or app developer creates a relationship with Microsoft-- like signing up for Azure, Microsoft Intune, or Microsoft 365.

Each Azure AD tenant is distinct and separate from other Azure AD tenants and has its own representation of work and school identities, consumer identities (if it's an Azure AD B2C tenant), and app registrations. An app registration inside of your tenant can allow authentications from accounts only within your tenant or all tenants.

Determining environment type

There are two types of environments you can create. Deciding which you need is based solely on the types of users your app will authenticate.

- Work and school (Azure AD accounts) or Microsoft accounts (such as outlook.com and live.com)
- Social and local accounts (Azure AD B2C)

The quickstart is broken into two scenarios depending on the type of app you want to build. If you need more help targeting an identity type, take a look at [about Microsoft identity platform](#)

Work and school accounts, or personal Microsoft accounts

Use an existing tenant

Many developers already have tenants through services or subscriptions that are tied to Azure AD tenants such as Microsoft 365 or Azure subscriptions.

1. To check the tenant, sign in to the [Azure portal](#) with the account you want to use to manage your application.
2. Check the upper right corner. If you have a tenant, you'll automatically be logged in and can see the tenant name directly under your account name.
 - Hover over your account name on the upper right-hand side of the Azure portal to see your name, email, directory / tenant ID (a GUID), and your domain.
 - If your account is associated with multiple tenants, you can select your account name to open a menu where you can switch between tenants. Each tenant has its own tenant ID.

TIP

If you need to find the tenant ID, you can:

- Hover over your account name to get the directory / tenant ID, or
- Select **Azure Active Directory > Properties > Directory ID** in the Azure portal

If you don't have an existing tenant associated with your account, you'll see a GUID under your account name and you won't be able to perform actions like registering apps until you follow the steps of the next section.

Create a new Azure AD tenant

If you don't already have an Azure AD tenant or want to create a new one for development, follow the [directory creation experience](#). You will have to provide the following info to create your new tenant:

- **Organization name**
- **Initial domain** - this will be part of *.onmicrosoft.com. You can customize the domain more later.
- **Country or region**

NOTE

When naming your tenant, use alphanumeric characters. Special characters are not allowed. The name must not exceed 256 characters.

Social and local accounts

To begin building apps that sign in social and local accounts, you'll need to create an Azure AD B2C tenant. To begin, follow [creating an Azure AD B2C tenant](#).

Next steps

- Try a coding quickstart and begin authenticating users.
- For more in-depth code samples, check out the **Tutorials** section of the documentation.
- Want to deploy your app to the cloud? Check out [deploying containers to Azure](#).

Quickstart: Register an application with the Microsoft identity platform

11/4/2019 • 3 minutes to read • [Edit Online](#)

Enterprise developers and software-as-a-service (SaaS) providers can develop commercial cloud services or line-of-business applications that can be integrated with Microsoft identity platform to provide secure sign-in and authorization for their services.

This quickstart shows you how to add and register an application using the **App registrations** experience in the Azure portal so that your app can be integrated with the Microsoft identity platform. To learn more about the new features and improvements in the new app registrations experience, see [this blog post](#).

Register a new application using the Azure portal

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the Azure AD tenant that you want.
3. Search for and select **Azure Active Directory**. On the **Active Directory** page, select **App registrations** and then select **New registration**.
4. When the **Register an application** page appears, enter your application's registration information:
 - **Name** - Enter a meaningful application name that will be displayed to users of the app.
 - **Supported account types** - Select which accounts you would like your application to support.

SUPPORTED ACCOUNT TYPES	DESCRIPTION
Accounts in this organizational directory only	Select this option if you're building a line-of-business (LOB) application. This option is not available if you're not registering the application in a directory. This option maps to Azure AD only single-tenant. This is the default option unless you're registering the app outside of a directory. In cases where the app is registered outside of a directory, the default is Azure AD multi-tenant and personal Microsoft accounts.
Accounts in any organizational directory	Select this option if you would like to target all business and educational customers. This option maps to an Azure AD only multi-tenant. If you registered the app as Azure AD only single-tenant, you can update it to be Azure AD multi-tenant and back to single-tenant through the Authentication blade.

SUPPORTED ACCOUNT TYPES	DESCRIPTION
Accounts in any organizational directory and personal Microsoft accounts	Select this option to target the widest set of customers. This option maps to Azure AD multi-tenant and personal Microsoft accounts. If you registered the app as Azure AD multi-tenant and personal Microsoft accounts, you cannot change this in the UI. Instead, you must use the application manifest editor to change the supported account types.

- **Redirect URI (optional)** - Select the type of app you're building, **Web** or **Public client (mobile & desktop)**, and then enter the redirect URI (or reply URL) for your application.

- For web applications, provide the base URL of your app. For example, `https://localhost:31544` might be the URL for a web app running on your local machine. Users would use this URL to sign in to a web client application.
- For public client applications, provide the URI used by Azure AD to return token responses. Enter a value specific to your application, such as `myapp://auth`.

To see specific examples for web applications or native applications, check out our [quickstarts](#).

5. When finished, select **Register**.

The screenshot shows the 'Register an application' form in the Azure portal. At the top, there's a breadcrumb navigation: Home > App registrations > Register an application. The main title is 'Register an application'. Below it, there's a field labeled 'Name' with the placeholder 'The user-facing display name for this application (this can be changed later.)'. A text input field contains 'ContosoApp_1' with a green checkmark icon to its right. Under 'Supported account types', there are three radio button options: 'Accounts in this organizational directory only (Contoso Enterprises)' (selected), 'Accounts in any organizational directory', and 'Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)'. Below this is a link 'Help me choose...'. The 'Redirect URI (optional)' section includes a dropdown menu set to 'Web' and a text input field containing 'https://contosoapp1/auth' with a green checkmark icon to its right. At the bottom of the form is a blue 'Register' button.

Azure AD assigns a unique application (client) ID to your app, and you're taken to your application's **Overview** page. To add additional capabilities to your application, you can select other configuration options including branding, certificates and secrets, API permissions, and more.

Home > App registrations > ContosoApp_1

ContosoApp_1
PREVIEW

Create application 12:18 PM X
Successfully created application ContosoApp_1.

Overview Delete Endpoints

Display name **ContosoApp_1**
Application (client) ID 95c232bc-5ab2-4954-8640-2a865eeb8597
Directory (tenant) ID 73e589d0-adbb-451c-8382-8c7f992efcccd

Supported account types **My organization only**
Redirect URLs **1 web, 0 public client**
Managed application in local directory **ContosoApp_1**

Call APIs

Build more powerful apps with rich user and business data from Microsoft services and your own company's data sources.

View API Permissions

Documentation

Azure Active Directory for Developers
Authentication scenarios
Authentication libraries
Code samples and tutorials
Microsoft Graph
Glossary
Help and Support

Sign in users in 5 minutes

Use our SDKs to sign in users and call APIs in a few steps

View all quickstart guides

The screenshot shows the Azure portal's 'App registrations' section. A new application named 'ContosoApp_1' has been successfully created. The 'Overview' tab is selected, displaying basic details like the display name, application ID, and tenant ID. It also shows supported account types ('My organization only'), redirect URLs ('1 web, 0 public client'), and a note that it's a managed application in a local directory. Below this, there's a 'Call APIs' section with icons for various Microsoft services, followed by a 'Sign in users in 5 minutes' section with icons for JS, ASP.NET, Windows, Android, and Apple. Navigation links on the left include 'Branding', 'Authentication', 'Certificates & secrets', 'API permissions', 'Expose an API', 'Owners', and 'Manifest'. Under 'Support + Troubleshooting', there are 'Troubleshooting' and 'New support request' options.

Next steps

- Learn about the [permissions and consent](#).
- To enable additional configuration features in your application registration, such as credentials and permissions, and enable sign-in for users from other tenants, see these quickstarts:
 - [Configure a client application to access web APIs](#)
 - [Configure an application to expose web APIs](#)
 - [Modify the accounts supported by an application](#)
- Choose a [quickstart](#) to quickly build an app and add functionality like getting tokens, refreshing tokens, signing in a user, displaying some user info, and more.
- Learn more about the two Azure AD objects that represent a registered application and the relationship between them, see [Application objects and service principal objects](#).
- Learn more about the branding guidelines you should use when developing apps, see [Branding guidelines for applications](#).

Quickstart: Configure a client application to access web APIs

11/4/2019 • 7 minutes to read • [Edit Online](#)

For a web/confidential client application to be able to participate in an authorization grant flow that requires authentication (and obtain an access token), it must establish secure credentials. The default authentication method supported by the Azure portal is client ID + secret key.

Additionally, before a client can access a web API exposed by a resource application (such as Microsoft Graph API), the consent framework ensures the client obtains the permission grant required based on the permissions requested. By default, all applications can choose permissions from the Microsoft Graph API. The [Graph API "Sign-in and read user profile" permission](#) is selected by default. You can select from [two types of permissions](#) for each desired web API:

- **Application permissions** - Your client application needs to access the web API directly as itself (no user context). This type of permission requires administrator consent and is also not available for public (desktop and mobile) client applications.
- **Delegated permissions** - Your client application needs to access the web API as the signed-in user, but with access limited by the selected permission. This type of permission can be granted by a user unless the permission requires administrator consent.

NOTE

Adding a delegated permission to an application does not automatically grant consent to the users within the tenant. Users must still manually consent for the added delegated permissions at runtime, unless the administrator grants consent on behalf of all users.

In this quickstart, we'll show you how to configure your app to:

- [Add redirect URIs to your application](#)
- [Configure advanced settings for your application](#)
- [Modify supported account types](#)
- [Add credentials to your web application](#)
- [Add permissions to access web APIs](#)

Prerequisites

To get started, make sure you complete these prerequisites:

- Learn about the supported [permissions and consent](#), which is important to understand when building applications that need to be used by other users or applications.
- Have a tenant that has applications registered to it.
 - If you don't have apps registered, [learn how to register applications with the Microsoft identity platform](#).

Sign in to the Azure portal and select the app

Before you can configure the app, follow these steps:

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.

2. If your account gives you access to more than one tenant, select your account in the top-right corner, and set your portal session to the desired Azure AD tenant.
3. Search for and select **Azure Active Directory**.
4. From the left pane, select **App registrations**.
5. Find and select the application you want to configure. Once you've selected the app, you'll see the application's **Overview** or main registration page.
6. Follow the steps to configure your application to access web APIs:
 - [Add redirect URIs to your application](#)
 - [Configure advanced settings for your application](#)
 - [Modify supported account types](#)
 - [Add credentials to your web application](#)
 - [Add permissions to access web APIs](#)

Add redirect URI(s) to your application

To add a redirect URI to your application:

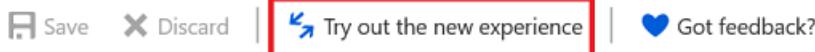
1. From the app's **Overview** page, select the **Authentication** section.
2. To add a custom redirect URI for web and public client applications, follow these steps:
 - a. Locate the **Redirect URI** section.
 - b. Select the type of application you're building, **Web** or **Public client (mobile & desktop)**.
 - c. Enter the Redirect URI for your application.
 - For web applications, provide the base URL of your application. For example, `http://localhost:31544` might be the URL for a web application running on your local machine. Users would use this URL to sign into a web client application.
 - For public applications, provide the URI used by Azure AD to return token responses. Enter a value specific to your application, for example: `https://MyFirstApp`.
3. To choose from suggested Redirect URIs for public clients (mobile, desktop), follow these steps:
 - a. Locate the Suggested **Redirect URIs for public clients (mobile, desktop)** section.
 - b. Select the appropriate Redirect URI(s) for your application using the checkboxes. You can also enter a custom redirect URI. If you're not sure what to use, check out the library documentation.

There are certain restrictions that apply to redirect URIs. Learn more about [redirect URI restrictions and limitations](#).

NOTE

Try out the new **Authentication** settings experience where you can configure settings for your application based on the platform or device that you want to target.

To see this view, select **Try out the new experience** from the default **Authentication** page view.



This takes you to the [new Platform configurations page](#).

Configure advanced settings for your application

Depending on the application you're registering, there are some additional settings that you may need to configure, such as:

- **Logout URL**

- For single-page apps, you can enable **Implicit grant** and select the tokens that you'd like the authorization endpoint to issue.
- For desktop apps that are acquiring tokens with Integrated Windows Authentication, device code flow, or username/password in the **Default client type** section, configure the **Treat application as public client** setting to **Yes**.
- For legacy apps that were using the Live SDK to integrate with the Microsoft account service, configure **Live SDK support**. New apps don't need this setting.
- **Default client type**

Modify supported account types

The **Supported account types** specify who can use the application or access the API.

Once you've [configured the supported account types](#) when you initially registered the application, you can only change this setting using the application manifest editor if:

- You change account types from **AzureADMyOrg** or **AzureADMultipleOrgs** to **AzureADandPersonalMicrosoftAccount**, or vice versa.
- You change account types from **AzureADMyOrg** to **AzureADMultipleOrgs**, or vice versa.

To change the supported account types for an existing app registration:

- See [Configure the application manifest](#) and update the `signInAudience` key.

Configure platform settings for your application

New_Web_App - Platform configurations



Save Discard | Switch to the old experience | Got feedback?

Platform configurations

Depending on the platform or device this application is targeting, additional configuration may be required such as redirect URLs, specific authentication settings, or fields specific to the platform.

[Add a platform](#)

Web

REDIRECT URI
http://localhost:31544

[Add URI](#)

LOGOUT URL
e.g. https://myapp.com/logout

IMPLICIT GRANT

Allows an application to request a token directly from the authorization endpoint. Recommended only if the application has a single page architecture (SPA), has no backend components, or invokes a Web API via JavaScript.

To enable the implicit grant flow, select the tokens you would like to be issued by the authorization endpoint:

Access tokens
 ID tokens

Supported account types

Who can use this application or access this API?

Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)

All users with a work or school, or personal Microsoft account can use your application or API. This includes Office 365 subscribers.

⚠ To change the supported accounts for an existing registration, use the manifest editor. Take care, as certain properties may cause errors for personal accounts.

Advanced settings

Live SDK support

Allow direct integration with the Microsoft account service (login.live.com). Required for integration with Microsoft account SDKs such as Xbox or Bing Ads

Yes No

Default client type

Treat application as a public client. Required for the use of the following flows where a redirect URI is not used:

Yes No

- Resource owner password credential (ROPC) [Learn more](#)
- Device code flow [Learn more](#)

To configure application settings based on the platform or device, you're targeting:

- In the **Platform configurations** page, select **Add a platform** and choose from the available options.

Configure platforms

Web applications



Web

Single-page apps, Web apps

Mobile applications



iOS

Objective-C, Swift, Xamarin



Android

Java, Kotlin, Xamarin

Desktop + devices



Desktop + devices

Windows, UWP, Console, IoT & Limited-entry Devices, Classic iOS + Android

- Enter the settings info based on the platform you selected.

PLATFORM	CHOICES	CONFIGURATION SETTINGS
Web applications	Web	Enter the Redirect URI for your application.
Mobile applications	iOS	Enter the app's Bundle ID , which you can find in XCode in Info.plist, or Build Settings. Adding the bundle ID automatically creates a redirect URI for the application.
	Android	* Provide the app's Package name , which you can find in the AndroidManifest.xml file. * Generate and enter the Signature hash . Adding the signature hash automatically creates a redirect URI for the application.
Desktop + devices	Desktop + devices	* Optional. Select one of the recommended Suggested redirect URLs if you're building apps for desktop and devices. * Optional. Enter a Custom redirect URI , which is used as the location where Azure AD will redirect users in response to authentication requests. For example, for .NET Core applications where you want interaction, use <code>https://localhost</code> .

IMPORTANT

For mobile applications that aren't using the latest MSAL library or not using a broker, you must configure the redirect URLs for these applications in **Desktop + devices**.

3. Depending on the platform you chose, there may be additional settings that you can configure. For **Web** apps, you can:

- Add more redirect URLs
- Configure **Implicit grant** to select the tokens you'd like to be issued by the authorization endpoint:
 - For single-page apps, select both **Access tokens** and **ID tokens**
 - For web apps, select **ID tokens**

Add credentials to your web application

To add a credential to your web application:

1. From the app's **Overview** page, select the **Certificates & secrets** section.
2. To add a certificate, follow these steps:
 - a. Select **Upload certificate**.
 - b. Select the file you'd like to upload. It must be one of the following file types: .cer, .pem, .crt.
 - c. Select **Add**.
3. To add a client secret, follow these steps:
 - a. Select **New client secret**.
 - b. Add a description for your client secret.
 - c. Select a duration.
 - d. Select **Add**.

NOTE

After you save the configuration changes, the right-most column will contain the client secret value. **Be sure to copy the value** for use in your client application code as it's not accessible once you leave this page.

Add permissions to access web APIs

To add permission(s) to access resource APIs from your client:

1. From the app's **Overview** page, select **API permissions**.
2. Select the **Add a permission** button.
3. By default, the view allows you to select from **Microsoft APIs**. Select the section of APIs that you're interested in:
 - **Microsoft APIs** - Lets you select permissions for Microsoft APIs such as Microsoft Graph.
 - **APIs my organization uses** - Lets you select permissions for APIs that have been exposed by your organization, or APIs that your organization has integrated with.
 - **My APIs** - Lets you select permissions for APIs that you have exposed.
4. Once you've selected the APIs, you'll see the **Request API Permissions** page. If the API exposes both delegated and application permissions, select which type of permission your application needs.
5. When finished, select **Add permissions**. You will return to the **API permissions** page, where the permissions have been saved and added to the table.

Next steps

Learn about these other related app management quickstarts for apps:

- [Register an application with the Microsoft identity platform](#)
- [Configure an application to expose web APIs](#)
- [Modify the accounts supported by an application](#)
- [Remove an application registered with the Microsoft identity platform](#)

To learn more about the two Azure AD objects that represent a registered application and the relationship between them, see [Application objects and service principal objects](#).

To learn more about the branding guidelines you should use when developing applications with Azure Active Directory, see [Branding guidelines for applications](#).

Quickstart: Configure an application to expose web APIs

8/13/2019 • 5 minutes to read • [Edit Online](#)

You can develop a web API and make it available to client applications by exposing [permissions/scopes](#) and [roles](#). A correctly configured web API is made available just like the other Microsoft web APIs, including the Graph API and the Office 365 APIs.

In this quickstart, you'll learn how to configure an application to expose a new scope to make it available to client applications.

Prerequisites

To get started, make sure you complete these prerequisites:

- Learn about the supported [permissions and consent](#), which is important to understand when building applications that need to be used by other users or applications.
- Have a tenant that has applications registered to it.
 - If you don't have apps registered, [learn how to register applications with the Microsoft identity platform](#).

Sign in to the Azure portal and select the app

Before you can configure the app, follow these steps:

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. In the left-hand navigation pane, select the **Azure Active Directory** service and then select **App registrations**.
4. Find and select the application you want to configure. Once you've selected the app, you'll see the application's **Overview** or main registration page.
5. Choose which method you want to use, UI or application manifest, to expose a new scope:
 - [Expose a new scope through the UI](#)
 - [Expose a new scope or role through the application manifest](#)

Expose a new scope through the UI

To expose a new scope through the UI:

1. From the app's **Overview** page, select the **Expose an API** section.
2. Select **Add a scope**.
3. If you have not set an **Application ID URI**, you will see a prompt to enter one. Enter your application ID URI or use the one provided and then select **Save and continue**.
4. When the **Add a scope** page appears, enter your scope's information:

FIELD	DESCRIPTION
Scope name	Enter a meaningful name for your scope. For example, <code>Employees.Read.All</code> .
Who can consent	Select whether this scope can be consented to by users, or if admin consent is required. Select Admins only for higher-privileged permissions.
Admin consent display name	Enter a meaningful description for your scope, which admins will see. For example, <code>Read-only access to Employee records</code>
Admin consent description	Enter a meaningful description for your scope, which admins will see. For example, <code>Allow the application to have read-only access to all Employee data.</code>

If users can consent to your scope, also add values for the following fields:

FIELD	DESCRIPTION
User consent display name	<p>Enter a meaningful name for your scope, which users will see.</p> <p>For example,</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">Read-only access to your Employee records</div>
User consent description	<p>Enter a meaningful description for your scope, which users will see.</p> <p>For example,</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">Allow the application to have read-only access to your Employee data.</div>

5. Set the **State** and select **Add scope** when you're done.
6. Follow the steps to [verify that the web API is exposed to other applications](#).

Expose a new scope or role through the application manifest

The screenshot shows the Microsoft Azure portal interface. On the left, there's a dark sidebar with various service icons and links like 'Create a resource', 'All services', 'Dashboard', etc. The main content area has a header 'ContosoApp_1 - Manifest' with tabs for 'Overview', 'Quickstart', 'Manage', 'Manifest' (which is selected), and 'Support + Troubleshooting'. Below the tabs is a note about updating the application via its JSON representation. The right side contains the actual JSON manifest code, which is quite long and includes fields like 'id', 'accessTokenAcceptedVersion', 'allowPublicClient', 'appId', 'appRoles', 'oauth2AllowUrlPathMatching', 'createdDateTime', 'groupMembershipClaims', 'identifierUris', 'informationalUrls', 'keyCredentials', 'knownClientApplications', 'logInUri', 'logOutUri', 'name', 'oauth2AllowIdTokenImplicitFlow', 'oauth2AllowImplicitFlow', 'oauth2Permissions', 'oauth2RequirePostResponse', 'orgRestrictions', 'parentalControlSettings', 'passwordCredentials', 'preAuthorizedApplications', 'publisherDomain', and 'replyUrlsWithType'.

```

1 {
2   "id": "495f2173-2e21-409a-8438-9be709a685ac",
3   "accessTokenAcceptedVersion": null,
4   "allowPublicClient": null,
5   "appId": "95c232bc-5ab2-4954-8640-2a865eeb8597",
6   "appRoles": [],
7   "oauth2AllowUrlPathMatching": false,
8   "createdDateTime": "2018-10-12T19:18:41Z",
9   "groupMembershipClaims": null,
10  "identifierUris": [],
11  "informationalUrls": {
12    "termsOfService": null,
13    "support": null,
14    "privacy": null,
15    "marketing": null
16  },
17  "keyCredentials": [],
18  "knownClientApplications": [],
19  "logInUri": null,
20  "logOutUri": null,
21  "name": "ContosoApp_1",
22  "oauth2AllowIdTokenImplicitFlow": false,
23  "oauth2AllowImplicitFlow": false,
24  "oauth2Permissions": [],
25  "oauth2RequirePostResponse": false,
26  "orgRestrictions": [],
27  "parentalControlSettings": {
28    "countriesBlockedForMinors": [],
29    "legalAgeGroupRule": "Allow"
30  },
31  "passwordCredentials": [],
32  "preAuthorizedApplications": [],
33  "publisherDomain": null,
34  "replyUrlsWithType": [
35    {
36      "url": "https://contosoapp1/auth",
37      "type": "Web"
38    }
39  ],
40  "requiredResourceAccess": [
41    {
42      "resourceAppId": "00000003-0000-0000-0000-000000000000",
43      "resourceAccess": [
44        {
45          "id": "e1fe6dd8-ba31-4d61-89e7-88639da4683d",
46          "type": "Scope"
47        }
48      ]
49    },
50    {
51      "samlMetadataUrl": null,
52      "signInUrl": null,
53      "signInAudience": "AzureADMyOrg",
54      "tags": [],
55      "tokenEncryptionKeyId": null
56    }
57  ]
58}

```

To expose a new scope through the application manifest:

- From the app's **Overview** page, select the **Manifest** section. A web-based manifest editor opens, allowing you to **Edit** the manifest within the portal. Optionally, you can select **Download** and edit the manifest locally, and then use **Upload** to reapply it to your application.

The following example shows how to expose a new scope called `Employees.Read.All` in the resource/API by adding the following JSON element to the `oauth2Permissions` collection.

```
{
  "adminConsentDescription": "Allow the application to have read-only access to all Employee data.",
  "adminConsentDisplayName": "Read-only access to Employee records",
  "id": "2b351394-d7a7-4a84-841e-08a6a17e4cb8",
  "isEnabled": true,
  "type": "User",
  "userConsentDescription": "Allow the application to have read-only access to your Employee data.",
  "userConsentDisplayName": "Read-only access to your Employee records",
  "value": "Employees.Read.All"
}
```

NOTE

The `id` value must be generated programmatically or by using a GUID generation tool such as [guidgen](#). The `id` represents a unique identifier for the scope as exposed by the web API. Once a client is appropriately configured with permissions to access your web API, it is issued an OAuth 2.0 access token by Azure AD. When the client calls the web API, it presents the access token that has the scope (scp) claim set to the permissions requested in its application registration.

You can expose additional scopes later as necessary. Consider that your web API might expose multiple scopes associated with a variety of different functions. Your resource can control access to the web API at runtime by evaluating the scope (`scp`) claim(s) in the received OAuth 2.0 access token.

2. When finished, click **Save**. Now your web API is configured for use by other applications in your directory.
3. Follow the steps to [verify that the web API is exposed to other applications](#).

Verify the web API is exposed to other applications

1. Go back to your Azure AD tenant, select **App registrations**, find and select the client application you want to configure.
2. Repeat the steps outlined in [Configure a client application to access web APIs](#).
3. When you get to the step to [select an API](#), select your resource. You should see the new scope, available for client permission requests.

More on the application manifest

The application manifest serves as a mechanism for updating the application entity, which defines all attributes of an Azure AD application's identity configuration. For more information on the Application entity and its schema, see the [Graph API Application entity documentation](#). The article contains complete reference information on the Application entity members used to specify permissions for your API, including:

- The `appRoles` member, which is a collection of [AppRole](#) entities, used to define [application permissions](#) for a web API.
- The `oauth2Permissions` member, which is a collection of [OAuth2Permission](#) entities, used to define [delegated permissions](#) for a web API.

For more information on application manifest concepts in general, see [Understanding the Azure Active Directory application manifest](#).

Next steps

Learn about these other related app management quickstarts for apps:

- [Register an application with the Microsoft identity platform](#)
- [Configure a client application to access web APIs](#)
- [Modify the accounts supported by an application](#)
- [Remove an application registered with the Microsoft identity platform](#)

To learn more about the two Azure AD objects that represent a registered application and the relationship between them, see [Application objects and service principal objects](#).

To learn more about the branding guidelines you should use when developing applications with Azure Active Directory, see [Branding guidelines for applications](#).

Quickstart: Modify the accounts supported by an application

7/22/2019 • 4 minutes to read • [Edit Online](#)

When registering an application in the Microsoft identity platform, you may want your application to be accessed only by users in your organization. Alternatively, you may also want your application to be accessible by users in external organizations, or by users in external organizations as well as users that are not necessarily part of an organization (personal accounts).

In this quickstart, you'll learn how to modify your application's configuration to change who, or what accounts, can access the application.

Prerequisites

To get started, make sure you complete these prerequisites:

- Learn about the supported [permissions and consent](#), which is important to understand when building applications that need to be used by other users or applications.
- Have a tenant that has applications registered to it.
 - If you don't have apps registered, [learn how to register applications with the Microsoft identity platform](#).

Sign in to the Azure portal and select the app

Before you can configure the app, follow these steps:

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. In the left-hand navigation pane, select the **Azure Active Directory** service and then select **App registrations**.
4. Find and select the application you want to configure. Once you've selected the app, you'll see the application's **Overview** or main registration page.
5. Follow the steps to [change the application registration to support different accounts](#).
6. If you have a single-page application, [enable OAuth 2.0 implicit grant](#).

Change the application registration to support different accounts

If you are writing an application that you want to make available to your customers or partners outside of your organization, you need to update the application definition in the Azure portal.

IMPORTANT

Azure AD requires the Application ID URI of multi-tenant applications to be globally unique. The App ID URI is one of the ways an application is identified in protocol messages. For a single-tenant application, it is sufficient for the App ID URI to be unique within that tenant. For a multi-tenant application, it must be globally unique so Azure AD can find the application across all tenants. Global uniqueness is enforced by requiring the App ID URI to have a host name that matches a verified domain of the Azure AD tenant. For example, if the name of your tenant is contoso.onmicrosoft.com, then a valid App ID URI would be <https://contoso.onmicrosoft.com/myapp>. If your tenant has a verified domain of contoso.com, then a valid App ID URI would also be <https://contoso.com/myapp>. If the App ID URI doesn't follow this pattern, setting an application as multi-tenant fails.

To change who can access your application

1. From the app's **Overview** page, select the **Authentication** section and change the value selected under **Supported account types**.
 - Select **Accounts in this directory only** if you are building a line-of-business (LOB) application. This option is not available if the application is not registered in a directory.
 - Select **Accounts in any organizational directory** if you would like to target all business and educational customers.
 - Select **Accounts in any organizational directory and personal Microsoft accounts** to target the widest set of customers.
2. Select **Save**.

Enable OAuth 2.0 implicit grant for single-page applications

Single-page applications (SPAs) are typically structured with a JavaScript-heavy front end that runs in the browser, which calls the application's web API back-end to perform its business logic. For SPAs hosted in Azure AD, you use OAuth 2.0 Implicit Grant to authenticate the user with Azure AD and obtain a token that you can use to secure calls from the application's JavaScript client to its back-end web API.

After the user has granted consent, this same authentication protocol can be used to obtain tokens to secure calls between the client and other web API resources configured for the application. To learn more about the implicit authorization grant, and help you decide whether it's right for your application scenario, learn about the OAuth 2.0 implicit grant flow in Azure AD [v1.0](#) and [v2.0](#).

By default, OAuth 2.0 implicit grant is disabled for applications. You can enable OAuth 2.0 implicit grant for your application by following the steps outlined below.

To enable OAuth 2.0 implicit grant

1. From the app's **Overview** page, select the **Authentication** section.
2. Under **Advanced settings**, locate the **Implicit grant** section.
3. Select **ID tokens**, **Access tokens**, or both.
4. Select **Save**.

Next steps

Learn about these other related app management quickstarts for apps:

- [Register an application with the Microsoft identity platform](#)
- [Configure a client application to access web APIs](#)
- [Configure an application to expose web APIs](#)
- [Remove an application registered with the Microsoft identity platform](#)

To learn more about the two Azure AD objects that represent a registered application and the relationship between them, see [Application objects and service principal objects](#).

To learn more about the branding guidelines you should use when developing applications with Azure Active Directory, see [Branding guidelines for applications](#).

Quickstart: Remove an application registered with the Microsoft identity platform

5/10/2019 • 2 minutes to read • [Edit Online](#)

Enterprise developers and software-as-a-service (SaaS) providers who have registered applications with Microsoft identity platform may need to remove an application's registration.

In this quickstart, you'll learn how to:

- Remove an application authored by you or your organization
- Remove an application authored by another organization

Prerequisites

You must have a tenant that has applications registered to it. To learn how to add and register apps, see [Register an application with the Microsoft identity platform](#).

Remove an application authored by you or your organization

Applications that you or your organization have registered are represented by both an application object and service principal object in your tenant. For more information, see [Application Objects and Service Principal Objects](#).

To remove an application

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. In the left-hand navigation pane, select the **Azure Active Directory** service, then select **App registrations**. Find and select the application that you want to configure. Once you've selected the app, you'll see the application's **Overview** page.
4. From the **Overview** page, select **Delete**.
5. Select **Yes** to confirm that you want to delete the app.

NOTE

To delete an application, you need to be listed as an owner of the application or have admin privileges.

Remove an application authored by another organization

If you are viewing **App registrations** in the context of a tenant, a subset of the applications that appear under the **All apps** tab are from another tenant and were registered into your tenant during the consent process. More specifically, they are represented by only a service principal object in your tenant, with no corresponding application object. For more information on the differences between application and service principal objects, see [Application and service principal objects in Azure AD](#).

In order to remove an application's access to your directory (after having granted consent), the company administrator must remove its service principal. The administrator must have global admin access, and can remove the application through the Azure portal or use the [Azure AD PowerShell Cmdlets](#) to remove access.

Next steps

Learn about these other related app management quickstarts:

- [Register an application with the Microsoft identity platform](#)
- [Configure a client application to access web APIs](#)
- [Configure an application to expose web APIs](#)
- [Modify the accounts supported by an application](#)

Quickstart: Sign in users and get an access token in a JavaScript SPA

11/7/2019 • 7 minutes to read • [Edit Online](#)

In this quickstart, you use a code sample to learn how a JavaScript single-page application (SPA) can sign in users of personal accounts, work accounts, and school accounts. A JavaScript SPA can also get an access token to call the Microsoft Graph API or any web API. (See [How the sample works](#) for an illustration.)

Prerequisites

- Azure subscription - [create one for free](#)
- [Node.js](#).
- Either [Visual Studio Code](#) (to edit project files), or [Visual Studio 2019](#) (to run the project as a Visual Studio Solution).

Register and download your quickstart application

To start your quickstart application, use either of the following options.

Option 1 (Express): Register and auto configure your app and then download your code sample

1. Sign in to the [Azure portal](#) by using either a work or school account, or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select the account at the top right, and then set your portal session to the Azure Active Directory (Azure AD) tenant you want to use.
3. Go to the new [Azure portal - App registrations](#) pane.
4. Enter a name for your application, and select **Register**.
5. Follow the instructions to download and automatically configure your new application.

Option 2 (Manual): Register and manually configure your application and code sample

Step 1: Register your application

1. Sign in to the [Azure portal](#) by using either a work or school account, or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account at the top right, and then set your portal session to the Azure AD tenant you want to use.
3. Go to the Microsoft identity platform for developers [App registrations](#) page.
4. Select **New registration**.
5. When the **Register an application** page appears, enter a name for your application.
6. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
7. Under the **Redirect URI** section, in the drop-down list, select the **Web** platform, and then set the value to `http://localhost:30662/`.
8. Select **Register**. On the app **Overview** page, note the **Application (client) ID** value for later use.
9. This quickstart requires the [Implicit grant flow](#) to be enabled. In the left pane of the registered application, select **Authentication**.
10. In the **Advanced settings** section, under **Implicit grant**, select the **ID tokens** and **Access tokens** check

boxes. ID tokens and access tokens are required, because this app needs to sign in users and call an API.

11. At the top of the pane, select **Save**.

Step 2: Download the project

Select the option that's suitable to your development environment:

- To run the project with a web server by using Node.js, [download the core project files](#). To open the files, use an editor such as [Visual Studio Code](#).
- (Optional) To run the project with the IIS server, [download the Visual Studio project](#). Extract the zip file to a local folder (for example, *C:\Azure-Samples*).

Step 3: Configure your JavaScript app

In the *JavaScriptSPA* folder, edit *index.html*, and set the `clientID` and `authority` values under `msalConfig`.

```
var msalConfig = {
  auth: {
    clientId: "Enter_the_Application_Id_here",
    authority: "https://login.microsoftonline.com/Enter_the_Tenant_info_here",
    redirectURI: "http://localhost:30662/"
  },
  cache: {
    cacheLocation: "localStorage",
    storeAuthStateInCookie: true
  }
};
```

NOTE

This quickstart supports Enter_the_Supported_Account_Info_Here.

Where:

- <Enter_the_Application_Id_here> is the **Application (client) ID** for the application you registered.
- <Enter_the_Tenant_info_here> is set to one of the following options:
 - If your application supports *accounts in this organizational directory*, replace this value with the **Tenant ID** or **Tenant name** (for example, *contoso.microsoft.com*).
 - If your application supports *accounts in any organizational directory*, replace this value with **organizations**.
 - If your application supports *accounts in any organizational directory and personal Microsoft accounts*, replace this value with **common**. To restrict support to *personal Microsoft accounts only*, replace this value with **consumers**.

TIP

To find the values of **Application (client) ID**, **Directory (tenant) ID**, and **Supported account types**, go to the app's [Overview](#) page in the Azure portal.

Step 4: Run the project

- If you're using [Node.js](#):

1. To start the server, run the following command from the project directory:

```
npm install  
node server.js
```

2. Open a web browser and go to <http://localhost:30662/>.
 3. Select **Sign In** to start the sign-in, and then call Microsoft Graph API.
- If you're using [Visual Studio](#), select the project solution, and then select F5 to run the project.

After the browser loads the application, select **Sign In**. The first time that you sign in, you're prompted to provide your consent to allow the application to access your profile and to sign you in. After you're signed in successfully, your user profile information should be displayed on the page.

More information

How the sample works

msal.js

The MSAL library signs in users and requests the tokens that are used to access an API that's protected by Microsoft identity platform. The quickstart *index.html* file contains a reference to the library:

```
<script src="https://secure.aadcdn.microsoftonline-p.com/lib/1.0.0/js/msal.min.js"></script>
```

TIP

You can replace the preceding version with the latest released version under [MSAL.js releases](#).

Alternatively, if you have Node.js installed, you can download the latest version through Node.js Package Manager (npm):

```
npm install msal
```

MSAL initialization

The quickstart code also shows how to initialize the MSAL library:

```
var msalConfig = {  
    auth: {  
        clientId: "Enter_the_Application_Id_here",  
        authority: "https://login.microsoftonline.com/Enter_the_Tenant_Info_Here",  
        redirectURI: "http://localhost:30662/"  
    },  
    cache: {  
        cacheLocation: "localStorage",  
        storeAuthStateInCookie: true  
    }  
};  
  
var myMSALObj = new Msal.UserAgentApplication(msalConfig);
```

WHERE	
<code>clientId</code>	The application ID of the application that's registered in the Azure portal.
<code>authority</code>	(Optional) The authority URL that supports account types, as described previously in the configuration section. The default authority is <code>https://login.microsoftonline.com/common</code> .
<code>redirectURI</code>	The application registration's configured reply/redirect URI. In this case, <code>http://localhost:30662/</code> .
<code>cacheLocation</code>	(Optional) Sets the browser storage for the auth state. The default is sessionStorage.
<code>storeAuthStateInCookie</code>	(Optional) The library that stores the authentication request state that's required for validation of the authentication flows in the browser cookies. This cookie is set for IE and Edge browsers to mitigate certain known issues .

For more information about available configurable options, see [Initialize client applications](#).

Sign in users

The following code snippet shows how to sign in users:

```
var requestObj = {
    scopes: ["user.read"]
};

myMSALObj.loginPopup(requestObj).then(function (loginResponse) {
    //Login Success callback code here
}).catch(function (error) {
    console.log(error);
});
```

WHERE	
<code>scopes</code>	(Optional) Contains scopes that are being requested for user consent at sign-in time. For example, <code>["user.read"]</code> for Microsoft Graph or <code>["<Application ID URL>/scope"]</code> for custom Web APIs (that is, <code>api://<Application ID>/access_as_user</code>).

TIP

Alternatively, you might want to use the `loginRedirect` method to redirect the current page to the sign-in page instead of a popup window.

Request tokens

MSAL uses three methods to acquire tokens: `acquireTokenRedirect`, `acquireTokenPopup`, and `acquireTokenSilent`

Get a user token silently

The `acquireTokenSilent` method handles token acquisitions and renewal without any user interaction. After the `loginRedirect` or `loginPopup` method is executed for the first time, `acquireTokenSilent` is the method commonly used to obtain tokens that are used to access protected resources for subsequent calls. Calls to request or renew tokens are made silently.

```
var requestObj = {
  scopes: ["user.read"]
};

myMSALObj.acquireTokenSilent(requestObj).then(function (tokenResponse) {
  // Callback code here
  console.log(tokenResponse.accessToken);
}).catch(function (error) {
  console.log(error);
});
```

WHERE

`scopes`

Contains scopes being requested to be returned in the access token for API. For example, ["user.read"] for Microsoft Graph or ["<Application ID URL>/scope"] for custom Web APIs (that is, `api://<Application ID>/access_as_user`).

Get a user token interactively

There are situations where you need to force users to interact with the Microsoft identity platform endpoint. For example:

- Users might need to reenter their credentials because their password has expired.
- Your application is requesting access to additional resource scopes that the user needs to consent to.
- Two-factor authentication is required.

The usual recommended pattern for most applications is to call `acquireTokenSilent` first, then catch the exception, and then call `acquireTokenPopup` (or `acquireTokenRedirect`) to start an interactive request.

Calling the `acquireTokenPopup` results in a popup window for signing in. (Or `acquireTokenRedirect` results in redirecting users to the Microsoft identity platform endpoint.) In that window, users need to interact by confirming their credentials, giving the consent to the required resource, or completing the two-factor authentication.

```
var requestObj = {
  scopes: ["user.read"]
};

myMSALObj.acquireTokenPopup(requestObj).then(function (tokenResponse) {
  // Callback code here
  console.log(tokenResponse.accessToken);
}).catch(function (error) {
  console.log(error);
});
```

NOTE

This quickstart uses the `loginRedirect` and `acquireTokenRedirect` methods with Microsoft Internet Explorer, because of a [known issue](#) related to the handling of popup windows by Internet Explorer.

Next steps

For a more detailed step-by-step guide on building the application for this quickstart, see:

[Tutorial to sign in and call MS Graph](#)

To browse the MSAL repo for documentation, FAQ, issues, and more, see:

[MSAL.js GitHub repo](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

Quickstart: Add Microsoft identity platform sign-in to an ASP.NET web app

10/30/2019 • 6 minutes to read • [Edit Online](#)

Applies to:

- Microsoft identity platform endpoint

In this quickstart, you'll enable an ASP.NET web app to sign in personal accounts (hotmail.com, outlook.com, others) and work and school accounts from any Azure Active Directory (Azure AD) instance.

Register and download your quickstart app

You have two options to start your quickstart application:

- [Express] [Option 1: Register and auto configure your app and then download your code sample](#)
- [Manual] [Option 2: Register and manually configure your application and code sample](#)

Option 1: Register and auto configure your app and then download your code sample

- Go to the new [Azure portal - App registrations](#) pane.
- Enter a name for your application and click **Register**.
- Follow the instructions to download and automatically configure your new application for you in one click.

Option 2: Register and manually configure your application and code sample

Step 1: Register your application

To register your application and add the app's registration information to your solution manually, follow these steps:

- Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account.
- If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
- Navigate to the Microsoft identity platform for developers [App registrations](#) page.
- Select **New registration**.
- When the **Register an application** page appears, enter your application's registration information:
 - In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `ASPNET-Quickstart`.
 - Add `http://localhost:44368/` in **Redirect URI**, and click **Register**.
 - From the left navigation pane under the Manage section, select **Authentication**
 - Under the **Implicit Grant** sub-section, select **ID tokens**.
 - And then select **Save**.

Step 2: Download your project

[Download the Visual Studio 2019 solution](#)

Step 3: Configure your Visual Studio project

1. Extract the zip file to a local folder closer to the root folder - for example, **C:\Azure-Samples**
2. Open the solution in Visual Studio (AppModelv2-WebApp-OpenIDConnect-DotNet.sln)
3. Depending on the version of Visual Studio, you might need to right click on the project **AppModelv2-WebApp-OpenIDConnect-DotNet** and **Restore NuGet packages**
4. Open the Package Manager Console (View -> Other Windows -> Package Manager Console) and run
`Update-Package Microsoft.CodeDom.Providers.DotNetCompilerPlatform -r`
5. Edit **Web.config** and replace the parameters **ClientId** and **Tenant** with:

```
<add key="ClientId" value="Enter_the_Application_Id_here" />
<add key="Tenant" value="Enter_the_Tenant_Info_Here" />
```

Where:

- **Enter_the_Application_Id_here** - is the Application Id for the application you registered.
- **Enter_the_Tenant_Info_Here** - is one of the options below:
 - If your application supports **My organization only**, replace this value with the **Tenant Id** or **Tenant name** (for example, contoso.onmicrosoft.com)
 - If your application supports **Accounts in any organizational directory**, replace this value with **organizations**
 - If your application supports **All Microsoft account users**, replace this value with **common**

TIP

- To find the values of *Application ID*, *Directory (tenant) ID*, and *Supported account types*, go to the **Overview** page
- Ensure the value for **redirectUri** in the **Web.config** corresponds with the **Redirect URI** defined for the App Registration in Azure AD (if not, navigate to the **Authentication** menu for the App Registration and update the **REDIRECT URI** to match)

More information

This section gives an overview of the code required to sign-in users. This overview can be useful to understand how the code works, main arguments, and also if you want to add sign-in to an existing ASP.NET application.

OWIN middleware NuGet packages

You can set up the authentication pipeline with cookie-based authentication using OpenID Connect in ASP.NET with OWIN Middleware packages. You can install these packages by running the following commands in Visual Studio's **Package Manager Console**:

```
Install-Package Microsoft.Owin.Security.OpenIdConnect
Install-Package Microsoft.Owin.Security.Cookies
Install-Package Microsoft.Owin.Host.SystemWeb
```

OWIN Startup Class

The OWIN middleware uses a *startup class* that runs when the hosting process initializes. In this quickstart, the *startup.cs* file located in root folder. The following code shows the parameter used by this quickstart:

```

public void Configuration(IAppBuilder app)
{
    app.SetDefaultSignInAsAuthenticationType(CookieAuthenticationDefaults.AuthenticationType);

    app.UseCookieAuthentication(new CookieAuthenticationOptions());
    app.UseOpenIdConnectAuthentication(
        new OpenIdConnectAuthenticationOptions
        {
            // Sets the ClientId, authority, RedirectUri as obtained from web.config
            ClientId = clientId,
            Authority = authority,
            RedirectUri = redirectUri,
            // PostLogoutRedirectUri is the page that users will be redirected to after sign-out. In this
            case, it is using the home page
            PostLogoutRedirectUri = redirectUri,
            Scope = OpenIdConnectScope.OpenIdProfile,
            // ResponseType is set to request the id_token - which contains basic information about the
            signed-in user
            ResponseType = OpenIdConnectResponseType.IdToken,
            // ValidateIssuer set to false to allow personal and work accounts from any organization to sign
            in to your application
            // To only allow users from a single organizations, set ValidateIssuer to true and 'tenant'
            setting in web.config to the tenant name
            // To allow users from only a list of specific organizations, set ValidateIssuer to true and use
            ValidIssuers parameter
            TokenValidationParameters = new TokenValidationParameters()
            {
                ValidateIssuer = false // Simplification (see note below)
            },
            // OpenIdConnectAuthenticationNotifications configures OWIN to send notification of failed
            authentications to OnAuthenticationFailed method
            Notifications = new OpenIdConnectAuthenticationNotifications()
            {
                AuthenticationFailed = OnAuthenticationFailed
            }
        }
    );
}

```

WHERE	
<code>ClientId</code>	Application ID from the application registered in the Azure portal
<code>Authority</code>	The STS endpoint for user to authenticate. Usually https://login.microsoftonline.com/{tenant}/v2.0 for public cloud, where {tenant} is the name of your tenant, your tenant Id, or <i>common</i> for a reference to the common endpoint (used for multi-tenant applications)
<code>RedirectUri</code>	URL where users are sent after authentication against Microsoft identity platform endpoint
<code>PostLogoutRedirectUri</code>	URL where users are sent after signing-off
<code>Scope</code>	The list of scopes being requested, separated by spaces
<code>ResponseType</code>	Request that the response from authentication contains an ID token

WHERE

TokenValidationParameters	A list of parameters for token validation. In this case, <code>ValidateIssuer</code> is set to <code>false</code> to indicate that it can accept sign-ins from any personal, or work or school account types
Notifications	A list of delegates that can be executed on different <code>OpenIdConnect</code> messages

NOTE

Setting `ValidateIssuer = false` is a simplification for this quickstart. In real applications you need to validate the issuer. See the samples to understand how to do that.

Initiate an authentication challenge

You can force a user to sign in by requesting an authentication challenge in your controller:

```
public void SignIn()
{
    if (!Request.IsAuthenticated)
    {
        HttpContext.GetOwinContext().Authentication.Challenge(
            new AuthenticationProperties{ RedirectUri = "/" },
            OpenIdConnectAuthenticationDefaults.AuthenticationType);
    }
}
```

TIP

Requesting an authentication challenge using the method above is optional and normally used when you want a view to be accessible from both authenticated and non-authenticated users. Alternatively, you can protect controllers by using the method described in the next section.

Protect a controller or a controller's method

You can protect a controller or controller actions using the `[Authorize]` attribute. This attribute restricts access to the controller or actions by allowing only authenticated users to access the actions in the controller, which means that authentication challenge will happen automatically when a *non-authenticated* user tries to access one of the actions or controller decorated by the `[Authorize]` attribute.

Next steps

Try out the ASP.NET tutorial for a complete step-by-step guide on building applications and new features, including a full explanation of this quickstart.

Learn the steps to create the application used in this quickstart

[Sign-in tutorial](#)

Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

Help and support for developers

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

Quickstart: Add sign-in with Microsoft to an ASP.NET Core web app

8/7/2019 • 5 minutes to read • [Edit Online](#)

Applies to:

- Microsoft identity platform endpoint

In this quickstart, you'll learn how an ASP.NET Core web app can sign in personal accounts (hotmail.com, outlook.com, others) and work and school accounts from any Azure Active Directory (Azure AD) instance.

Register and download your quickstart app

You have two options to start your quickstart application:

- [Express] [Option 1: Register and auto configure your app and then download your code sample](#)
- [Manual] [Option 2: Register and manually configure your application and code sample](#)

Option 1: Register and auto configure your app and then download your code sample

- Go to the [Azure portal - App registrations](#).
- Enter a name for your application and select **Register**.
- Follow the instructions to download and automatically configure your new application for you in one click.

Option 2: Register and manually configure your application and code sample

Step 1: Register your application

To register your application and manually add the app's registration information to your solution, follow these steps:

- Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account.
- If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
- Navigate to the Microsoft identity platform for developers [App registrations](#) page.
- Select **New registration**.
- When the **Register an application** page appears, enter your application's registration information:
 - In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `AspNetCore-Quickstart`.
 - In **Redirect URI**, add `https://localhost:44321/`, and select **Register**.
- Select the **Authentication** menu, and then add the following information:
 - In **Redirect URIs**, add `https://localhost:44321/signin-oidc`, and select **Save**.
 - In the **Advanced settings** section, set **Logout URL** to `https://localhost:44321/signout-oidc`.
 - Under **Implicit grant**, check **ID tokens**.
 - Select **Save**.

Step 2: Download your ASP.NET Core project

- Download the Visual Studio 2019 solution

Step 3: Configure your Visual Studio project

1. Extract the zip file to a local folder within the root folder - for example, **C:\Azure-Samples**
2. If you use Visual Studio 2019, open the solution in Visual Studio (optional).
3. Edit the **appsettings.json** file. Find `ClientId` and update the value of `ClientId` with the **Application (client) ID** value of the application you registered.

```
"ClientId": "Enter_the_Application_Id_here"  
"TenantId": "Enter_the_Tenant_Info_Here"
```

Where:

- `Enter_the_Application_Id_here` - is the **Application (client) ID** for the application you registered in the Azure portal. You can find **Application (client) ID** in the app's **Overview** page.
- `Enter_the_Tenant_Info_Here` - is one of the following options:
 - If your application supports **Accounts in this organizational directory only**, replace this value with the **Tenant ID** or **Tenant name** (for example, contoso.microsoft.com)
 - If your application supports **Accounts in any organizational directory**, replace this value with `organizations`
 - If your application supports **All Microsoft account users**, replace this value with `common`

TIP

To find the values of **Application (client) ID**, **Directory (tenant) ID**, and **Supported account types**, go to the app's **Overview** page in the Azure portal.

More information

This section gives an overview of the code required to sign in users. This overview can be useful to understand how the code works, main arguments, and also if you want to add sign-in to an existing ASP.NET Core application.

Startup class

Microsoft.AspNetCore.Authentication middleware uses a Startup class that is executed when the hosting process initializes:

```

public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddAuthentication(AzureADDefaults.AuthenticationScheme)
        .AddAzureAD(options => Configuration.Bind("AzureAd", options));

    services.Configure<OpenIdConnectOptions>(AzureADDefaults.OpenIdScheme, options =>
    {
        options.Authority = options.Authority + "/v2.0/";           // Microsoft identity platform

        options.TokenValidationParameters.ValidateIssuer = false; // accept several tenants (here simplified)
    });

    services.AddMvc(options =>
    {
        var policy = new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .Build();
        options.Filters.Add(new AuthorizeFilter(policy));
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}

```

The method `AddAuthentication` configures the service to add cookie-based authentication, which is used on browser scenarios and to set the challenge to OpenID Connect.

The line containing `.AddAzureAd` adds the Microsoft identity platform authentication to your application. It's then configured to sign in using the Microsoft identity platform endpoint.

WHERE	
ClientId	Application (client) ID from the application registered in the Azure portal.
Authority	The STS endpoint for the user to authenticate. Usually, this is https://login.microsoftonline.com/{tenant}/v2.0 for public cloud, where {tenant} is the name of your tenant or your tenant ID, or <i>common</i> for a reference to the common endpoint (used for multi-tenant applications)
TokenValidationParameters	A list of parameters for token validation. In this case, <code>ValidateIssuer</code> is set to <code>false</code> to indicate that it can accept sign-ins from any personal, or work or school accounts.

NOTE

Setting `ValidateIssuer = false` is a simplification for this quickstart. In real applications you need to validate the issuer. See the samples to understand how to do that.

Protect a controller or a controller's method

You can protect a controller or controller methods using the `[Authorize]` attribute. This attribute restricts access to

the controller or methods by only allowing authenticated users, which means that authentication challenge can be started to access the controller if the user isn't authenticated.

Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

Next steps

Check out the GitHub repo for this ASP.NET Core tutorial for more information including instructions on how to add authentication to a brand new ASP.NET Core Web application, how to call Microsoft Graph, and other Microsoft APIs, how to call your own APIs, how to add authorization, how to sign in users in national clouds, or with social identities and more:

[ASP.NET Core Web App tutorial](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

Quickstart: Add sign-in using OIDC to a Node.js Web App

10/30/2019 • 3 minutes to read • [Edit Online](#)

Applies to:

- Microsoft identity platform endpoint

In this quickstart, you'll learn how to set up OpenID Connect authentication in a web application built using Node.js with Express. The sample is designed to run on any platform.

Prerequisites

To run this sample, you will need:

- Install Node.js from <http://nodejs.org/>
- Either a [Microsoft account](#) or [Office 365 Developer Program](#)

Register your application

- Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account.
- If your account is present in more than one Azure AD tenant:
 - Select your profile from the menu on the top-right corner of the page, and then **Switch directory**.
 - Change your session to the Azure AD tenant where you want to create your application.
- Navigate to [Azure Active Directory > App registrations](#) to register your app.
- Select **New registration**.
- When the **Register an application** page appears, enter your app's registration information:
 - In the **Name** section, enter a meaningful name that will be displayed to users of the app. For example: MyWebApp
 - In the **Supported account types** section, select **Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)**.

If there are more than one redirect URIs, you'll need to add these from the **Authentication** tab later after the app has been successfully created.

- Select **Register** to create the app.
- On the app's **Overview** page, find the **Application (client) ID** value and record it for later. You'll need this value to configure the application later in this project.
- In the list of pages for the app, select **Authentication**.
 - In the **Redirect URIs** section, select **Web** in the combo-box and enter the following redirect URI:
`http://localhost:3000/auth/openid/return`
 - In the **Advanced settings** section, set **Logout URL** to `http://localhost:3000`.

- In the **Advanced settings > Implicit grant** section, check **ID tokens** as this sample requires the [Implicit grant flow](#) to be enabled to sign-in the user.
9. Select **Save**.
10. From the **Certificates & secrets** page, in the **Client secrets** section, choose **New client secret**.
- Enter a key description (for instance app secret).
 - Select a key duration of either **In 1 year**, **In 2 years**, or **Never Expires**.
 - When you click the **Add** button, the key value will be displayed. Copy the key value and save it in a safe location.
- You'll need this key later to configure the application. This key value will not be displayed again, nor retrievable by any other means, so record it as soon as it is visible from the Azure portal.
- ## Download the sample application and modules
- Next, clone the sample repo and install the NPM modules.
- From your shell or command line:
- ```
$ git clone git@github.com:AzureADQuickStarts/AppModelv2-WebApp-OpenIDConnect-nodejs.git
```
- or
- ```
$ git clone https://github.com/AzureADQuickStarts/AppModelv2-WebApp-OpenIDConnect-nodejs.git
```
- From the project root directory, run the command:
- ```
$ npm install
```
- ## Configure the application
- Provide the parameters in `exports.creds` in config.js as instructed.
- Update `<tenant_name>` in `exports.identityMetadata` with the Azure AD tenant name of the format \*.onmicrosoft.com.
  - Update `exports.clientID` with the Application ID noted from app registration.
  - Update `exports.clientSecret` with the Application secret noted from app registration.
  - Update `exports.redirectURL` with the Redirect URI noted from app registration.
- Optional configuration for production apps:**
- Update `exports.destroySessionUrl` in config.js, if you want to use a different `post_logout_redirect_uri`.
  - Set `exports.useMongoDBSessionStore` in config.js to true, if you want to use [mongoDB](#) or other [compatible session stores](#). The default session store in this sample is `express-session`. The default session store is not suitable for production.
  - Update `exports.databaseUri`, if you want to use mongoDB session store and a different database URI.
  - Update `exports.mongoDBSessionMaxAge`. Here you can specify how long you want to keep a session in mongoDB. The unit is second(s).
- ## Build and run the application
- Start mongoDB service. If you are using mongoDB session store in this app, you have to [install mongoDB](#) and start the service first. If you are using the default session store, you can skip this step.
- Run the app using the following command from your command line.

```
$ node app.js
```

**Is the server output hard to understand?**: We use `bunyan` for logging in this sample. The console won't make much sense to you unless you also install bunyan and run the server like above but pipe it through the bunyan binary:

```
$ npm install -g bunyan
$ node app.js | bunyan
```

## You're done!

You will have a server successfully running on `http://localhost:3000`.

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

## Next steps

Learn more about the web app scenario that the Microsoft identity platform supports:

[Web app that signs in users scenario](#)

# Quickstart: Add sign-in with Microsoft to a Java web app

11/8/2019 • 5 minutes to read • [Edit Online](#)

## Applies to:

- Microsoft identity platform endpoint

In this quickstart, you'll learn how to integrate a Java web application with the Microsoft identity platform. Your app will sign in a user, get an access token to call the Microsoft Graph API, and make a request to the Microsoft Graph API.

When you've completed this quickstart, your application will accept sign-ins of personal Microsoft accounts (including outlook.com, live.com, and others) and work or school accounts from any company or organization that uses Azure Active Directory.

## Prerequisites

To run this sample you will need:

- Java Development Kit (JDK) 8 or greater, and [Maven](#).
- An Azure Active Directory (Azure AD) tenant. For more information on how to get an Azure AD tenant, see [How to get an Azure AD tenant](#).

## Register and download your quickstart app

You have two options to start your quickstart application: express (Option 1), or manual (Option 2)

### Option 1: Register and auto configure your app and then download your code sample

- Go to the [Azure portal - App registrations](#).
- Enter a name for your application and select **Register**.
- Follow the instructions to download and automatically configure your new application.

### Option 2: Register and manually configure your application and code sample

#### Step 1: Register your application

To register your application and manually add the app's registration information to your solution, follow these steps:

- Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account.
- If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
- Navigate to the Microsoft identity platform for developers [App registrations](#) page.
- Select **New registration**.
- When the **Register an application** page appears, enter your application's registration information:

- In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `java-webapp`.
  - Leave **Redirect URI** blank for now, and select **Register**.
6. On the **Overview** page, find the **Application (client) ID** and the **Directory (tenant) ID** values of the application. Copy these values for later.
7. Select the **Authentication** from the menu, and then add the following information:

- In **Redirect URIs**, add `http://localhost:8080/msal4jsamples/secure/aad` and `http://localhost:8080/msal4jsamples/graph/me`.
- Select **Save**.

8. Select the **Certificates & secrets** from the menu and in the **Client secrets** section, click on **New client secret**:

- Type a key description (for instance app secret).
- Select a key duration **In 1 year**.
- The key value will display when you select **Add**.
- Copy the value of the key for later. This key value will not be displayed again, nor retrievable by any other means, so record it as soon as it is visible from the Azure portal.

#### Step 2: Download the code sample

[Download the Code Sample](#)

#### Step 3: Configure the code sample

1. Extract the zip file to a local folder.
2. If you use an integrated development environment, open the sample in your favorite IDE (optional).
3. Open the application.properties file, which can be found in src/main/resources/ folder and replace the value of the fields `aad.clientId`, `aad.authority` and `aad.secretKey` with the respective values of **Application Id**, **Tenant Id** and **Client Secret** as the following:

```
aad.clientId=Enter_the_Application_Id_here
aad.authority=https://login.microsoftonline.com/Enter_the_Tenant_Info_Here/
aad.secretKey=Enter_the_Client_Secret_Here
aad.redirectUriSignIn=http://localhost:8080/msal4jsample/secure/aad
aad.redirectUriGraph=http://localhost:8080/msal4jsample/graph/me
```

Where:

- `Enter_the_Application_Id_here` - is the Application Id for the application you registered.
- `Enter_the_Client_Secret_Here` - is the **Client Secret** you created in **Certificates & Secrets** for the application you registered.
- `Enter_the_Tenant_Info_Here` - is the **Directory (tenant) ID** value of the application you registered.

#### Step 4: Run the code sample

To run the project, you can either:

Run it directly from your IDE by using the embedded spring boot server or package it to a WAR file using [maven](#) and deploy it to a J2EE container solution such as [Apache Tomcat](#).

##### Running from IDE

If you are running the web application from an IDE, click on run, then navigate to the home page of the project. For this sample, the standard home page URL is `http://localhost:8080`

1. On the front page, select the **Login** button to redirect to Azure Active Directory and prompt the user for their credentials.

2. After the user is authenticated, they are redirected to <http://localhost:8080/msal4jsamples/secure/aad>. They are now signed in, and the page will show information about the signed-in account. The sample UI has the following buttons:

- *Sign Out*: Signs the current user out of the application and redirects them to the home page.
- *Show User Info*: Acquires a token for Microsoft Graph and calls Microsoft Graph with a request containing the token, which returns basic information about the signed-in user.

#### IMPORTANT

This quickstart application uses a client secret to identify itself as confidential client. Because the client secret is added as a plain-text to your project files, for security reasons it is recommended that you use a certificate instead of a client secret before considering the application as production application. For more information on how to use a certificate, see [Certificate credentials for application authentication](#).

## More information

### Getting MSAL

MSAL4J is the Java library used to sign in users and request tokens used to access an API protected by the Microsoft identity Platform.

Add MSAL4J to your application by using Maven or Gradle to manage your dependencies by making the following changes to the application's pom.xml (Maven) or build.gradle (Gradle) file.

```
<dependency>
 <groupId>com.microsoft.azure</groupId>
 <artifactId>msal4j</artifactId>
 <version>1.0.0</version>
</dependency>
```

```
compile group: 'com.microsoft.azure', name: 'msal4j', version: '1.0.0'
```

### MSAL initialization

Add a reference to MSAL4J by adding the following code to the top of the file where you will be using MSAL4J:

```
import com.microsoft.aad.msal4j.*;
```

## Next Steps

Learn more about permissions and consent:

### Permissions and Consent

To know more about the auth flow for this scenario, see the Oauth 2.0 authorization code flow:

### Authorization Code Oauth flow

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

### Microsoft identity platform survey

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

# Quickstart: Add sign-in with Microsoft to a Python web app

11/7/2019 • 4 minutes to read • [Edit Online](#)

## Applies to:

- Microsoft identity platform endpoint

In this quickstart, you'll learn how to integrate a Python web application with the Microsoft identity platform. Your app will sign in a user, get an access token to call the Microsoft Graph API, and make a request to the Microsoft Graph API.

When you've completed the guide, your application will accept sign-ins of personal Microsoft accounts (including outlook.com, live.com, and others) and work or school accounts from any company or organization that uses Azure Active Directory.

## Prerequisites

To run this sample, you will need:

- [Python 2.7+](#) or [Python 3+](#)
- [Flask](#), [Flask-Session](#), [requests](#)
- [MSAL Python](#)

## Register and download your quickstart app

You have two options to start your quickstart application: express (Option 1), and manual (Option 2)

### Option 1: Register and auto configure your app and then download your code sample

- Go to the [Azure portal - App registrations](#).
- Select **New registration**.
- Enter a name for your application and select **Register**.
- Follow the instructions to download and automatically configure your new application.

### Option 2: Register and manually configure your application and code sample

#### Step 1: Register your application

To register your application and add the app's registration information to your solution manually, follow these steps:

- Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account.
- If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
- Navigate to the Microsoft identity platform for developers [App registrations](#) page.
- Select **New registration**.

- When the **Register an application** page appears, enter your application's registration information:
  - In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `python-webapp`.
  - Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
  - Under the **Redirect URI** section, in the drop-down list, select the **Web** platform, and then set the value to `http://localhost:5000/getAToken`.
  - Select **Register**. On the app **Overview** page, note the **Application (client) ID** value for later use.

- On the left hand menu, choose **Certificates & secrets** and click on **New client secret** in the **Client Secrets** section:

- Type a key description (of instance app secret).
- Select a key duration of **In 1 year**.
- When you click on **Add**, the key value will be displayed.
- Copy the value of the key. You will need it later.

- Select the **API permissions** section

- Click the **Add a permission** button and then,
- Ensure that the **Microsoft APIs** tab is selected
- In the *Commonly used Microsoft APIs* section, click on **Microsoft Graph**
- In the **Delegated permissions** section, ensure that the right permissions are checked:  
**User.ReadBasic.All**. Use the search box if necessary.
- Select the **Add permissions** button

#### **Step 2: Download your project**

[Download the Code Sample](#)

#### **Step 3: Configure the Application**

- Extract the zip file to a local folder closer to the root folder - for example, **C:\Azure-Samples**
- If you use an integrated development environment, open the sample in your favorite IDE (optional).
- Open the **app\_config.py** file, which can be found in the root folder and replace with the following code snippet:

```
CLIENT_ID = "Enter_the_Application_Id_here"
CLIENT_SECRET = "Enter_the_Client_Secret_Here"
AUTHORITY = "https://login.microsoftonline.com/Enter_the_Tenant_Name_Here"
```

Where:

- `Enter_the_Application_Id_here` - is the Application Id for the application you registered.
- `Enter_the_Client_Secret_Here` - is the **Client Secret** you created in **Certificates & Secrets** for the application you registered.
- `Enter_the_Tenant_Name_Here` - is the **Directory (tenant) ID** value of the application you registered.

#### **Step 4: Run the code sample**

- You will need to install MSAL Python library, Flask framework, Flask-Sessions for server-side session management and requests using pip as follows:

```
pip install -r requirements.txt
```

- Run `app.py` from shell or command line:

```
python app.py
```

### IMPORTANT

This quickstart application uses a client secret to identify itself as confidential client. Because the client secret is added as a plain-text to your project files, for security reasons, it is recommended that you use a certificate instead of a client secret before considering the application as production application. For more information on how to use a certificate, see [these instructions](#).

## More information

### Getting MSAL

MSAL is the library used to sign in users and request tokens used to access an API protected by the Microsoft identity Platform. You can add MSAL Python to your application using Pip.

```
pip install msal
```

### MSAL initialization

You can add the reference to MSAL Python by adding the following code to the top of the file where you will be using MSAL:

```
import msal
```

## Next steps

Learn more about web apps that sign in users, and then that calls web APIs:

[Scenario: Web apps that sign in users](#)

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

# Quickstart: Call an ASP.NET Web API protected by Azure AD

10/31/2019 • 6 minutes to read • [Edit Online](#)

In this quickstart, you expose a Web API and protect it so that only authenticated user can access it. This sample shows how to expose a ASP.NET Web API so it can accept tokens issued by personal accounts (including outlook.com, live.com, and others) as well as work and school accounts from any company or organization that has integrated with Azure Active Directory.

The sample also includes a Windows Desktop application (WPF) client that demonstrates how you can request an access token to access a Web API.

## Prerequisites

To run this sample, you will need the following:

- Visual Studio 2017 or 2019. Download [Visual Studio for free](#).
- Either a [Microsoft account](#) or [Office 365 Developer Program](#)

## Download or clone this sample

You can clone this sample from your shell or command line:

```
git clone https://github.com/AzureADQuickStarts/AppModelv2-NativeClient-DotNet.git
```

Or, you can [download the sample as a ZIP file](#).

## Register your Web API in the application registration portal

### Choose the Azure AD tenant where you want to create your applications

If you want to register your apps manually, as a first step you'll need to:

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account is present in more than one Azure AD tenant, select your profile at the top-right corner in the menu on top of the page, and then **switch directory**. Change your portal session to the desired Azure AD tenant.

### Register the service app (`TodoListService`)

1. Navigate to the Microsoft identity platform for developers [App registrations](#) page.
2. Select **New registration**.
3. When the **Register an application** page appears, enter your application's registration information:
  - In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `AppModelv2-NativeClient-DotNet-TodoListService`.
  - Change **Supported account types** to **Accounts in any organizational directory**.
  - Select **Register** to create the application.
4. On the app **Overview** page, find the **Application (client) ID** value and record it for later. You'll need it to

configure the Visual Studio configuration file for this project ( `clientId` in `TodoListService\Web.config` ).

5. Select the **Expose an API** section, and:

- Select **Add a scope**
- accept the proposed Application ID URI (`api://{clientId}`) by selecting **Save and Continue**
- Enter the following parameters:
  - for **Scope name** use `access_as_user`
  - Ensure the **Admins and users** option is selected for **Who can consent**
  - in **Admin consent display name** type `Access TodoListService as a user`
  - in **Admin consent description** type `Accesses the TodoListService Web API as a user`
  - in **User consent display name** type `Access TodoListService as a user`
  - in **User consent description** type `Accesses the TodoListService Web API as a user`
  - Keep **State** as **Enabled**
  - Select **Add scope**

### Configure the service and client projects to match the registered Web API

1. Open the solution in Visual Studio and then open the **Web.config** file under the root of **TodoListService** project.
2. Replace the value of `ida:ClientId` parameter with the **Client ID (Application ID)** from the application you just registered in the Application Registration Portal.

### Add the new scope to the *TodoListClient*'s app.config

1. Open the **app.config** file located in **TodoListClient** project's root folder and then paste **Application ID** from the application you just registered for your *TodoListService* under `TodoListServiceScope` parameter, replacing the string `{Enter the Application ID of your TodoListService from the app registration portal}`.

Note: Make sure it uses the following format:

```
api://{TodoListService-Application-ID}/access_as_user
```

(where `{TodoListService-Application-ID}` is the GUID representing the Application ID for your *TodoListService*).

## Register the client app (*TodoListClient*)

In this step, you configure your *TodoListClient* project by registering a new application in the Application registration portal. In the cases where the client and server are considered *the same application* you may also just reuse the same application registered in the 'Step 2.'. Using the same application is needed if you want users to sign in with Microsoft personal accounts

### Register the *TodoListClient* application in the *Application registration portal*

1. Navigate to the Microsoft identity platform for developers [App registrations](#) page.
2. Select **New registration**.
3. When the **Register an application** page appears, enter your application's registration information:
  - In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `NativeClient-DotNet-TodoListClient`.
  - Change **Supported account types** to **Accounts in any organizational directory**.
  - Select **Register** to create the application.
4. From the app's Overview page, select the **Authentication** section.
  - In the **Redirect URLs | Suggested Redirect URLs for public clients (mobile, desktop)** section, check `urn:ietf:wg:oauth:2.0:oob`

- Select **Save**.
5. Select the **API permissions** section
- Click the **Add a permission** button and then,
  - Select the **My APIs** tab.
  - In the list of APIs, select the `AppModelv2-NativeClient-DotNet-TodoListService API`, or the name you entered for the Web API.
  - Check the **access\_as\_user** permission if it's not already checked. Use the search box if necessary.
  - Select the **Add permissions** button

### Configure your *TodoListClient* project

1. In the *Application registration portal*, in the **Overview** page copy the value of the **Application (client) ID**
2. Open the **app.config** file located in the **TodoListClient** project's root folder and then paste the value in the `ida:ClientId` parameter value

## Run your project

1. Press `<F5>` to run your project. Your *TodoListClient* should open.
2. Select **Sign in** at the top right and sign in with the same user you have used to register your application, or a user in the same directory.
3. At this point, if you are signing in for the first time, you may be prompted to consent to *TodoListService* Web Api.
4. The sign-in also request the access token to the `access_as_user` scope to access *TodoListService* Web Api and manipulate the To-Do list.

## Pre-authorize your client application

One of the ways to allow users from other directories to access your Web API is by *pre-authorizing* the client applications to access your Web API by adding the Application IDs from client applications in the list of *pre-authorized* applications for your Web API. By adding a pre-authorized client, you will not require user to consent to use your Web API. Follow the steps below to pre-authorize your Web Application:

1. Go back to the *Application registration portal* and open the properties of your **TodoListService**.
2. In the **Expose an API** section, click on **Add a client application** under the *Authorized client applications* section.
3. In the *Client ID* field, paste the application ID of the `TodoListClient` application.
4. In the *Authorized scopes* section, select the scope for this Web API `api://<Application ID>/access_as_user`.
5. Press the **Add application** button at the bottom of the page.

## Run your project

1. Press `<F5>` to run your project. Your *TodoListClient* should open.
2. Select **Sign in** at the top right (or Clear Cache/Sign-in) and then sign-in either using a personal Microsoft account (live.com or hotmail.com) or work or school account.

## Optional: Restrict sign-in access to your application

By default, when you download this code sample and configure the application to use the Azure Active Directory v2 endpoint following the preceding steps, both personal accounts - like outlook.com, live.com, and others - as well as Work or school accounts from any organizations that are integrated with Azure AD can request tokens and access your Web API.

To restrict who can sign in to your application, use one of the options:

## Option 1: Restrict access to a single organization (single-tenant)

You can restrict sign-in access for your application to only user accounts that are in a single Azure AD tenant - including *guest accounts* of that tenant. This scenario is a common for *line-of-business applications*:

1. Open the **App\_Start\Startup.Auth** file, and change the value of the metadata endpoint that's passed into the `OpenIdConnectSecurityTokenProvider` to  
`"https://login.microsoftonline.com/{Tenant ID}/v2.0/.well-known/openid-configuration"` (you can also use the Tenant Name, such as `contoso.onmicrosoft.com` ).
2. In the same file, set the `ValidIssuer` property on the `TokenValidationParameters` to  
`"https://sts.windows.net/{Tenant ID}/"` and the `ValidateIssuer` argument to `true`.

## Option 2: Use a custom method to validate issuers

You can implement a custom method to validate issuers by using the **IssuerValidator** parameter. For more information about how to use this parameter, read about the [TokenValidationParameters class](#).

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

## Next steps

Learn more about the protected web API scenario that the Microsoft identity platform supports:

[Protected web API scenario](#)

# Quickstart: Sign in users and call the Microsoft Graph API from an Android app

11/12/2019 • 13 minutes to read • [Edit Online](#)

This quickstart uses a code sample to demonstrate how an Android application can sign in personal, work, or school accounts using the Microsoft identity platform, and then get an access token and call the Microsoft Graph API.

Applications must be represented by an app object in Azure Active Directory so that the Microsoft identity platform can share tokens with your application.

As a convenience, the code sample comes with a default `redirect_uri` preconfigured in the `AndroidManifest.xml` file so that you don't have to first register your own app object. A `redirect_uri` is partly based on your app's signing key. The sample project is preconfigured with a signing key so that the provided `redirect_uri` will work. To learn more about registering an app object and integrating it with your application, see the [Sign in users and call the Microsoft Graph from an Android app](#) tutorial.

## NOTE

### Prerequisites

- Android Studio
- Android 16+

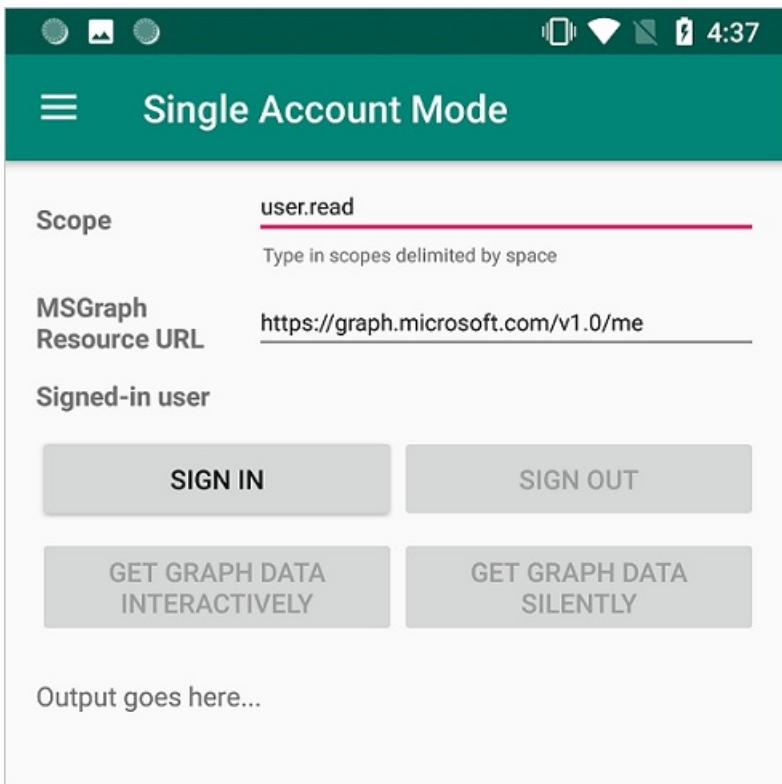
## Step 1: Get the sample app

[Download the code.](#)

## Step 2: Run the sample app

Select your emulator, or physical device, from Android Studio's **available devices** dropdown and run the app.

The sample app starts on the **Single Account Mode** screen. A default scope, `user.read`, is provided by default, which is used when reading your own profile data during the Microsoft Graph API call. The URL for the Microsoft Graph API call is provided by default. You can change both of these if you wish.



Use the app menu to change between single and multiple account modes.

In single account mode, sign in using a work or home account:

1. Select **Get graph data interactively** to prompt the user for their credentials. You'll see the output from the call to the Microsoft Graph API in the bottom of the screen.
2. Once signed in, select **Get graph data silently** to make a call to the Microsoft Graph API without prompting the user for credentials again. You'll see the output from the call to the Microsoft Graph API in the bottom of the screen.

In multiple account mode, you can repeat the same steps. Additionally, you can remove the signed-in account, which also removes the cached tokens for that account.

## How the sample works

The code is organized into fragments that show how to write a single and multiple accounts MSAL app. The code files are organized as follows:

FILE	DEMONSTRATES
MainActivity	Manages the UI
MSGraphRequestWrapper	Calls the Microsoft Graph API using the token provided by MSAL
MultipleAccountModeFragment	Initializes a multi-account application, loads a user account, and gets a token to call the Microsoft Graph API
SingleAccountModeFragment	Initializes a single-account application, loads a user account, and gets a token to call the Microsoft Graph API
res/auth_config_multiple_account.json	The multiple account configuration file

FILE	DEMONSTRATES
res/auth_config_single_account.json	The single account configuration file
Gradle Scripts/build.gradle (Module:app)	The MSAL library dependencies are added here

We'll now look at these files in more detail and call out the MSAL-specific code in each.

## Adding MSAL to the app

MSAL ([com.microsoft.identity.client](#)) is the library used to sign in users and request tokens used to access an API protected by Microsoft identity platform. Gradle 3.0+ installs the library when you add the following to **Gradle Scripts > build.gradle (Module: app)** under **Dependencies**:

```
implementation 'com.microsoft.identity.client:msal:1.0.0'
```

You can see this in the sample project in build.gradle (Module: app):

```
dependencies {
 ...
 implementation 'com.microsoft.identity.client:msal:1.0.+'
 ...
}
```

This instructs Gradle to download and build MSAL from maven central.

## MSAL imports

The imports that are relevant to the MSAL library are `com.microsoft.identity.client.*`. For example, you'll see `import com.microsoft.identity.client.PublicClientApplication;` which is the namespace for the `PublicClientApplication` class, which represents your public client application.

## SingleAccountModeFragment.java

This file demonstrates how to create a single account MSAL app and call a Microsoft Graph API.

Single account apps are only used by a single user. For example, you might just have one account that you sign into your mapping app with.

### Single account MSAL initialization

In `auth_config_single_account.json`, in `onCreateView()`, a single account `PublicClientApplication` is created using the config information stored in the `auth_config_single_account.json` file. This is how you initialize the MSAL library for use in a single-account MSAL app:

```

...
// Creates a PublicClientApplication object with res/raw/auth_config_single_account.json
PublicClientApplication.createSingleAccountPublicClientApplication(getApplicationContext(),
 R.raw.auth_config_single_account,
 new IPublicClientApplication.ISingleAccountApplicationCreatedListener() {
 @Override
 public void onCreated(ISingleAccountPublicClientApplication application) {
 /**
 * This test app assumes that the app is only going to support one account.
 * This requires "account_mode" : "SINGLE" in the config json file.
 */
 mSingleAccountApp = application;
 loadAccount();
 }

 @Override
 public void onError(MsalException exception) {
 displayError(exception);
 }
 });

```

### Sign in a user

In `SingleAccountModeFragment.java`, the code to sign in a user is in `initializeUI()`, in the `signInButton` click handler.

Call `signIn()` before trying to acquire tokens. `signIn()` behaves as though `acquireToken()` is called, resulting in an interactive prompt for the user to sign in.

Signing in a user is an asynchronous operation. A callback is passed that calls the Microsoft Graph API and update the UI once the user signs in:

```
mSingleAccountApp.signIn(getActivity(), null, getScopes(), getAuthInteractiveCallback());
```

### Sign out a user

In `SingleAccountModeFragment.java`, the code to sign out a user is in `initializeUI()`, in the `signOutButton` click handler. Signing a user out is an asynchronous operation. Signing the user out also clears the token cache for that account. A callback is created to update the UI once the user account is signed out:

```

mSingleAccountApp.signOut(new ISingleAccountPublicClientApplication.SignOutCallback() {
 @Override
 public void onSignOut() {
 updateUI(null);
 performOperationOnSignOut();
 }

 @Override
 public void onError(@NonNull MsalException exception) {
 displayError(exception);
 }
});

```

### Get a token interactively or silently

To present the fewest number of prompts to the user, you'll typically get a token silently. Then, if there's an error, attempt to get to token interactively. The first time the app calls `signIn()`, it effectively acts as a call to `acquireToken()`, which will prompt the user for credentials.

Some situations when the user may be prompted to select their account, enter their credentials, or consent to the permissions your app has requested are:

- The first time the user signs in to the application
- If a user resets their password, they'll need to enter their credentials
- If consent is revoked
- If your app explicitly requires consent
- When your application is requesting access to a resource for the first time
- When MFA or other Conditional Access policies are required

The code to get a token interactively, that is with UI that will involve the user, is in `SingleAccountModeFragment.java`, in `initializeUI()`, in the `callGraphApiInteractiveButton` click handler:

```
/**
 * If acquireTokenSilent() returns an error that requires an interaction (MsalUiRequiredException),
 * invoke acquireToken() to have the user resolve the interrupt interactively.
 *
 * Some example scenarios are
 * - password change
 * - the resource you're acquiring a token for has a stricter set of requirement than your Single Sign-On
refresh token.
 * - you're introducing a new scope which the user has never consented for.
 */
mSingleAccountApp.acquireToken(getActivity(), getScopes(), getAuthInteractiveCallback());
```

If the user has already signed in, `acquireTokenSilentAsync()` allows apps to request tokens silently as shown in `initializeUI()`, in the `callGraphApiSilentButton` click handler:

```
/**
 * Once you've signed the user in,
 * you can perform acquireTokenSilent to obtain resources without interrupting the user.
 */
mSingleAccountApp.acquireTokenSilentAsync(getScopes(), AUTHORITY, getAuthSilentCallback());
```

### Load an account

The code to load an account is in `SingleAccountModeFragment.java` in `loadAccount()`. Loading the user's account is an asynchronous operation, so callbacks to handle when the account loads, changes, or an error occurs is passed to MSAL. The following code also handles `onAccountChanged()`, which occurs when an account is removed, the user changes to another account, and so on.

```

private void loadAccount() {
 ...

 mSingleAccountApp.getCurrentAccountAsync(new
 ISingleAccountPublicClientApplication.CurrentAccountCallback() {
 @Override
 public void onAccountLoaded(@Nullable IAccount activeAccount) {
 // You can use the account data to update your UI or your app database.
 updateUI(activeAccount);
 }

 @Override
 public void onAccountChanged(@Nullable IAccount priorAccount, @Nullable IAccount currentAccount) {
 if (currentAccount == null) {
 // Perform a cleanup task as the signed-in account changed.
 performOperationOnSignOut();
 }
 }

 @Override
 public void onError(@NonNull MsalException exception) {
 displayError(exception);
 }
 });
}

```

### Call Microsoft Graph

When a user is signed in, the call to Microsoft Graph is made via an HTTP request by `callGraphAPI()` which is defined in `SingleAccountModeFragment.java`. This function is a wrapper that simplifies the sample by doing some tasks such as getting the access token from the `authenticationResult` and packaging the call to the `MSGraphRequestWrapper`, and displaying the results of the call.

```

private void callGraphAPI(final IAuthenticationResult authenticationResult) {
 MSGraphRequestWrapper.callGraphAPIUsingVolley(
 getContext(),
 graphResourceTextView.getText().toString(),
 authenticationResult.getAccessToken(),
 new Response.Listener<JSONObject>() {
 @Override
 public void onResponse(JSONObject response) {
 /* Successfully called graph, process data and send to UI */
 ...
 }
 },
 new Response.ErrorListener() {
 @Override
 public void onErrorResponse(VolleyError error) {
 ...
 }
 });
}

```

### auth\_config\_single\_account.json

This is the configuration file for a MSAL app that uses a single account.

See [Understand the Android MSAL configuration file](#) for an explanation of these fields.

Note the presence of `"account_mode" : "SINGLE"`, which configures this app to use a single account.

`"client_id"` is preconfigured to use an app object registration that Microsoft maintains. `"redirect_uri"` is preconfigured to use the signing key provided with the code sample.

```
{
 "client_id" : "0984a7b6-bc13-4141-8b0d-8f767e136bb7",
 "authorization_user_agent" : "DEFAULT",
 "redirect_uri" : "msauth://com.azureexamples.msalandroidapp/1wIqXSqBj7w%2Bh11ZifsqnwggyKrY%3D",
 "account_mode" : "SINGLE",
 "broker_redirect_uri_registered": true,
 "authorities" : [
 {
 "type": "AAD",
 "audience": {
 "type": "AzureADandPersonalMicrosoftAccount",
 "tenant_id": "common"
 }
 }
]
}
```

## MultipleAccountModeFragment.java

This file demonstrates how to create a multiple account MSAL app and call a Microsoft Graph API.

An example of a multiple account app is a mail app that allows you to work with multiple user accounts such as a work account and a personal account.

### Multiple account MSAL initialization

In the `MultipleAccountModeFragment.java` file, in `onCreateView()`, a multiple account app object (`IMultipleAccountPublicClientApplication`) is created using the config information stored in the `auth_config_multiple_account.json` file :

```
// Creates a PublicClientApplication object with res/raw/auth_config_multiple_account.json
PublicClientApplication.createMultipleAccountPublicClientApplication(getApplicationContext(),
 R.raw.auth_config_multiple_account,
 new IPublicClientApplication.IMultipleAccountApplicationCreatedListener() {
 @Override
 public void onCreated(IMultipleAccountPublicClientApplication application) {
 mMultipleAccountApp = application;
 loadAccounts();
 }

 @Override
 public void onError(MsalException exception) {
 ...
 }
 });
});
```

The created `MultipleAccountPublicClientApplication` object is stored in a class member variable so that it can be used to interact with the MSAL library to acquire tokens and load and remove the user account.

### Load an account

Multiple account apps usually call `getAccounts()` to select the account to use for MSAL operations. The code to load an account is in the `MultipleAccountModeFragment.java` file, in `loadAccounts()`. Loading the user's account is an asynchronous operation. So a callback handles the situations when the account is loaded, changes, or an error occurs.

```

/**
 * Load currently signed-in accounts, if there's any.
 */
private void loadAccounts() {
 if (mMultipleAccountApp == null) {
 return;
 }

 mMultipleAccountApp.getAccounts(new IPublicClientApplication.LoadAccountsCallback() {
 @Override
 public void onTaskCompleted(final List<IAccount> result) {
 // You can use the account data to update your UI or your app database.
 accountList = result;
 updateUI(accountList);
 }

 @Override
 public void onError(MsalException exception) {
 displayError(exception);
 }
 });
}

```

#### Get a token interactively or silently

Some situations when the user may be prompted to select their account, enter their credentials, or consent to the permissions your app has requested are:

- The first time users sign in to the application
- If a user resets their password, they'll need to enter their credentials
- If consent is revoked
- If your app explicitly requires consent
- When your application is requesting access to a resource for the first time
- When MFA or other Conditional Access policies are required

Multiple account apps should typically acquire tokens interactively, that is with UI that involves the user, with a call to `acquireToken()`. The code to get a token interactively is in the `MultipleAccountModeFragment.java` file in `initializeUI()`, in the `callGraphApiInteractiveButton` click handler:

```

/**
 * Acquire token interactively. It will also create an account object for the silent call as a result (to be
 * obtained by getAccount()).
 *
 * If acquireTokenSilent() returns an error that requires an interaction,
 * invoke acquireToken() to have the user resolve the interrupt interactively.
 *
 * Some example scenarios are
 * - password change
 * - the resource you're acquiring a token for has a stricter set of requirement than your SSO refresh token.
 * - you're introducing a new scope which the user has never consented for.
 */
mMultipleAccountApp.acquireToken(getActivity(), getScopes(), getAuthInteractiveCallback());

```

Apps shouldn't require the user to sign in every time they request a token. If the user has already signed in, `acquireTokenSilentAsync()` allows apps to request tokens without prompting the user, as shown in the `MultipleAccountModeFragment.java` file, in `initializeUI()` in the `callGraphApiSilentButton` click handler:

```

/**
 * Performs acquireToken without interrupting the user.
 *
 * This requires an account object of the account you're obtaining a token for.
 * (can be obtained via getAccount()).
 */
mMultipleAccountApp.acquireTokenSilentAsync(getScopes(),
 accountList.get(accountListSpinner.getSelectedItemPosition()),
 AUTHORITY,
 getAuthSilentCallback());

```

### Remove an account

The code to remove an account, and any cached tokens for the account, is in the `MultipleAccountModeFragment.java` file in `initializeUI()` in the handler for the remove account button. Before you can remove an account, you need an account object, which you obtain from MSAL methods like `getAccounts()` and `acquireToken()`. Because removing an account is an asynchronous operation, the `onRemoved` callback is supplied to update the UI.

```

/**
 * Removes the selected account and cached tokens from this app (or device, if the device is in shared mode).
 */
mMultipleAccountApp.removeAccount(accountList.get(accountListSpinner.getSelectedItemPosition()),
 new IMultipleAccountPublicClientApplication.RemoveAccountCallback() {
 @Override
 public void onRemoved() {
 ...
 /* Reload account asynchronously to get the up-to-date list. */
 loadAccounts();
 }

 @Override
 public void onError(@NonNull MsalException exception) {
 displayError(exception);
 }
 });

```

### auth\_config\_multiple\_account.json

This is the configuration file for a MSAL app that uses multiple accounts.

See [Understand the Android MSAL configuration file](#) for an explanation of the various fields.

Unlike the `auth_config_single_account.json` configuration file, this config file has `"account_mode" : "MULTIPLE"` instead of `"account_mode" : "SINGLE"` because this is a multiple account app.

`"client_id"` is preconfigured to use an app object registration that Microsoft maintains. `"redirect_uri"` is preconfigured to use the signing key provided with the code sample.

```
{
 "client_id" : "0984a7b6-bc13-4141-8b0d-8f767e136bb7",
 "authorization_user_agent" : "DEFAULT",
 "redirect_uri" : "msauth://com.azuresamples.msalandroidapp/1wIqXSqBj7w%2Bh11ZifsqnwggyKrY%3D",
 "account_mode" : "MULTIPLE",
 "broker_redirect_uri_registered": true,
 "authorities" : [
 {
 "type": "AAD",
 "audience": {
 "type": "AzureADandPersonalMicrosoftAccount",
 "tenant_id": "common"
 }
 }
]
}
```

## Next steps

### Learn the steps to create the application used in this quickstart

Try out the [Sign in users and call the Microsoft Graph from an Android app](#) tutorial for a step-by-step guide for building an Android app that gets an access token and uses it to call the Microsoft Graph API.

[Call Graph API Android tutorial](#)

### MSAL for Android library wiki

Read more information about MSAL library for Android:

[MSAL for Android library wiki](#)

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

# Quickstart: Sign in users and call the Microsoft Graph API from an iOS or macOS app

10/30/2019 • 6 minutes to read • [Edit Online](#)

This quickstart contains a code sample that demonstrates how a native iOS or macOS application can use the Microsoft identity platform to sign in personal, work and school accounts, get an access token, and call the Microsoft Graph API.

This quickstart applies to both iOS and macOS apps. Some steps are needed only for iOS apps. Those steps call out that they are only for iOS.

## NOTE

### Prerequisites

- XCode 10+
- iOS 10+
- macOS 10.12+

## Register and download your quickstart app

You have two options to start your quickstart application:

- [Express] [Option 1: Register and auto configure your app and then download your code sample](#)
- [Manual] [Option 2: Register and manually configure your application and code sample](#)

### Option 1: Register and auto configure your app and then download the code sample

#### Step 1: Register your application

To register your app,

1. Go to the new [Azure portal - App registrations](#) pane.
2. Enter a name for your application and select **Register**.
3. Follow the instructions to download and automatically configure your new application with just one click.

### Option 2: Register and manually configure your application and code sample

#### Step 1: Register your application

To register your application and add the app's registration information to your solution manually, follow these steps:

1. Navigate to the Microsoft identity platform for developers [App registrations](#) page.
2. Select **New registration**.
3. When the **Register an application** page appears, enter your application's registration information:
  - In the **Name** section, enter a meaningful application name that will be displayed to users of the app when they sign in or consent to your app.
  - Skip other configurations on this page.
  - Select **Register**.
4. In the **Manage** section, select **Authentication** > **Add Platform** > **iOS**.

- Enter the **Bundle Identifier** for your application. The bundle identifier is just a unique string that uniquely identifies your application, for example `com.<yourname>.identitysample.MSALMacOS`. Make a note of the value you use.
- Note that the iOS configuration is also applicable to macOS applications.

5. Select `Configure` and save the **MSAL Configuration** details for later in this quickstart.

#### Step 2: Download the sample project

- [Download the Code Sample for iOS](#)
- [Download the Code Sample for macOS](#)

#### Step 3: Install dependencies

In a terminal window, navigate to the folder with the downloaded code sample and run `pod install` to install the latest MSAL library.

#### Step 4: Configure your project

If you selected Option 1 above, you can skip these steps.

1. Extract the zip file and open the project in XCode.
2. Edit **ViewController.swift** and replace the line starting with 'let kClientID' with the following code snippet. Remember to update the value for `kClientID` with the clientID that you saved when you registered your app in the portal earlier in this quickstart:

```
let kClientID = "Enter_the_Application_Id_Here"
```

3. Open the project settings. In the **Identity** section, enter the **Bundle Identifier** that you entered into the portal.
4. For iOS only, right-click **Info.plist** and select **Open As > Source Code**.
5. For iOS only, under the dict root node, replace `Enter_the_bundle_Id_Here` with the **Bundle Id** that you used in the portal.

```
<key>CFBundleURLTypes</key>
<array>
<dict>
<key>CFBundleURLSchemes</key>
<array>
<string>msauth.Enter_the_Bundle_Id_Here</string>
</array>
</dict>
</array>
```

6. Build & run the app!

## More Information

Read these sections to learn more about this quickstart.

### Get MSAL

MSAL ([MSAL.framework](#)) is the library used to sign in users and request tokens used to access an API protected by Microsoft identity platform. You can add MSAL to your application using the following process:

```
$ vi Podfile
```

Add the following to this podfile (with your project's target):

```
use_frameworks!

target 'MSALiOS' do
 pod 'MSAL'
end
```

Run CocoaPods installation command:

```
pod install
```

## Initialize MSAL

You can add the reference for MSAL by adding the following code:

```
import MSAL
```

Then, initialize MSAL using the following code:

```
let authority = try MSALAADAuthority(url: URL(string: kAuthority)!)

let msalConfiguration = MSALPublicClientApplicationConfig(clientId: kClientId, redirectUri: nil, authority:
authority)
self.applicationContext = try MSALPublicClientApplication(configuration: msalConfiguration)
```

WHERE:	
<code>clientId</code>	The Application ID from the application registered in <a href="https://portal.azure.com">portal.azure.com</a>
<code>authority</code>	The Microsoft identity platform endpoint. In most of cases this will be <a href="https://login.microsoftonline.com/common">https://login.microsoftonline.com/common</a>
<code>redirectUri</code>	The redirect URI of the application. You can pass 'nil' to use the default value, or your custom redirect URI.

## For iOS only, additional app requirements

Your app must also have the following in your `AppDelegate`. This lets MSAL SDK handle token response from the Auth broker app when you do authentication.

```
func application(_ app: UIApplication, open url: URL, options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
 return MSALPublicClientApplication.handleMSALResponse(url, sourceApplication:
options[UIApplication.OpenURLOptionsKey.sourceApplication] as? String)
}
```

## NOTE

On iOS 13+, if you adopt `UISceneDelegate` instead of `UIApplicationDelegate`, place this code into the `scene:openURLContexts:` callback instead (See [Apple's documentation](#)). If you support both `UISceneDelegate` and `UIApplicationDelegate` for compatibility with older iOS, MSAL callback needs to be placed into both places.

```
func scene(_ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>) {

 guard let urlContext = URLContexts.first else {
 return
 }

 let url = urlContext.url
 let sourceApp = urlContext.options.sourceApplication

 MSALPublicClientApplication.handleMSALResponse(url, sourceApplication: sourceApp)
}
```

Finally, your app must have an `LSApplicationQueriesSchemes` entry in your **Info.plist** alongside the `CFBundleURLTypes`. The sample comes with this included.

```
<key>LSApplicationQueriesSchemes</key>
<array>
 <string>msauthv2</string>
 <string>msauthv3</string>
</array>
```

## Sign in users & request tokens

MSAL has two methods used to acquire tokens: `acquireToken` and `acquireTokenSilent`.

### acquireToken: Get a token interactively

Some situations require users to interact with Microsoft identity platform. In these cases, the end user may be required to select their account, enter their credentials, or consent to your app's permissions. For example,

- The first time users sign in to the application
- If a user resets their password, they'll need to enter their credentials
- When your application is requesting access to a resource for the first time
- When MFA or other Conditional Access policies are required

```
let parameters = MSALInteractiveTokenParameters(scopes: kScopes, webViewParameters: self.webViewParamaters!)
self.applicationContext!.acquireToken(with: parameters) { (result, error) in /* Add your handling logic */}
```

### WHERE:

`scopes`

Contains the scopes being requested (that is, `[ "user.read" ]` for Microsoft Graph or `[ "<Application ID URL>/scope" ]` for custom Web APIs (`api://<Application ID>/access_as_user`)

### acquireTokenSilent: Get an access token silently

Apps shouldn't require their users to sign in every time they request a token. If the user has already signed in, this method allows apps to request tokens silently.

```
guard let account = try self.applicationContext!.allAccounts().first else { return }

let silentParams = MSALSilentTokenParameters(scopes: kScopes, account: account)
self.applicationContext!.acquireTokenSilent(with: silentParams) { (result, error) in /* Add your handling
logic */}
```

WHERE:	
<code>scopes</code>	Contains the scopes being requested (that is, [ "user.read" ] for Microsoft Graph or [ "<Application ID URL>/scope" ] for custom Web APIs ( api://<Application ID>/access_as_user )
<code>account</code>	The account a token is being requested for. This quickstart is about a single account application. If you want to build a multi-account app you'll need to define logic to identify which account to use for token requests using <code>applicationContext.account(forHomeAccountId: self.homeAccountId)</code>

## Next steps

Try out the iOS tutorial for a complete step-by-step guide on building applications, including a complete explanation of this quickstart.

**Learn how to create the application used in this quickstart**

[Call Graph API iOS tutorial](#)

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

# Quickstart: Call the Microsoft Graph API from a Universal Windows Platform (UWP) application

10/27/2019 • 4 minutes to read • [Edit Online](#)

This quickstart contains a code sample that demonstrates how a Universal Windows Platform (UWP) application can sign in users with personal accounts or work and school accounts, get an access token, and call the Microsoft Graph API.

## Register and download your quickstart app

### Step 1: Register your application

To register your application and add the app's registration information to your solution, follow these steps:

1. Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. Navigate to the Microsoft identity platform for developers [App registrations](#) page.
4. Select **New registration**.
5. When the **Register an application** page appears, enter your application's registration information:
  - In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `UWP-App-calling-MsGraph`.
  - In the **Supported account types** section, select **Accounts in any organizational directory and personal Microsoft accounts (for example, Skype, Xbox, Outlook.com)**.
  - Select **Register** to create the application.
6. In the list of pages for the app, select **Authentication**.
7. Expand the **Desktop + devices** section. (If **Desktop + devices** is not visible, first click the top banner to view the preview Authentication experience)
8. Under the **Redirect URI** section, select **Add URI**. Type `urn:ietf:wg:oauth:2.0:oob`.
9. Select **Save**.

### Step 2: Download your Visual Studio project

- [Download the Visual Studio project](#)

### Step 3: Configure your Visual Studio project

1. Extract the zip file to a local folder close to the root of the disk, for example, `C:\Azure-Samples`.
2. Open the project in Visual Studio. You might be prompted to install a UWP SDK. In that case, accept.
3. Edit `MainPage.Xaml.cs` and replace the values of the `ClientId` field:

```
private const string ClientId = "Enter_the_Application_Id_here";
```

Where:

- `Enter_the_Application_Id_here` - is the Application Id for the application you registered.

**TIP**

To find the value of *Application ID*, go to the **Overview** section in the portal

**Step 4: Run your application**

If you want to try the quickstart on your Windows machine:

1. In the Visual Studio toolbar, choose the right platform (probably **x64** or **x86**, not ARM).

Observe that the target device changes from *Device* to *Local Machine*

2. Select Debug | **Start Without Debugging**

## More information

This section provides more information about the quickstart.

### MSAL.NET

MSAL ([Microsoft.Identity.Client](#)) is the library used to sign in users and request security tokens. The security tokens are used to access an API protected by Microsoft Identity platform for developers. You can install MSAL by running the following command in Visual Studio's *Package Manager Console*:

```
Install-Package Microsoft.Identity.Client -IncludePrerelease
```

### MSAL initialization

You can add the reference for MSAL by adding the following code:

```
using Microsoft.Identity.Client;
```

Then, MSAL is initialized using the following code:

```
public static IPublicClientApplication PublicClientApp;
PublicClientApp = new PublicClientApplicationBuilder.Create(ClientId)
 .Build();
```

**WHERE:**

`ClientId`

Is the **Application (client) ID** for the application registered in the Azure portal. You can find this value in the app's **Overview** page in the Azure portal.

## Requesting tokens

MSAL has two methods for acquiring tokens in a UWP app: `AcquireTokenInteractive` and `AcquireTokenSilent`.

### Get a user token interactively

Some situations require forcing users to interact with the Microsoft identity platform endpoint through a popup window to either validate their credentials or to give consent. Some examples include:

- The first-time users sign in to the application
- When users may need to reenter their credentials because the password has expired
- When your application is requesting access to a resource, that the user needs to consent to

- When two factor authentication is required

```
authResult = await App.PublicClientApp.AcquireTokenInteractive(scopes)
 .ExecuteAsync();
```

WHERE:	
<code>scopes</code>	Contains the scopes being requested, such as <code>{ "user.read" }</code> for Microsoft Graph or <code>{ "api://&lt;Application ID&gt;/access_as_user" }</code> for custom Web APIs.

#### Get a user token silently

Use the `AcquireTokenSilent` method to obtain tokens to access protected resources after the initial `AcquireTokenInteractive` method. You don't want to require the user to validate their credentials every time they need to access a resource. Most of the time you want token acquisitions and renewal without any user interaction

```
var accounts = await App.PublicClientApp.GetAccountsAsync();
var firstAccount = accounts.FirstOrDefault();
authResult = await App.PublicClientApp.AcquireTokenSilent(scopes, firstAccount)
 .ExecuteAsync();
```

WHERE:	
<code>scopes</code>	Contains the scopes being requested, such as <code>{ "user.read" }</code> for Microsoft Graph or <code>{ "api://&lt;Application ID&gt;/access_as_user" }</code> for custom Web APIs
<code>firstAccount</code>	Specifies the first user account in the cache (MSAL supports multiple users in a single app)

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

## Next steps

Try out the Windows desktop tutorial for a complete step-by-step guide on building applications and new features, including a full explanation of this quickstart.

[UWP - Call Graph API tutorial](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

# Quickstart: Acquire a token and call Microsoft Graph API from a Windows desktop app

8/7/2019 • 4 minutes to read • [Edit Online](#)

In this quickstart, you'll learn how to write a Windows desktop .NET (WPF) application that can sign in personal, work and school accounts, get an access token, and call the Microsoft Graph API.

## Register and download your quickstart app

You have two options to start your quickstart application:

- [Express] [Option 1: Register and auto configure your app and then download your code sample](#)
- [Manual] [Option 2: Register and manually configure your application and code sample](#)

### Option 1: Register and auto configure your app and then download your code sample

1. Go to the new [Azure portal - App registrations](#).
2. Enter a name for your application and select **Register**.
3. Follow the instructions to download and automatically configure your new application with just one click.

### Option 2: Register and manually configure your application and code sample

#### Step 1: Register your application

To register your application and add the app's registration information to your solution manually, follow these steps:

1. Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. Navigate to the Microsoft identity platform for developers [App registrations](#) page.
4. Select **New registration**.
  - In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `Win-App-calling-MsGraph`.
  - In the **Supported account types** section, select **Accounts in any organizational directory and personal Microsoft accounts (for example, Skype, Xbox, Outlook.com)**.
  - Select **Register** to create the application.
5. In the list of pages for the app, select **Authentication**.
6. Expand the **Desktop + devices** section. (If **Desktop + devices** is not visible, first click the top banner to view the preview Authentication experience)
7. Under the **Redirect URI** section, select **Add URI**. Type `urn:ietf:wg:oauth:2.0:oob`.
8. Select **Save**.

#### Step 2: Download your Visual Studio project

[Download the Visual Studio project \(View Project on GitHub\)](#)

#### Step 3: Configure your Visual Studio project

1. Extract the zip file to a local folder close to the root of the disk, for example, `C:\Azure-Samples`.
2. Open the project in Visual Studio.

3. Edit **App.Xaml.cs** and replace the values of the fields `ClientId` and `Tenant` with the following code:

```
private static string ClientId = "Enter_the_Application_Id_here";
private static string Tenant = "Enter_the_Tenant_Info_Here";
```

Where:

- `Enter_the_Application_Id_here` - is the **Application (client) ID** for the application you registered.
- `Enter_the_Tenant_Info_Here` - is set to one of the following options:
  - If your application supports **Accounts in this organizational directory**, replace this value with the **Tenant Id** or **Tenant name** (for example, contoso.microsoft.com)
  - If your application supports **Accounts in any organizational directory**, replace this value with `organizations`
  - If your application supports **Accounts in any organizational directory and personal Microsoft accounts**, replace this value with `common`

#### TIP

To find the values of **Application (client) ID**, **Directory (tenant) ID**, and **Supported account types**, go to the app's **Overview** page in the Azure portal.

## More information

### MSAL.NET

MSAL ([Microsoft.Identity.Client](#)) is the library used to sign in users and request tokens used to access an API protected by Microsoft identity platform. You can install MSAL by running the following command in Visual Studio's **Package Manager Console**:

```
Install-Package Microsoft.Identity.Client -IncludePrerelease
```

### MSAL initialization

You can add the reference for MSAL by adding the following code:

```
using Microsoft.Identity.Client;
```

Then, initialize MSAL using the following code:

```
public static IPublicClientApplication PublicClientApp;
PublicClientApplicationBuilder.Create(ClientId)
 .WithAuthority(AzureCloudInstance.AzurePublic, Tenant)
 .Build();
```

#### WHERE:

`ClientId`

Is the **Application (client) ID** for the application registered in the Azure portal. You can find this value in the app's **Overview** page in the Azure portal.

## Requesting tokens

MSAL has two methods for acquiring tokens: `AcquireTokenInteractive` and `AcquireTokenSilent`.

### Get a user token interactively

Some situations require forcing users to interact with the Microsoft identity platform endpoint through a popup window to either validate their credentials or to give consent. Some examples include:

- The first time users sign in to the application
- When users may need to reenter their credentials because the password has expired
- When your application is requesting access to a resource that the user needs to consent to
- When two factor authentication is required

```
authResult = await App.PublicClientApp.AcquireTokenInteractive(_scopes)
 .ExecuteAsync();
```

WHERE:	
<code>_scopes</code>	Contains the scopes being requested, such as <code>{ "user.read" }</code> for Microsoft Graph or <code>{ "api://&lt;Application ID&gt;/access_as_user" }</code> for custom Web APIs.

### Get a user token silently

You don't want to require the user to validate their credentials every time they need to access a resource. Most of the time you want token acquisitions and renewal without any user interaction. You can use the

`AcquireTokenSilent` method to obtain tokens to access protected resources after the initial

`AcquireTokenInteractive` method:

```
var accounts = await App.PublicClientApp.GetAccountsAsync();
var firstAccount = accounts.FirstOrDefault();
authResult = await App.PublicClientApp.AcquireTokenSilent(scopes, firstAccount)
 .ExecuteAsync();
```

WHERE:	
<code>scopes</code>	Contains the scopes being requested, such as <code>{ "user.read" }</code> for Microsoft Graph or <code>{ "api://&lt;Application ID&gt;/access_as_user" }</code> for custom Web APIs.
<code>firstAccount</code>	Specifies the first user in the cache (MSAL supports multiple users in a single app).

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

## Next steps

Try out the Windows desktop tutorial for a complete step-by-step guide on building applications and new features,

including a full explanation of this quickstart.

[Call Graph API tutorial](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

# Quickstart: Acquire a token and call Microsoft Graph API using console app's identity

10/30/2019 • 6 minutes to read • [Edit Online](#)

In this quickstart, you'll learn how to write a .NET Core application that can get an access token using the app's own identity and then call the Microsoft Graph API to display a [list of users](#) in the directory. This scenario is useful for situations where headless, unattended job or a windows service needs to run with an application identity, instead of a user's identity.

## Prerequisites

This quickstart requires [.NET Core 2.2](#).

## Register and download your quickstart app

### Step 1: Register your application

To register your application and add the app's registration information to your solution manually, follow these steps:

1. Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. Navigate to the Microsoft identity platform for developers [App registrations](#) page.
4. Select **New registration**.
5. When the **Register an application** page appears, enter your application's registration information.
6. In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `Daemon-console`, then select **Register** to create the application.
7. Once registered, select the **Certificates & secrets** menu.
8. Under **Client secrets**, select **+ New client secret**. Give it a name and select **Add**. Copy the secret on a safe location. You will need it to use in your code.
9. Now, select the **API Permissions** menu, select **+ Add a permission** button, select **Microsoft Graph**.
10. Select **Application permissions**.
11. Under **User** node, select **User.Read.All**, then select **Add permissions**

### Step 2: Download your Visual Studio project

[Download the Visual Studio project](#)

### Step 3: Configure your Visual Studio project

1. Extract the zip file to a local folder close to the root of the disk, for example, `C:\Azure-Samples`.
2. Open the solution in Visual Studio - **1-Call-MSGraph\daemon-console.sln** (optional).
3. Edit **appsettings.json** and replace the values of the fields `ClientId`, `Tenant` and `ClientSecret` with the following:

```
"Tenant": "Enter_the_Tenant_Id_Here",
"ClientId": "Enter_the_Application_Id_Here",
"ClientSecret": "Enter_the_Client_Secret_Here"
```

Where:

- `Enter_the_Application_Id_Here` - is the **Application (client) ID** for the application you registered.
- `Enter_the_Tenant_Id_Here` - replace this value with the **Tenant Id** or **Tenant name** (for example, contoso.microsoft.com)
- `Enter_the_Client_Secret_Here` - replace this value with the client secret created on step 1.

#### TIP

To find the values of **Application (client) ID**, **Directory (tenant) ID**, go to the app's **Overview** page in the Azure portal. To generate a new key, go to **Certificates & secrets** page.

#### Step 4: Admin consent

If you try to run the application at this point, you'll receive *HTTP 403 - Forbidden* error:

`Insufficient privileges to complete the operation`. This happens because any *app-only permission* requires

Admin consent, which means that a global administrator of your directory must give consent to your application.

Select one of the options below depending on your role:

##### Global tenant administrator

If you are a global tenant administrator, go to **API Permissions** page in the Azure Portal's Application Registration (Preview) and select **Grant admin consent for {Tenant Name}** (Where {Tenant Name} is the name of your directory).

##### Standard user

If you're a standard user of your tenant, then you need to ask a global administrator to grant admin consent for your application. To do this, give the following URL to your administrator:

```
https://login.microsoftonline.com/Enter_the_Tenant_Id_Here/adminconsent?
client_id=Enter_the_Application_Id_Here
```

Where:

- `Enter_the_Tenant_Id_Here` - replace this value with the **Tenant Id** or **Tenant name** (for example, contoso.microsoft.com)
- `Enter_the_Application_Id_Here` - is the **Application (client) ID** for the application you registered.

#### NOTE

You may see the error '*AADSTS50011: No reply address is registered for the application*' after granting consent to the app using the preceding URL. This happen because this application and the URL do not have a redirect URI - please ignore the error.

#### Step 5: Run the application

If you're using Visual Studio, press **F5** to run the application, otherwise, run the application via command prompt or console:

```
cd {ProjectFolder}\daemon-console\1-Call-Graph
dotnet run
```

Where:

- *{ProjectFolder}* is the folder where you extracted the zip file. Example **C:\Azure-Samples\active-directory-dotnetcore-daemon-v2**

You should see a list of users in your Azure AD directory as result.

#### IMPORTANT

This quickstart application uses a client secret to identify itself as confidential client. Because the client secret is added as a plain-text to your project files, for security reasons, it is recommended that you use a certificate instead of a client secret before considering the application as production application. For more information on how to use a certificate, see [these instructions](#) in the GitHub repository for this sample.

## More information

### MSAL.NET

MSAL ([Microsoft.Identity.Client](#)) is the library used to sign in users and request tokens used to access an API protected by Microsoft identity platform. As described, this quickstart requests tokens by using the application own identity instead of delegated permissions. The authentication flow used in this case is known as [client credentials oauth flow](#). For more information on how to use MSAL.NET with client credentials flow, see [this article](#).

You can install MSAL.NET by running the following command in Visual Studio's **Package Manager Console**:

```
Install-Package Microsoft.Identity.Client
```

Alternatively, if you are not using Visual Studio, you can run the following command to add MSAL to your project:

```
dotnet add package Microsoft.Identity.Client
```

### MSAL initialization

You can add the reference for MSAL by adding the following code:

```
using Microsoft.Identity.Client;
```

Then, initialize MSAL using the following code:

```
IConfidentialClientApplication app;
app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
 .WithClientSecret(config.ClientSecret)
 .WithAuthority(new Uri(config.Authority))
 .Build();
);
```

WHERE:

WHERE:	
<code>config.ClientSecret</code>	Is the client secret created for the application in Azure Portal.
<code>config.ClientId</code>	Is the <b>Application (client) ID</b> for the application registered in the Azure portal. You can find this value in the app's <b>Overview</b> page in the Azure portal.
<code>config.Authority</code>	(Optional) The STS endpoint for user to authenticate. Usually <a href="https://login.microsoftonline.com/{tenant}">https://login.microsoftonline.com/{tenant}</a> for public cloud, where {tenant} is the name of your tenant or your tenant Id.

For more information, please see the [reference documentation for `ConfidentialClientApplication`](#)

## Requesting tokens

To request a token using app's identity, use `AcquireTokenForClient` method:

```
result = await app.AcquireTokenForClient(scopes)
 .ExecuteAsync();
```

WHERE:	
<code>scopes</code>	Contains the scopes requested. For confidential clients, this should use the format similar to <code>{Application ID URI}/.default</code> to indicate that the scopes being requested are the ones statically defined in the app object set in the Azure Portal (for Microsoft Graph, <code>{Application ID URI}</code> points to <a href="https://graph.microsoft.com">https://graph.microsoft.com</a> ). For custom Web APIs, <code>{Application ID URI}</code> is defined under <b>Expose an API</b> section in Azure Portal's Application Registration (Preview).

For more information, please see the [reference documentation for `AcquireTokenForClient`](#)

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

## Next steps

To learn more about daemon applications, see the scenario landing page

[Daemon application that calls web APIs](#)

For the daemon application tutorial, see:

[Daemon .NET Core console tutorial](#)

Learn more about permissions and consent:

## Permissions and Consent

To know more about the auth flow for this scenario, see the Oauth 2.0 client credentials flow:

### [Client credentials Oauth flow](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

### [Microsoft identity platform survey](#)

# Quickstart: Acquire a token and call Microsoft Graph API from a Python console app using app's identity

10/27/2019 • 6 minutes to read • [Edit Online](#)

In this quickstart, write a Python application that gets an access token using the app's identity, and then calls the Microsoft Graph API to display a [list of users](#) in the directory. This scenario is useful for situations where headless, unattended job or a windows service needs to run with an application identity, instead of a user's identity.

## Prerequisites

To run this sample, you need:

- [Python 2.7+](#) or [Python 3+](#)
- [MSAL Python](#)

## Register and download your quickstart app

### Step 1: Register your application

To register your application and add the app's registration information to your solution manually, follow these steps:

1. Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. Navigate to the Microsoft identity platform for developers [App registrations](#) page.
4. Select **New registration**.
5. When the **Register an application** page appears, enter your application's registration information.
6. In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `Daemon-console`, then select **Register** to create the application.
7. Once registered, select the **Certificates & secrets** menu.
8. Under **Client secrets**, select **+ New client secret**. Give it a name and select **Add**. Copy the secret on a safe location. You will need it to use in your code.
9. Now, select the **API Permissions** menu, select **+ Add a permission** button, select **Microsoft Graph**.
10. Select **Application permissions**.
11. Under **User** node, select **User.Read.All**, then select **Add permissions**

### Step 2: Download your Python project

[Download the Python daemon project](#)

### Step 3: Configure your Python project

1. Extract the zip file to a local folder close to the root of the disk, for example, `C:\Azure-Samples`.
2. Navigate to the sub folder **1-Call-MsGraph-WithSecret**.
3. Edit **parameters.json** and replace the values of the fields `authority`, `client_id`, and `secret` with the following snippet:

```
"authority": "https://login.microsoftonline.com/Enter_the_Tenant_Id_Here",
"client_id": "Enter_the_Application_Id_Here",
"secret": "Enter_the_Client_Secret_Here"
```

Where:

- `Enter_the_Application_Id_Here` - is the **Application (client) ID** for the application you registered.
- `Enter_the_Tenant_Id_Here` - replace this value with the **Tenant Id** or **Tenant name** (for example, contoso.microsoft.com)
- `Enter_the_Client_Secret_Here` - replace this value with the client secret created on step 1.

#### TIP

To find the values of **Application (client) ID**, **Directory (tenant) ID**, go to the app's **Overview** page in the Azure portal. To generate a new key, go to **Certificates & secrets** page.

#### Step 4: Admin consent

If you try to run the application at this point, you'll receive *HTTP 403 - Forbidden* error:

`Insufficient privileges to complete the operation`. This error happens because any *app-only permission* requires Admin consent: a global administrator of your directory must give consent to your application. Select one of the options below depending on your role:

##### Global tenant administrator

If you are a global tenant administrator, go to **API Permissions** page in the Azure Portal's Application Registration (Preview) and select **Grant admin consent for {Tenant Name}** (Where {Tenant Name} is the name of your directory).

##### Standard user

If you're a standard user of your tenant, then you need to ask a global administrator to grant admin consent for your application. To do this, give the following URL to your administrator:

```
https://login.microsoftonline.com/Enter_the_Tenant_Id_Here/adminconsent?client_id=Enter_the_Application_Id_Here
```

Where:

- `Enter_the_Tenant_Id_Here` - replace this value with the **Tenant Id** or **Tenant name** (for example, contoso.microsoft.com)
- `Enter_the_Application_Id_Here` - is the **Application (client) ID** for the application you registered.

#### Step 5: Run the application

You'll need to install the dependencies of this sample once

```
pip install -r requirements.txt
```

Then, run the application via command prompt or console:

```
python confidential_client_secret_sample.py parameters.json
```

You should see on the console output some Json fragment representing a list of users in your Azure AD directory.

## IMPORTANT

This quickstart application uses a client secret to identify itself as confidential client. Because the client secret is added as a plain-text to your project files, for security reasons, it is recommended that you use a certificate instead of a client secret before considering the application as production application. For more information on how to use a certificate, see [these instructions](#) in the same GitHub repository for this sample, but in the second folder **2-Call-MsGraph-WithCertificate**

## More information

### MSAL Python

[MSAL Python](#) is the library used to sign in users and request tokens used to access an API protected by Microsoft identity platform. As described, this quickstart requests tokens by using the application own identity instead of delegated permissions. The authentication flow used in this case is known as [client credentials oauth flow](#). For more information on how to use MSAL Python with daemon apps, see [this article](#).

You can install MSAL Python by running the following pip command.

```
pip install msal
```

### MSAL initialization

You can add the reference for MSAL by adding the following code:

```
import msal
```

Then, initialize MSAL using the following code:

```
app = msal.ConfidentialClientApplication(
 config["client_id"], authority=config["authority"],
 client_credential=config["secret"])
```

WHERE:	
<code>config["secret"]</code>	Is the client secret created for the application in Azure Portal.
<code>config["client_id"]</code>	Is the <b>Application (client) ID</b> for the application registered in the Azure portal. You can find this value in the app's <b>Overview</b> page in the Azure portal.
<code>config["authority"]</code>	The STS endpoint for user to authenticate. Usually <a href="https://login.microsoftonline.com/{tenant}">https://login.microsoftonline.com/{tenant}</a> for public cloud, where {tenant} is the name of your tenant or your tenant Id.

For more information, please see the [reference documentation for `ConfidentialClientApplication`](#)

### Requesting tokens

To request a token using app's identity, use `AcquireTokenForClient` method:

```
result = None
result = app.acquire_token_silent(config["scope"], account=None)

if not result:
 logging.info("No suitable token exists in cache. Let's get a new one from AAD.")
 result = app.acquire_token_for_client(scopes=config["scope"])
```

WHERE:	
<code>config["scope"]</code>	Contains the scopes requested. For confidential clients, this should use the format similar to <code>{Application ID URI}/.default</code> to indicate that the scopes being requested are the ones statically defined in the app object set in the Azure Portal (for Microsoft Graph, <code>{Application ID URI}</code> points to <code>https://graph.microsoft.com</code> ). For custom Web APIs, <code>{Application ID URI}</code> is defined under <b>Expose an API</b> section in Azure Portal's Application Registration (Preview).

For more information, please see the reference documentation for [AcquireTokenForClient](#)

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

## Next steps

To learn more about daemon applications, see the scenario landing page

[Daemon application that calls web APIs](#)

For the daemon application tutorial, see:

[Daemon Python console tutorial](#)

Learn more about permissions and consent:

[Permissions and Consent](#)

To know more about the auth flow for this scenario, see the Oauth 2.0 client credentials flow:

[Client credentials Oauth flow](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

# Sign in users and call the Microsoft Graph API from a JavaScript single-page application (SPA)

10/23/2019 • 13 minutes to read • [Edit Online](#)

This guide demonstrates how a JavaScript single-page application (SPA) can:

- Sign in personal accounts, as well as work and school accounts
- Acquire an access token
- Call the Microsoft Graph API or other APIs that require access tokens from the *Microsoft identity platform endpoint*

## How the sample app generated by this guide works

### More information

The sample application created by this guide enables a JavaScript SPA to query the Microsoft Graph API or a web API that accepts tokens from the Microsoft identity platform endpoint. In this scenario, after a user signs in, an access token is requested and added to HTTP requests through the authorization header. Token acquisition and renewal are handled by the Microsoft Authentication Library (MSAL).

### Libraries

This guide uses the following library:

LIBRARY	DESCRIPTION
<a href="#">msal.js</a>	Microsoft Authentication Library for JavaScript Preview

#### NOTE

*Msal.js* targets the Microsoft identity platform endpoint, which enables personal accounts and school and work accounts to sign in and acquire tokens. The Microsoft identity platform endpoint has [some limitations](#). To understand the differences between the v1.0 and v2.0 endpoints, see the [endpoint comparison guide](#).

## Set up your web server or project

Prefer to download this sample's project instead? Do either of the following:

- To run the project by using a local web server, such as Node.js, [download the project files](#).
- (Optional) To run the project by using the Microsoft Internet Information Services (IIS) server, [download the Visual Studio project](#).

To configure the code sample before you execute it, skip to the [configuration step](#).

## Prerequisites

- To run this tutorial, you need a local web server, such as [Node.js](#), [.NET Core](#), or IIS Express integration with [Visual Studio 2017](#).

- If you're using Node.js to run the project, install an integrated development environment (IDE), such as [Visual Studio Code](#), to edit the project files.
- Instructions in this guide are based on both Node.js and Visual Studio 2017, but you can use any other development environment or web server.

## Create your project

### Option 1: Node.js or other web servers

Make sure you have [Node.js](#) installed, and then create a folder to host your application.

### Option 2: Visual Studio

If you're using Visual Studio and are creating a new project, follow these steps:

1. In Visual Studio, select **File > New > Project**.
2. Under **Visual C#\Web**, select **ASP.NET Web Application (.NET Framework)**.
3. Enter a name for your application, and then select **OK**.
4. Under **New ASP.NET Web Application**, select **Empty**.

## Create the SPA UI

1. Create an *index.html* file for your JavaScript SPA. If you're using Visual Studio, select the project (project root folder). Right-click and select **Add > New Item > HTML page**, and name the file *index.html*.
2. In the *index.html* file, add the following code:

```
<!DOCTYPE html>
<html>
<head>
 <title>Quickstart for MSAL JS</title>
 <script src="https://cdnjs.cloudflare.com/ajax/libs/bluebird/3.3.4/bluebird.min.js"></script>
 <script src="https://secure.aadcdn.microsoftonline-p.com/lib/1.0.0/js/msal.js"></script>
</head>
<body>
 <h2>Welcome to MSAL.js Quickstart</h2>

 <h4 id="WelcomeMessage"></h4>
 <button id="SignIn" onclick="signIn()">Sign In</button>

 <pre id="json"></pre>
 <script>
 //JS code
 </script>
</body>
</html>
```

#### TIP

You can replace the version of MSAL.js in the preceding script with the latest released version under [MSAL.js releases](#).

## Use the Microsoft Authentication Library (MSAL) to sign in the user

Add the following code to your `index.html` file within the `<script></script>` tags:

```
var msalConfig = {
 auth: {
 clientId: "Enter_the_Application_Id_here",
 authority: "https://login.microsoftonline.com/Enter_the_Tenant_Info_Here"
 }
}.
```

```

 ,
 cache: {
 cacheLocation: "localStorage",
 storeAuthStateInCookie: true
 }
 });

var graphConfig = {
 graphMeEndpoint: "https://graph.microsoft.com/v1.0/me"
};

// this can be used for login or token request, however in more complex situations
// this can have diverging options
var requestObj = {
 scopes: ["user.read"]
};

var myMSALObj = new Msal.UserAgentApplication(msalConfig);
// Register Callbacks for redirect flow
myMSALObj.handleRedirectCallback(authRedirectCallBack);

function signIn() {

 myMSALObj.loginPopup(requestObj).then(function (loginResponse) {
 //Login Success
 showWelcomeMessage();
 acquireTokenPopupAndCallMSGraph();
 }).catch(function (error) {
 console.log(error);
 });
}

function acquireTokenPopupAndCallMSGraph() {
 //Always start with acquireTokenSilent to obtain a token in the signed in user from cache
 myMSALObj.acquireTokenSilent(requestObj).then(function (tokenResponse) {
 callMSGraph(graphConfig.graphMeEndpoint, tokenResponse.accessToken, graphAPICallback);
 }).catch(function (error) {
 console.log(error);
 // Upon acquireTokenSilent failure (due to consent or interaction or login required ONLY)
 // Call acquireTokenPopup(popup window)
 if (requiresInteraction(error.errorCode)) {
 myMSALObj.acquireTokenPopup(requestObj).then(function (tokenResponse) {
 callMSGraph(graphConfig.graphMeEndpoint, tokenResponse.accessToken, graphAPICallback);
 }).catch(function (error) {
 console.log(error);
 });
 }
 });
}

function graphAPICallback(data) {
 document.getElementById("json").innerHTML = JSON.stringify(data, null, 2);
}

function showWelcomeMessage() {
 var divWelcome = document.getElementById('WelcomeMessage');
 divWelcome.innerHTML = 'Welcome ' + myMSALObj.getAccount().userName + " to Microsoft Graph API";
 var loginbutton = document.getElementById('SignIn');
 loginbutton.innerHTML = 'Sign Out';
 loginbutton.setAttribute('onclick', 'signOut();');
}

//This function can be removed if you do not need to support IE
function acquireTokenRedirectAndCallMSGraph() {
 //Always start with acquireTokenSilent to obtain a token in the signed in user from cache
 myMSALObj.acquireTokenSilent(requestObj).then(function (tokenResponse) {

```

```

myMSALObj.acquireTokenSilent(requestObj).then(function (tokenResponse) {
 callMSGraph(graphConfig.graphMeEndpoint, tokenResponse.accessToken, graphAPICallback);
}).catch(function (error) {
 console.log(error);
 // Upon acquireTokenSilent failure (due to consent or interaction or login required ONLY)
 // Call acquireTokenRedirect
 if (requiresInteraction(error.errorCode)) {
 myMSALObj.acquireTokenRedirect(requestObj);
 }
});
}

function authRedirectCallBack(error, response) {
 if (error) {
 console.log(error);
 }
 else {
 if (response.tokenType === "access_token") {
 callMSGraph(graphConfig.graphEndpoint, response.accessToken, graphAPICallback);
 } else {
 console.log("token type is:" + response.tokenType);
 }
 }
}

function requiresInteraction(errorCode) {
 if (!errorCode || !errorCode.length) {
 return false;
 }
 return errorCode === "consent_required" ||
 errorCode === "interaction_required" ||
 errorCode === "login_required";
}

// Browser check variables
var ua = window.navigator.userAgent;
var msie = ua.indexOf('MSIE ');
var msie11 = ua.indexOf('Trident/');
var msedge = ua.indexOf('Edge/');
var isIE = msie > 0 || msie11 > 0;
var isEdge = msedge > 0;
//If you support IE, our recommendation is that you sign-in using Redirect APIs
//If you as a developer are testing using Edge InPrivate mode, please add "isEdge" to the if check
// can change this to default an experience outside browser use
var loginType = isIE ? "REDIRECT" : "POPUP";

if (loginType === 'POPUP') {
 if (myMSALObj.getAccount()) {// avoid duplicate code execution on page load in case of iframe and popup window.
 showWelcomeMessage();
 acquireTokenPopupAndCallMSGraph();
 }
}
else if (loginType === 'REDIRECT') {
 document.getElementById("SignIn").onclick = function () {
 myMSALObj.loginRedirect(requestObj);
 };
 if (myMSALObj.getAccount() && !myMSALObj.isCallback(window.location.hash)) {// avoid duplicate code execution on page load in case of iframe and popup window.
 showWelcomeMessage();
 acquireTokenRedirectAndCallMSGraph();
 }
}
} else {
 console.error('Please set a valid login type');
}

```

## More information

After a user selects the **Sign In** button for the first time, the `signIn` method calls `loginPopup` to sign in the user. This method opens a pop-up window with the *Microsoft identity platform endpoint* to prompt and validate the user's credentials. After a successful sign-in, the user is redirected back to the original `index.html` page. A token is received, processed by `msal.js`, and the information contained in the token is cached. This token is known as the *ID token* and contains basic information about the user, such as the user display name. If you plan to use any data provided by this token for any purposes, you need to make sure this token is validated by your backend server to guarantee that the token was issued to a valid user for your application.

The SPA generated by this guide calls `acquireTokenSilent` and/or `acquireTokenPopup` to acquire an *access token* used to query the Microsoft Graph API for user profile info. If you need a sample that validates the ID token, take a look at [this](#) sample application in GitHub. The sample uses an ASP.NET Web API for token validation.

#### Getting a user token interactively

After the initial sign-in, you do not want to ask users to reauthenticate every time they need to request a token to access a resource. So `acquireTokenSilent` should be used most of the time to acquire tokens. There are situations, however, where you need to force users to interact with Microsoft identity platform endpoint. Examples include:

- Users need to reenter their credentials because the password has expired.
- Your application is requesting access to a resource, and you need the user's consent.
- Two-factor authentication is required.

Calling `acquireTokenPopup` opens a pop-up window (or `acquireTokenRedirect` redirects users to the Microsoft identity platform endpoint). In that window, users need to interact by confirming their credentials, giving consent to the required resource, or completing the two-factor authentication.

#### Getting a user token silently

The `acquireTokenSilent` method handles token acquisition and renewal without any user interaction. After `loginPopup` (or `loginRedirect`) is executed for the first time, `acquireTokenSilent` is the method commonly used to obtain tokens used to access protected resources for subsequent calls. (Calls to request or renew tokens are made silently.) `acquireTokenSilent` may fail in some cases. For example, the user's password may have expired. Your application can handle this exception in two ways:

1. Make a call to `acquireTokenPopup` immediately, which triggers a user sign-in prompt. This pattern is commonly used in online applications where there is no unauthenticated content in the application available to the user. The sample generated by this guided setup uses this pattern.
2. Applications can also make a visual indication to the user that an interactive sign-in is required, so the user can select the right time to sign in, or the application can retry `acquireTokenSilent` at a later time. This is commonly used when the user can use other functionality of the application without being disrupted. For example, there might be unauthenticated content available in the application. In this situation, the user can decide when they want to sign in to access the protected resource, or to refresh the outdated information.

#### NOTE

This quickstart uses the `loginRedirect` and `acquireTokenRedirect` methods when Internet Explorer is the browser used. We follow this practice because of a [known issue](#) related to the way Internet Explorer handles pop-up windows.

## Call the Microsoft Graph API by using the token you just acquired

Add the following code to your `index.html` file within the `<script></script>` tags:

```

function callMSGraph(theUrl, accessToken, callback) {
 var xmlhttp = new XMLHttpRequest();
 xmlhttp.onreadystatechange = function () {
 if (this.readyState == 4 && this.status == 200)
 callback(JSON.parse(this.responseText));
 }
 xmlhttp.open("GET", theUrl, true); // true for asynchronous
 xmlhttp.setRequestHeader('Authorization', 'Bearer ' + accessToken);
 xmlhttp.send();
}

```

## More information about making a REST call against a protected API

In the sample application created by this guide, the `callMSGraph()` method is used to make an HTTP `GET` request against a protected resource that requires a token. The request then returns the content to the caller. This method adds the acquired token in the *HTTP Authorization header*. For the sample application created by this guide, the resource is the Microsoft Graph API `me` endpoint, which displays the user's profile information.

## Add a method to sign out the user

Add the following code to your `index.html` file within the `<script></script>` tags:

```

/**
 * Sign out the user
 */
function signOut() {
 myMSALObj.logout();
}

```

## Register your application

1. Sign in to the [Azure portal](#).
2. If your account gives you access to more than one tenant, select the account at the upper right, and then set your portal session to the Azure AD tenant that you want to use.
3. Go to the Microsoft identity platform for developers [App registrations](#) page.
4. When the **Register an application** page appears, enter a name for your application.
5. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
6. In the **Redirect URI** section, select the **Web** platform from the drop-down list, and then set the value to the application URL that's based on your web server.  
For information about how to set and obtain the redirect URL for Node.js and Visual Studio, see the following "Set a redirect URL for Node.js" section and [Set a redirect URL for Visual Studio](#).
7. Select **Register**.
8. On the app **Overview** page, note the **Application (client) ID** value for later use.
9. This quickstart requires the [Implicit grant flow](#) to be enabled. In the left pane of the registered application, select **Authentication**.
10. In **Advanced settings**, under **Implicit grant**, select the **ID tokens** and **Access tokens** check boxes. ID tokens and access tokens are required because this app must sign in users and call an API.

## 11. Select **Save**.

### Set a redirect URL for Node.js

For Node.js, you can set the web server port in the *server.js* file. This tutorial uses port 30662, but you can use any other available port.

To set up a redirect URL in the application registration information, switch back to the **Application Registration** pane, and do either of the following:

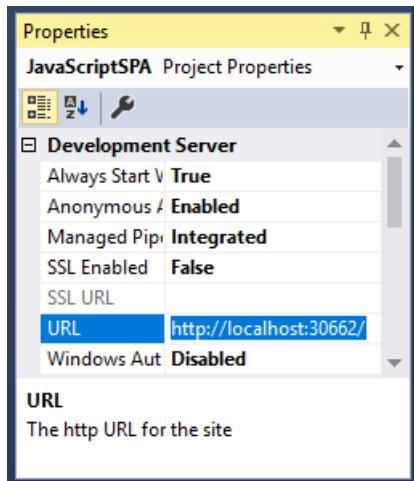
- Set `http://localhost:30662/` as the **Redirect URL**.
- If you're using a custom TCP port, use `http://localhost:<port>/` (where *<port>* is the custom TCP port number).

### Set a redirect URL for Visual Studio

To obtain the redirect URL for Visual Studio, follow these steps:

1. In Solution Explorer, select the project.

The **Properties** window opens. If it doesn't, press F4.



2. Copy the **URL** value.
3. Switch back to the **Application Registration** pane, and paste the copied value as the **Redirect URL**.

### Configure your JavaScript SPA

1. In the *index.html* file that you created during project setup, add the application registration information. At the top of the file, within the `<script></script>` tags, add the following code:

```
var msalConfig = {
 auth: {
 clientId: "<Enter_the_Application_Id_here>",
 authority: "https://login.microsoftonline.com/<Enter_the_Tenant_info_here>"
 },
 cache: {
 cacheLocation: "localStorage",
 storeAuthStateInCookie: true
 }
};
```

Where:

- *<Enter\_the\_Application\_Id\_here>* is the **Application (client) ID** for the application you registered.
- *<Enter\_the\_Tenant\_info\_here>* is set to one of the following options:
  - If your application supports *Accounts in this organizational directory*, replace this value with the

**Tenant ID** or **Tenant name** (for example, `contoso.microsoft.com`).

- If your application supports *Accounts in any organizational directory*, replace this value with **organizations**.
- If your application supports *Accounts in any organizational directory and personal Microsoft accounts*, replace this value with **common**. To restrict support to *Personal Microsoft accounts only*, replace this value with **consumers**.

## Test your code

Test your code by using either of the following environments.

### Test with Node.js

If you're not using Visual Studio, make sure your web server is started.

1. Configure the server to listen to a TCP port that's based on the location of your `index.html` file. For Node.js, start the web server to listen to the port by running the following commands at a command-line prompt from the application folder:

```
npm install
node server.js
```

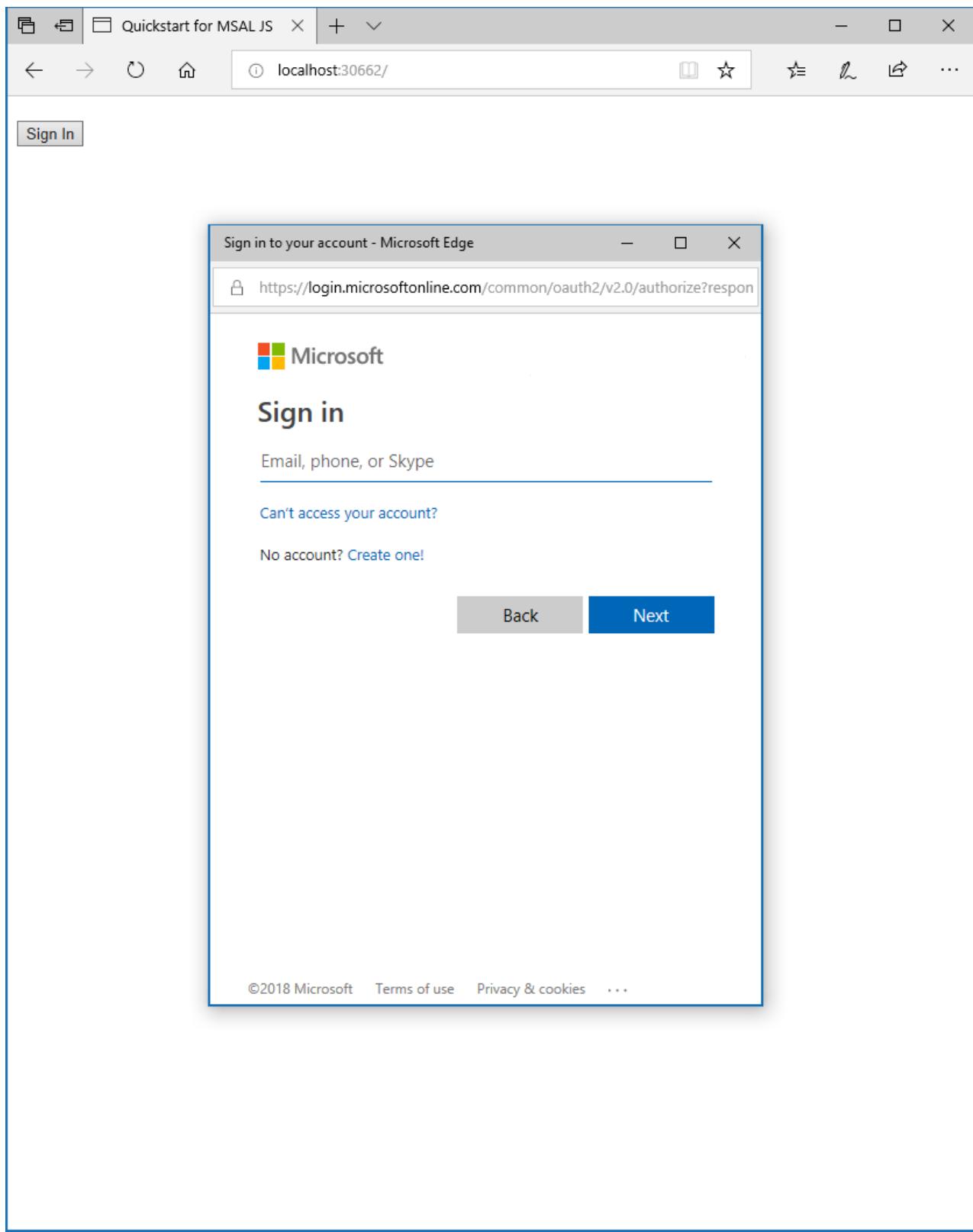
2. In your browser, enter `http://<span></span>localhost:30662` or `http://<span></span>localhost:{port}`, where *port* is the port that your web server is listening to. You should see the contents of your `index.html` file and the **Sign In** button.

### Test with Visual Studio

If you're using Visual Studio, select the project solution, and then press F5 to run your project. The browser opens to the `http://localhost:{port}` location, and the **Sign In** button should be visible.

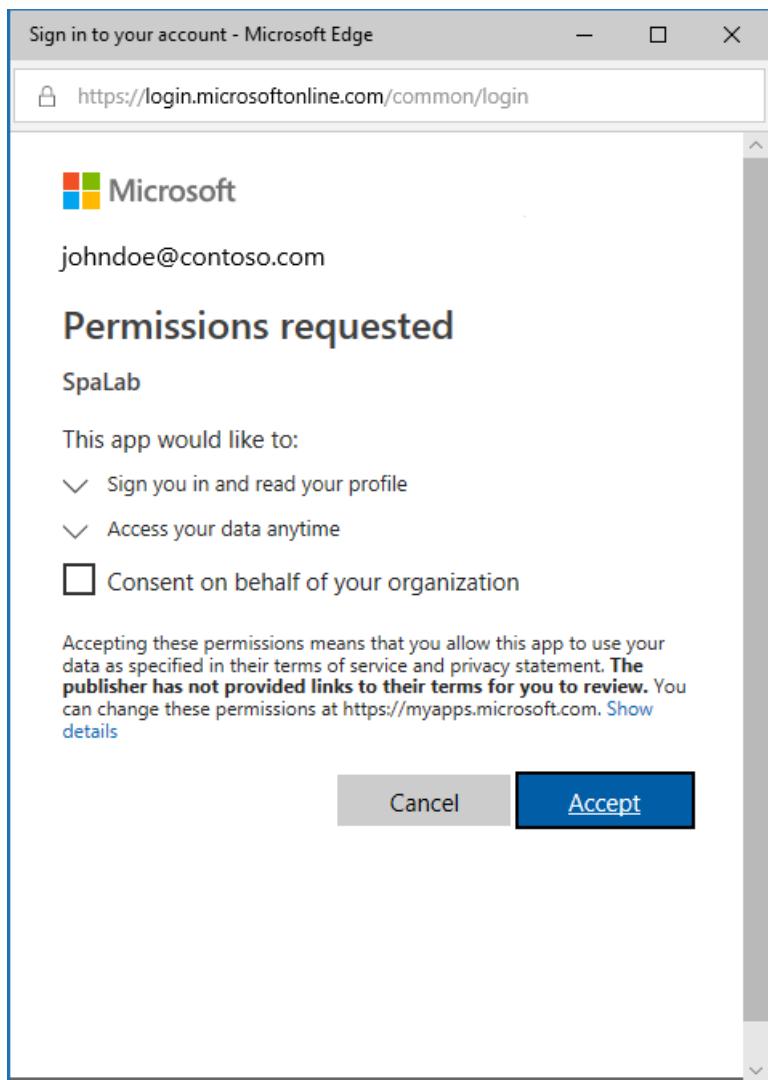
## Test your application

After the browser loads your `index.html` file, select **Sign In**. You're prompted to sign in with the Microsoft identity platform endpoint:



### Provide consent for application access

The first time that you sign in to your application, you're prompted to grant it access to your profile and sign you in:



## View application results

After you sign in, your user profile information is returned in the Microsoft Graph API response that's displayed:

```
Welcome John Doe to Microsoft Graph API!!

Sign Out

{
 "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users/$entity",
 "id": "1cd4bcac-b808-423a-9e2f-827fbb1bb739",
 "businessPhones": [],
 "displayName": "John Doe",
 "givenName": "John",
 "jobTitle": "Director",
 "mail": null,
 "mobilePhone": null,
 "officeLocation": null,
 "preferredLanguage": "en-US",
 "surname": "Doe",
 "userPrincipalName": "johndoe@contoso.com"
}
```

### More information about scopes and delegated permissions

The Microsoft Graph API requires the `user.read` scope to read a user's profile. By default, this scope is automatically added in every application that's registered on the registration portal. Other APIs for Microsoft Graph, as well as custom APIs for your back-end server, might require additional scopes. For example, the Microsoft Graph API requires the `Calendars.Read` scope in order to list the user's calendars.

To access the user's calendars in the context of an application, add the `Calendars.Read` delegated permission to the application registration information. Then, add the `Calendars.Read` scope to the `acquireTokenSilent` call.

**NOTE**

The user might be prompted for additional consents as you increase the number of scopes.

If a back-end API doesn't require a scope (not recommended), you can use *clientId* as the scope in the calls to acquire tokens.

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

### [Help and support for developers](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

### [Microsoft identity platform survey](#)

# Add sign-in to Microsoft to an ASP.NET web app

10/23/2019 • 14 minutes to read • [Edit Online](#)

This guide demonstrates how to implement sign-in to Microsoft through an ASP.NET MVC solution by using a traditional web browser-based application and OpenID Connect.

When you've completed this guide, your application will be able to accept sign-ins of personal accounts from the likes of outlook.com and live.com. Additionally, work and school accounts from any company or organization that's integrated with Azure Active Directory (Azure AD) will be able to sign in to your app.

This guide requires Microsoft Visual Studio 2019. Don't have it? [Download Visual Studio 2019 for free.](#)

## How the sample app generated by this guide works

The sample application you create is based on a scenario where you use the browser to access an ASP.NET website that prompts a user to authenticate through a sign-in button. In this scenario, most of the work to render the web page occurs on the server side.

## Libraries

This guide uses the following libraries:

LIBRARY	DESCRIPTION
<a href="#">Microsoft.Owin.Security.OpenIdConnect</a>	Middleware that enables an application to use OpenIdConnect for authentication
<a href="#">Microsoft.Owin.Security.Cookies</a>	Middleware that enables an application to maintain a user session by using cookies
<a href="#">Microsoft.Owin.Host.SystemWeb</a>	Middleware that enables OWIN-based applications to run on Internet Information Services (IIS) by using the ASP.NET request pipeline

## Set up your project

This section describes how to install and configure the authentication pipeline through OWIN middleware on an ASP.NET project by using OpenID Connect.

Prefer to download this sample's Visual Studio project instead? [Download a project](#) and skip to the [Register your application](#) to configure the code sample before executing.

### Create your ASP.NET project

1. In Visual Studio: Go to **File > New > Project**.
2. Under **Visual C#\Web**, select **ASP.NET Web Application (.NET Framework)**.
3. Name your application and select **OK**.
4. Select **Empty**, and then select the check box to add **MVC** references.

## Add authentication components

1. In Visual Studio: Go to **Tools > NuGet Package Manager > Package Manager Console**.
2. Add *OWIN middleware NuGet packages* by typing the following in the Package Manager Console window:

```
Install-Package Microsoft.Owin.Security.OpenIdConnect
Install-Package Microsoft.Owin.Security.Cookies
Install-Package Microsoft.Owin.Host.SystemWeb
```

### About these libraries

These libraries enable single sign-on (SSO) by using OpenID Connect through cookie-based authentication. After authentication is completed and the token representing the user is sent to your application, OWIN middleware creates a session cookie. The browser then uses this cookie on subsequent requests so that the user doesn't have to retype the password, and no additional verification is needed.

## Configure the authentication pipeline

The following steps are used to create an OWIN middleware Startup class to configure OpenID Connect authentication. This class is executed automatically when your IIS process starts.

### TIP

If your project doesn't have a `Startup.cs` file in the root folder:

1. Right-click the project's root folder, and then select **Add > New Item > OWIN Startup class**.
2. Name it **Startup.cs**.

Make sure the class selected is an OWIN Startup class and not a standard C# class. Confirm this by verifying that you see `[assembly: OwinStartup(typeof({NameSpace}.Startup))]` above the namespace.

1. Add *OWIN* and *Microsoft.IdentityModel* references to `Startup.cs`:

```
using Microsoft.Owin;
using Owin;
using Microsoft.IdentityModel.Protocols.OpenIdConnect;
using Microsoft.IdentityModel.Tokens;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.OpenIdConnect;
using Microsoft.Owin.Security.Notifications;
```

2. Replace Startup class with the following code:

```
public class Startup
{
 // The Client ID is used by the application to uniquely identify itself to Azure AD.
 string clientId = System.Configuration.ConfigurationManager.AppSettings["ClientId"];

 // RedirectUri is the URL where the user will be redirected to after they sign in.
 string redirectUri = System.Configuration.ConfigurationManager.AppSettings["RedirectUri"];

 // Tenant is the tenant ID (e.g. contoso.onmicrosoft.com, or 'common' for multi-tenant)
 static string tenant = System.Configuration.ConfigurationManager.AppSettings["Tenant"];

 // Authority is the URL for authority, composed by Azure Active Directory v2.0 endpoint and the
 // tenant name (e.g. https://login.microsoftonline.com/contoso.onmicrosoft.com/v2.0)
```

```

 string authority = String.Format(System.Globalization.CultureInfo.InvariantCulture,
System.Configuration.ConfigurationManager.AppSettings["Authority"], tenant);

 /// <summary>
 /// Configure OWIN to use OpenIdConnect
 /// </summary>
 /// <param name="app"></param>
 public void Configuration(IAppBuilder app)
 {
 app.SetDefaultSignInAsAuthenticationType(CookieAuthenticationDefaults.AuthenticationType);

 app.UseCookieAuthentication(new CookieAuthenticationOptions());
 app.UseOpenIdConnectAuthentication(
 new OpenIdConnectAuthenticationOptions
 {
 // Sets the ClientId, authority, RedirectUri as obtained from web.config
 ClientId = clientId,
 Authority = authority,
 RedirectUri = redirectUri,
 // PostLogoutRedirectUri is the page that users will be redirected to after sign-out. In
this case, it is using the home page
 PostLogoutRedirectUri = redirectUri,
 Scope = OpenIdConnectScope.OpenIdProfile,
 // ResponseType is set to request the id_token - which contains basic information about
the signed-in user
 ResponseType = OpenIdConnectResponseType.IdToken,
 // ValidateIssuer set to false to allow personal and work accounts from any organization
to sign in to your application
 // To only allow users from a single organizations, set ValidateIssuer to true and
'tenant' setting in web.config to the tenant name
 // To allow users from only a list of specific organizations, set ValidateIssuer to true
and use ValidIssuers parameter
 TokenValidationParameters = new TokenValidationParameters()
 {
 ValidateIssuer = false // This is a simplification
 },
 // OpenIdConnectAuthenticationNotifications configures OWIN to send notification of
failed authentications to OnAuthenticationFailed method
 Notifications = new OpenIdConnectAuthenticationNotifications
 {
 AuthenticationFailed = OnAuthenticationFailed
 }
 }
);
 }

 /// <summary>
 /// Handle failed authentication requests by redirecting the user to the home page with an error in
the query string
 /// </summary>
 /// <param name="context"></param>
 /// <returns></returns>
 private Task OnAuthenticationFailed(AuthenticationFailedNotification<OpenIdConnectMessage,
OpenIdConnectAuthenticationOptions> context)
 {
 context.HandleResponse();
 context.Response.Redirect("/?errormessage=" + context.Exception.Message);
 return Task.FromResult(0);
 }
}

```

#### NOTE

Setting `ValidateIssuer = false` is a simplification for this quickstart. In real applications, you must validate the issuer. See the samples to learn how to do that.

## More information

The parameters you provide in `OpenIDConnectAuthenticationOptions` serve as coordinates for the application to communicate with Azure AD. Because the OpenID Connect middleware uses cookies in the background, you must also set up cookie authentication as the preceding code shows. The `ValidateIssuer` value tells OpenIdConnect not to restrict access to one specific organization.

## Add a controller to handle sign-in and sign-out requests

To create a new controller to expose sign-in and sign-out methods, follow these steps:

1. Right-click the **Controllers** folder and select **Add > Controller**.
2. Select **MVC (.NET version) Controller – Empty**.
3. Select **Add**.
4. Name it **HomeController** and then select **Add**.
5. Add OWIN references to the class:

```
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.OpenIdConnect;
```

6. Add the following two methods to handle sign-in and sign-out to your controller by initiating an authentication challenge:

```
/// <summary>
/// Send an OpenID Connect sign-in request.
/// Alternatively, you can just decorate the SignIn method with the [Authorize] attribute
/// </summary>
public void SignIn()
{
 if (!Request.IsAuthenticated)
 {
 HttpContext.GetOwinContext().Authentication.Challenge(
 new AuthenticationProperties{ RedirectUri = "/" },
 OpenIdConnectAuthenticationDefaults.AuthenticationType);
 }
}

/// <summary>
/// Send an OpenID Connect sign-out request.
/// </summary>
public void SignOut()
{
 HttpContext.GetOwinContext().Authentication.SignOut(
 OpenIdConnectAuthenticationDefaults.AuthenticationType,
 CookieAuthenticationDefaults.AuthenticationType);
}
```

## Create the app's home page for user sign-in

In Visual Studio, create a new view to add the sign-in button and to display user information after authentication:

1. Right-click the **Views\Home** folder and select **Add View**.
2. Name the new view **Index**.
3. Add the following HTML, which includes the sign-in button, to the file:

```

<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>Sign in with Microsoft Guide</title>
</head>
<body>
@if (!Request.IsAuthenticated)
{
 <!-- If the user is not authenticated, display the sign-in button -->

 <svg xmlns="http://www.w3.org/2000/svg" xml:space="preserve" width="300px" height="50px"
viewBox="0 0 3278 522" class="SignInButton">
 <style type="text/css">.fil0:hover {fill: #4B4B4B;} .fnt0 {font-size: 260px;font-family: 'Segoe
UI Semibold', 'Segoe UI'; text-decoration: none;}</style>
 <rect class="fil0" x="2" y="2" width="3174" height="517" fill="black" />
 <rect x="150" y="129" width="122" height="122" fill="#F35325" />
 <rect x="284" y="129" width="122" height="122" fill="#81BC06" />
 <rect x="150" y="263" width="122" height="122" fill="#05A6F0" />
 <rect x="284" y="263" width="122" height="122" fill="#FFBA08" />
 <text x="470" y="357" fill="white" class="fnt0">Sign in with Microsoft</text>
 </svg>

}
else
{

Hello @System.Security.Claims.ClaimsPrincipal.Current.FindFirst("name").Value;

 @Html.ActionLink("See Your Claims", "Index", "Claims")

 @Html.ActionLink("Sign out", "SignOut", "Home")
}
@if (!string.IsNullOrWhiteSpace(Request.QueryString["errormessage"]))
{
 <div style="background-color:red;color:white;font-weight: bold;">Error:
 @Request.QueryString["errormessage"]</div>
}
</body>
</html>

```

## More information

This page adds a sign-in button in SVG format with a black background:



For more sign-in buttons, go to the [Branding guidelines](#).

## Add a controller to display user's claims

This controller demonstrates the uses of the `[Authorize]` attribute to protect a controller. This attribute restricts access to the controller by allowing only authenticated users. The following code makes use of the attribute to display user claims that were retrieved as part of sign-in:

1. Right-click the **Controllers** folder, and then select **Add > Controller**.
2. Select **MVC {version} Controller – Empty**.
3. Select **Add**.
4. Name it **ClaimsController**.

5. Replace the code of your controller class with the following code. This adds the `[Authorize]` attribute to the class:

```
[Authorize]
public class ClaimsController : Controller
{
 /// <summary>
 /// Add user's claims to viewbag
 /// </summary>
 /// <returns></returns>
 public ActionResult Index()
 {
 var userClaims = User.Identity as System.Security.Claims.ClaimsIdentity;

 //You get the user's first and last name below:
 ViewBag.Name = userClaims?.FindFirst("name")?.Value;

 // The 'preferred_username' claim can be used for showing the username
 ViewBag.Username = userClaims?.FindFirst("preferred_username")?.Value;

 // The subject/ NameIdentifier claim can be used to uniquely identify the user across the web
 ViewBag.Subject =
 userClaims?.FindFirst(System.Security.Claims.ClaimTypes.NameIdentifier)?.Value;

 // TenantId is the unique Tenant Id - which represents an organization in Azure AD
 ViewBag.TenantId =
 userClaims?.FindFirst("http://schemas.microsoft.com/identity/claims/tenantid")?.Value;

 return View();
 }
}
```

## More information

Because of the use of the `[Authorize]` attribute, all methods of this controller can be executed only if the user is authenticated. If the user isn't authenticated and tries to access the controller, OWIN initiates an authentication challenge and forces the user to authenticate. The preceding code looks at the list of claims for specific user attributes included in the user's Id token. These attributes include the user's full name and username, as well as the global user identifier subject. It also contains the *Tenant ID*, which represents the ID for the user's organization.

## Create a view to display the user's claims

In Visual Studio, create a new view to display the user's claims in a web page:

1. Right-click the **Views\Claims** folder, and then select **Add View**.
2. Name the new view **Index**.
3. Add the following HTML to the file:

```

<html>
<head>
 <meta name="viewport" content="width=device-width" />
 <title>Sign in with Microsoft Sample</title>
 <link href="@Url.Content("~/Content/bootstrap.min.css")" rel="stylesheet" type="text/css" />
</head>
<body style="padding:50px">
 <h3>Main Claims:</h3>
 <table class="table table-striped table-bordered table-hover">
 <tr><td>Name</td><td>@ViewBag.Name</td></tr>
 <tr><td>Username</td><td>@ViewBag.Username</td></tr>
 <tr><td>Subject</td><td>@ViewBag.Subject</td></tr>
 <tr><td>TenantId</td><td>@ViewBag.TenantId</td></tr>
 </table>

 <h3>All Claims:</h3>
 <table class="table table-striped table-bordered table-hover table-condensed">
 @foreach (var claim in System.Security.Claims.ClaimsPrincipal.Current.Claims)
 {
 <tr><td>@claim.Type</td><td>@claim.Value</td></tr>
 }
 </table>

 @Html.ActionLink("Sign out", "SignOut", "Home", null, new { @class = "btn btn-primary" })
</body>
</html>

```

## Register your application

To register your application and add your application registration information to your solution, you have two options:

### Option 1: Express mode

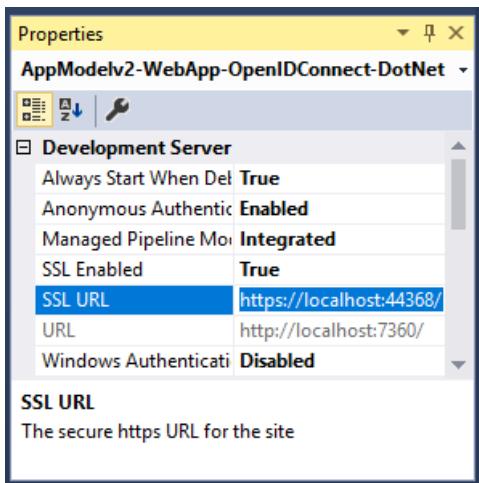
To quickly register your application, follow these steps:

1. Go to the new [Azure portal - App registrations](#) pane.
2. Enter a name for your application and select **Register**.
3. Follow the instructions to download and automatically configure your new application in a single click.

### Option 2: Advanced mode

To register your application and add the app's registration information to your solution manually, follow these steps:

1. Open Visual Studio, and then:
  - a. in Solution Explorer, select the project and view the Properties window (if you don't see a Properties window, press F4).
  - b. Change SSL Enabled to `True`.
  - c. Right-click the project in Visual Studio, select **Properties**, and then select the **Web** tab. In the **Servers** section, change the **Project Url** setting to the **SSL URL**.
  - d. Copy the SSL URL. You'll add this URL to the list of Redirect URLs in the Registration portal's list of Redirect URLs in the next step.



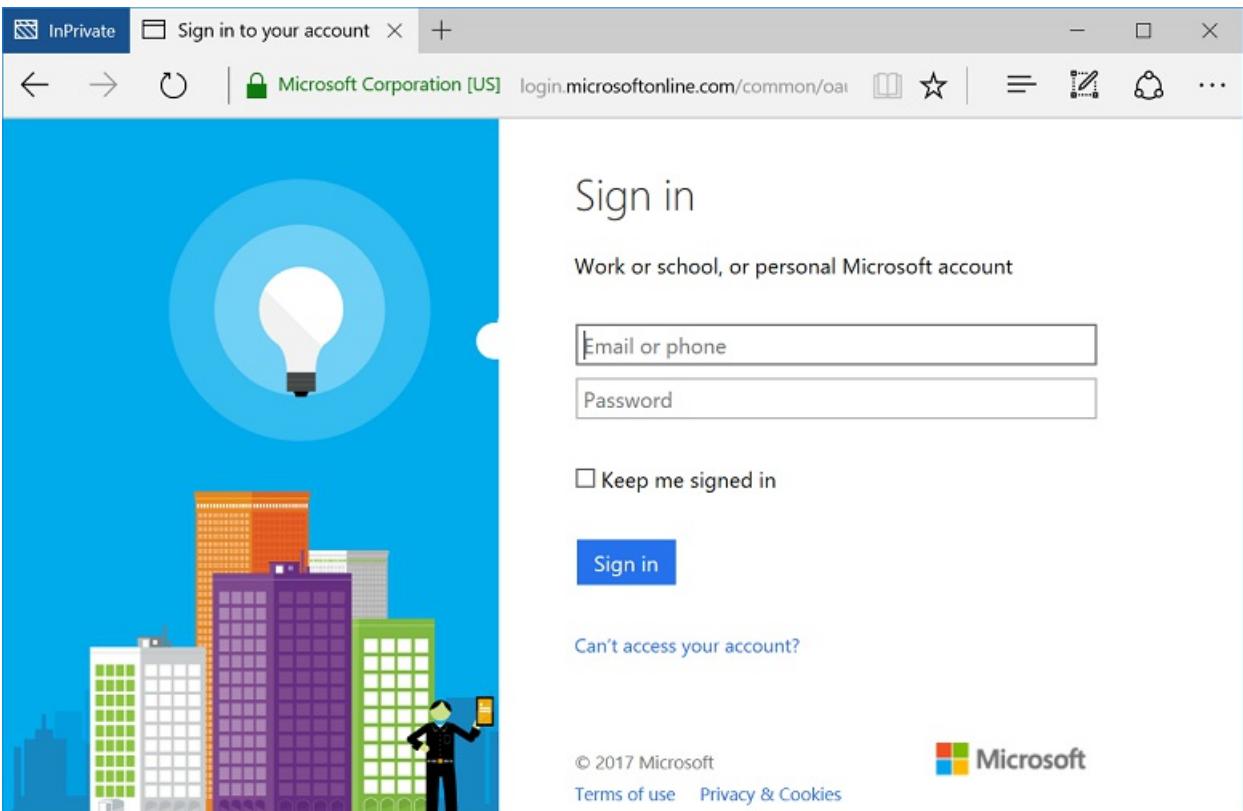
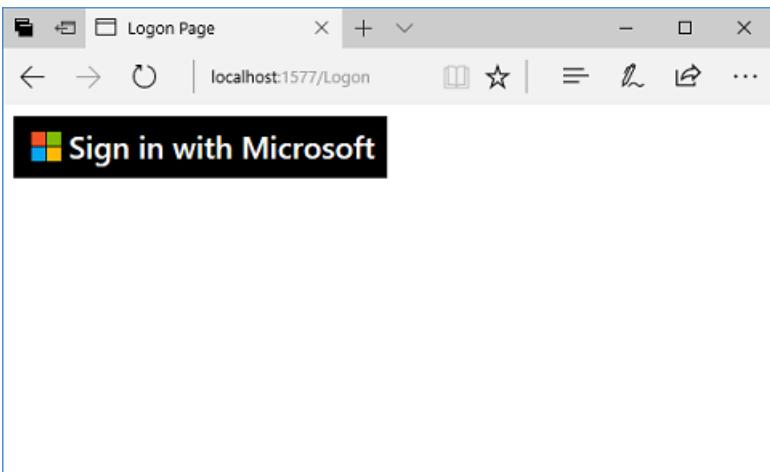
2. Sign in to the [Azure portal](#) by using a work or school account, or by using a personal Microsoft account.
3. If your account gives you access to more than one tenant, select your account in the upper-right corner, and set your portal session to the Azure AD tenant that you want.
4. Go to the Microsoft identity platform for developers [App registrations](#) page.
5. Select **New registration**.
6. When the **Register an application** page appears, enter your application's registration information:
  - a. In the **Name** section, enter a meaningful application name that will be displayed to users of the app, like **ASPNET-Tutorial**.
  - b. Add the SSL URL you copied from Visual Studio in step 1 (for example, `https://localhost:44368/`) in **Reply URL**, and select **Register**.
7. Select the **Authentication** menu, select **ID tokens** under **Implicit Grant**, and then select **Save**.
8. Add the following in the web.config file, located in the root folder in the `configuration\appSettings` section:

```
<add key="ClientId" value="Enter_the_Application_Id_here" />
<add key="redirectUri" value="Enter_the_Redirect_URL_here" />
<add key="Tenant" value="common" />
<add key="Authority" value="https://login.microsoftonline.com/{0}/v2.0" />
```
9. Replace `ClientId` with the Application ID you just registered.
10. Replace `redirectUri` with the SSL URL of your project.

## Test your code

To test your application in Visual Studio, press F5 to run your project. The browser opens to the `http://localhost:{port}` location, and you see the **Sign in with Microsoft** button. Select the button to start the sign-in process.

When you're ready to run your test, use an Azure AD account (work or school account) or a personal Microsoft account (live.com or outlook.com) to sign in.



## Permissions and consent in the Microsoft identity platform endpoint

Applications that integrate with Microsoft identity platform follow an authorization model that gives users and administrators control over how data can be accessed. After a user authenticates with Azure AD to access this application, they will be prompted to consent to the permissions requested by the application ("View your basic profile" and "Maintain access to data you have given it access to"). After accepting these permissions, the user will continue on to the application results. However, the user may instead be prompted with a **Need admin consent** page if either of the following occur:

- The application developer adds any additional permissions that require **Admin consent**.
- Or the tenant is configured (in **Enterprise Applications -> User Settings**) where users cannot consent to apps accessing company data on their behalf.

For more information, refer to [Permissions and consent in the Microsoft identity platform endpoint](#).

## View application results

After you sign in, the user is redirected to the home page of your website. The home page is the HTTPS URL that's specified in your application registration info in the Microsoft Application Registration Portal. The home page

includes a "Hello <user>" welcome message, a link to sign out, and a link to view the user's claims. The link for the user's claims connects to the Claims controller that you created earlier.

## View the user's claims

To view the user's claims, select the link to browse to the controller view that's available only to authenticated users.

### View the claims results

After you browse to the controller view, you should see a table that contains the basic properties for the user:

PROPERTY	VALUE	DESCRIPTION
Name	User's full name	The user's first and last name
Username	user@domain.com	The username that's used to identify the user
Subject	Subject	A string that uniquely identifies the user across the web
Tenant ID	Guid	A <b>guid</b> that uniquely represents the user's Azure AD organization

Additionally, you should see a table of all claims that are in the authentication request. For more information, see the [list of claims that are in an Azure AD ID token](#).

## Test access to a method that has an Authorize attribute (optional)

To test access as an anonymous user to a controller that's protected by the `Authorize` attribute, follow these steps:

1. Select the link to sign out the user, and complete the sign-out process.
2. In your browser, type `http://localhost:{port}/claims` to access your controller that's protected by the `Authorize` attribute.

### Expected results after access to a protected controller

You're prompted to authenticate to use the protected controller view.

## Advanced options

### Protect your entire website

To protect your entire website, in the **Global.asax** file, add the `AuthorizeAttribute` attribute to the `GlobalFilters` filter in the `Application_Start` method:

```
GlobalFilters.Filters.Add(new AuthorizeAttribute());
```

### Restrict who can sign in to your application

By default when you build the application created by this guide, your application will accept sign-ins of personal accounts (including outlook.com, live.com, and others) as well as work and school accounts from any company or organization that's integrated with Azure AD. This is a recommended option for SaaS applications.

To restrict user sign-in access for your application, multiple options are available.

#### Option 1: Restrict users from only one organization's Active Directory instance to sign in to your application (single-tenant)

This option is frequently used for *LOB applications*: If you want your application to accept sign-ins only from accounts that belong to a specific Azure AD instance (including *guest accounts* of that instance), follow these steps:

1. In the `web.config` file, change the value for the `Tenant` parameter from `Common` to the tenant name of the

organization, such as `contoso.onmicrosoft.com`.

2. In your [OWIN Startup class](#), set the `ValidateIssuer` argument to `true`.

#### **Option 2: Restrict access to users in a specific list of organizations**

You can restrict sign-in access to only those user accounts that are in an Azure AD organization that's on the list of allowed organizations:

1. In your [OWIN Startup class](#), set the `ValidateIssuer` argument to `true`.
2. Set the value of the `ValidIssuers` parameter to the list of allowed organizations.

#### **Option 3: Use a custom method to validate issuers**

You can implement a custom method to validate issuers by using the **IssuerValidator** parameter. For more information about how to use this parameter, see [TokenValidationParameters class](#).

## Next steps

Learn about how web apps can call web APIs.

### **Learn how to create the application used in this quickstart guide**

Learn more about Web apps calling web APIs with the Microsoft identity platform:

#### [Web apps calling Web APIs](#)

Learn how to build Web apps calling Microsoft Graph:

#### [Microsoft Graph ASP.NET tutorial](#)

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

#### [Help and support for developers](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a two-question survey:

#### [Microsoft identity platform survey](#)

# Tutorial: Sign in users and call the Microsoft Graph from an Android app

10/31/2019 • 12 minutes to read • [Edit Online](#)

## NOTE

This tutorial has not yet been updated to work with MSAL for Android version 1.0 library. It works with an earlier version, as configured in this tutorial.

In this tutorial, you'll learn how to integrate an Android app with the Microsoft identity platform. Your app will sign in a user, get an access token to call the Microsoft Graph API, and make a request to the Microsoft Graph API.

- Integrate an Android app with the Microsoft identity platform
- Sign in a user
- Get an access token to call the Microsoft Graph API
- Call the Microsoft Graph API.

When you've completed this tutorial, your application will accept sign-ins of personal Microsoft accounts (including outlook.com, live.com, and others) as well as work or school accounts from any company or organization that uses Azure Active Directory.

If you don't have an Azure subscription, create a [free account](#) before you begin.

## How this tutorial works

The app in this tutorial will sign in users and get data on their behalf. This data will be accessed through a protected API (Microsoft Graph API) that requires authorization and is protected by the Microsoft identity platform.

More specifically:

- Your app will sign in the user either through a browser or the Microsoft Authenticator and Intune Company Portal.
- The end user will accept the permissions your application has requested.
- Your app will be issued an access token for the Microsoft Graph API.
- The access token will be included in the HTTP request to the web API.
- Process the Microsoft Graph response.

This sample uses the Microsoft Authentication library for Android (MSAL) to implement Authentication: [com.microsoft.identity.client](#).

MSAL will automatically renew tokens, deliver single sign-on (SSO) between other apps on the device, and manage the Account(s).

## Prerequisites

- This tutorial requires Android Studio version 3.5.

# Create a project

This tutorial will create a new project. If you want to download the completed tutorial instead, [download the code](#).

1. Open Android Studio, and select **Start a new Android Studio project**.
2. Select **Basic Activity** and select **Next**.
3. Name your application.
4. Save the package name. You will enter it later into the Azure portal.
5. Change the language from **Kotlin** to **Java**.
6. Set the **Minimum API level** to **API 19** or higher, and click **Finish**.
7. In the project view, choose **Project** in the dropdown to display source and non-source project files, open **app/build.gradle** and set `targetSdkVersion` to `28`.

## Register your application

1. Go to the [Azure portal](#).
2. Open the [App registrations blade](#) and click **+New registration**.
3. Enter a **Name** for your app and then, without setting a Redirect URI, click **Register**.
4. In the **Manage** section of the pane that appears, select **Authentication > + Add a platform > Android**.  
(You may have to select "Switch to the new experience" near the top of the blade to see this section)
5. Enter your project's Package Name. If you downloaded the code, this value is  
`com.azure.samples.msalandroidapp`.
6. In the **Signature hash** section of the [Configure your Android app](#) page, click **Generating a development Signature Hash**. and copy the KeyTool command to use for your platform.

### NOTE

KeyTool.exe is installed as part of the Java Development Kit (JDK). You must also install the OpenSSL tool to execute the KeyTool command.

7. Enter the **Signature hash** generated by KeyTool.
8. Click **Configure** and save the **MSAL Configuration** that appears in the **Android configuration** page so you can enter it when you configure your app later. Click **Done**.

## Build your app

### Add your app registration

1. In Android Studio's project pane, navigate to **app\src\main\res**.
2. Right-click **res** and choose **New > Directory**. Enter `raw` as the new directory name and click **OK**.
3. In **app > src > main > res > raw**, create a new JSON file called `auth_config.json` and paste the MSAL Configuration that you saved earlier. See [MSAL Configuration for more info](#).
4. In **app > src > main > AndroidManifest.xml**, add the `BrowserTabActivity` activity below to the application body. This entry allows Microsoft to call back to your application after it completes the authentication:

```
<!--Intent filter to capture System Browser or Authenticator calling back to our app after sign-in-->
<activity
 android:name="com.microsoft.identity.client.BrowserTabActivity">
 <intent-filter>
 <action android:name="android.intent.action.VIEW" />
 <category android:name="android.intent.category.DEFAULT" />
 <category android:name="android.intent.category.BROWSABLE" />
 <data android:scheme="msauth"
 android:host="Enter_the_Package_Name"
 android:path="/Enter_the_Signature_Hash" />
 </intent-filter>
</activity>
```

Substitute the package name you registered in the Azure portal for the `android:host=` value. Substitute the key hash you registered in the Azure portal for the `android:path=` value. The Signature Hash should not be URL encoded.

5. Inside the **AndroidManifest.xml**, just above the `<application>` tag, add the following permissions:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

## Create the app's UI

1. In the Android Studio project window, navigate to **app > src > main > res > layout** and open **activity\_main.xml** and open the **Text** view.
2. Change the activity layout, for example: `<androidx.coordinatorlayout.widget.CoordinatorLayout` to `<androidx.coordinatorlayout.widget.DrawerLayout`.
3. Add the `android:orientation="vertical"` property to the `LinearLayout` node.
4. Paste the following code into the `LinearLayout` node, replacing the current content:

```

<TextView
 android:text="Welcome, "
 android:textColor="#3f3f3f"
 android:textSize="50px"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_marginLeft="10dp"
 android:layout_marginTop="15dp"
 android:id="@+id/welcome"
 android:visibility="invisible"/>

<Button
 android:id="@+id/callGraph"
 android:text="Call Microsoft Graph"
 android:textColor="#FFFFFF"
 android:background="#00a1f1"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_marginTop="200dp"
 android:textAllCaps="false" />

<TextView
 android:text="Getting Graph Data..."
 android:textColor="#3f3f3f"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_marginLeft="5dp"
 android:id="@+id/graphData"
 android:visibility="invisible"/>

<LinearLayout
 android:layout_width="match_parent"
 android:layout_height="0dip"
 android:layout_weight="1"
 android:gravity="center|bottom"
 android:orientation="vertical" >

 <Button
 android:text="Sign Out"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_marginBottom="15dp"
 android:textColor="#FFFFFF"
 android:background="#00a1f1"
 android:textAllCaps="false"
 android:id="@+id/clearCache"
 android:visibility="invisible" />
</LinearLayout>
```

## Add MSAL to your project

1. In the Android Studio project window, navigate to **app > src > build.gradle**.
2. Under **Dependencies**, paste the following:

```
implementation 'com.android.volley:volley:1.1.1'
implementation 'com.microsoft.identity.client:msal:0.3+'
```

## Use MSAL

Now make changes inside `MainActivity.java` to add and use MSAL in your app. In the Android Studio project window, navigate to **app > src > main > java > com.example.(your app)**, and open `MainActivity.java`.

### Required imports

Add the following imports near the top of `MainActivity.java`:

```
import android.app.Activity;
import android.content.Intent;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
import com.android.volley.*;
import com.android.volley.toolbox.JsonObjectRequest;
import com.android.volley.toolbox.Volley;
import org.json.JSONObject;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import com.microsoft.identity.client.*;
import com.microsoft.identity.client.exception.*;
```

#### Instantiate MSAL

Inside the `MainActivity` class, we'll need to instantiate MSAL along with a few configurations about what our app will do including the scopes and web API we want to access.

Copy the following variables inside the `MainActivity` class:

```
final static String SCOPES [] = {"https://graph.microsoft.com/User.Read"};
final static String MSGRAPH_URL = "https://graph.microsoft.com/v1.0/me";

/* UI & Debugging Variables */
private static final String TAG = MainActivity.class.getSimpleName();
Button callGraphButton;
Button signOutButton;

/* Azure AD Variables */
private PublicClientApplication sampleApp;
private IAuthenticationResult authResult;
```

Replace the contents of `onCreate()` with the following code to instantiate MSAL:

```

super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);

callGraphButton = (Button) findViewById(R.id.callGraph);
signOutButton = (Button) findViewById(R.id.clearCache);

callGraphButton.setOnClickListener(new View.OnClickListener() {
 public void onClick(View v) {
 onCallGraphClicked();
 }
});

signOutButton.setOnClickListener(new View.OnClickListener() {
 public void onClick(View v) {
 onSignOutClicked();
 }
});

/* Configure your sample app and save state for this activity */
sampleApp = new PublicClientApplication(
 this.getApplicationContext(),
 R.raw.auth_config);

/* Attempt to get a user and acquireTokenSilent */
sampleApp.getAccounts(new PublicClientApplication.AccountsLoadedCallback() {
 @Override
 public void onAccountsLoaded(final List<IAccount> accounts) {
 if (!accounts.isEmpty()) {
 /* This sample doesn't support multi-account scenarios, use the first account */
 sampleApp.acquireTokenSilentAsync(SCOPES, accounts.get(0), getAuthSilentCallback());
 } else {
 /* No accounts */
 }
 }
});

```

The code above attempts to sign in users silently when they open your application through `getAccounts()` and, if successful, `acquireTokenSilentAsync()`. In the next few sections, we'll implement the callback handler for the case there are no signed in accounts.

#### Use MSAL to get Tokens

Now, we can implement the app's UI processing logic and getting tokens interactively through MSAL.

MSAL exposes two primary methods for getting tokens: `acquireTokenSilentAsync()` and `acquireToken()`.

`acquireTokenSilentAsync()` signs in a user and get tokens without any user interaction if an account is present. If it succeeds, MSAL will handoff the tokens to your app, if it fails it will generate a `MsalUiRequiredException`. If this exception is generated, or you want the user to have an interactive sign in experience (credentials, mfa, or other Conditional Access policies may or may not be required), then use `acquireToken()`.

`acquireToken()` displays UI when attempting to sign in the user and get tokens. However, it may use session cookies in the browser, or an account in the Microsoft authenticator, to provide the interactive-SSO experience.

Create the following three UI methods inside the `MainActivity` class:

```

/* Set the UI for successful token acquisition data */
private void updateSuccessUI() {
 callGraphButton.setVisibility(View.INVISIBLE);
 signOutButton.setVisibility(View.VISIBLE);
 findViewById(R.id.welcome).setVisibility(View.VISIBLE);
 ((TextView) findViewById(R.id.welcome)).setText("Welcome, " +
 authResult.getAccount().getUsername());
 findViewById(R.id.graphData).setVisibility(View.VISIBLE);
}

/* Set the UI for signed out account */
private void updateSignedOutUI() {
 callGraphButton.setVisibility(View.VISIBLE);
 signOutButton.setVisibility(View.INVISIBLE);
 findViewById(R.id.welcome).setVisibility(View.INVISIBLE);
 findViewById(R.id.graphData).setVisibility(View.INVISIBLE);
 ((TextView) findViewById(R.id.graphData)).setText("No Data");

 Toast.makeText(getApplicationContext(), "Signed Out!", Toast.LENGTH_SHORT)
 .show();
}

/* Use MSAL to acquireToken for the end-user
 * Callback will call Graph api w/ access token & update UI
 */
private void onCallGraphClicked() {
 sampleApp.acquireToken(getActivity(), SCOPES, getAuthInteractiveCallback());
}

```

Add the following methods to get the current activity and process silent & interactive callbacks:

```

public Activity getActivity() {
 return this;
}

/* Callback used in for silent acquireToken calls.
 * Looks if tokens are in the cache (refreshes if necessary and if we don't forceRefresh)
 * else errors that we need to do an interactive request.
 */
private AuthenticationCallback getAuthSilentCallback() {
 return new AuthenticationCallback() {

 @Override
 public void onSuccess(IAuthenticationResult authenticationResult) {
 /* Successfully got a token, call graph now */
 Log.d(TAG, "Successfully authenticated");

 /* Store the authResult */
 authResult = authenticationResult;

 /* call graph */
 callGraphAPI();

 /* update the UI to post call graph state */
 updateSuccessUI();
 }

 @Override
 public void onError(MsalException exception) {
 /* Failed to acquireToken */
 Log.d(TAG, "Authentication failed: " + exception.toString());

 if (exception instanceof MsalClientException) {
 /* Exception inside MSAL, more info inside the exception */
 } else if (exception instanceof MsalServiceException) {
 /* Exception when communicating with the STS, likely config issue */
 }
 }
 };
}

```

```

 } else if (exception instanceof MsalUiRequiredException) {
 /* Tokens expired or no session, retry with interactive */
 }
 }

 @Override
 public void onCancel() {
 /* User cancelled the authentication */
 Log.d(TAG, "User cancelled login.");
 }
};

/* Callback used for interactive request. If succeeds we use the access
 * token to call the Microsoft Graph. Does not check cache
 */
private AuthenticationCallback getAuthInteractiveCallback() {
 return new AuthenticationCallback() {

 @Override
 public void onSuccess(IAuthenticationResult authenticationResult) {
 /* Successfully got a token, call graph now */
 Log.d(TAG, "Successfully authenticated");
 Log.d(TAG, "ID Token: " + authenticationResult.getIdToken());

 /* Store the auth result */
 authResult = authenticationResult;

 /* call graph */
 callGraphAPI();

 /* update the UI to post call graph state */
 updateSuccessUI();
 }

 @Override
 public void onError(MsalException exception) {
 /* Failed to acquireToken */
 Log.d(TAG, "Authentication failed: " + exception.toString());

 if (exception instanceof MsalClientException) {
 /* Exception inside MSAL, more info inside the exception */
 } else if (exception instanceof MsalServiceException) {
 /* Exception when communicating with the STS, likely config issue */
 }
 }

 @Override
 public void onCancel() {
 /* User cancelled the authentication */
 Log.d(TAG, "User cancelled login.");
 }
 };
}

```

#### Use MSAL for Sign-out

Next, add support for sign-out.

#### IMPORTANT

Signing out with MSAL removes all known information about a user from the application, but the user will still have an active session on their device. If the user attempts to sign in again they may see sign-in UI, but may not need to reenter their credentials because the device session is still active.

To add sign-out capability, add the following method inside the `MainActivity` class. This method cycles through all

accounts and removes them:

```
/* Clears an account's tokens from the cache.
 * Logically similar to "sign out" but only signs out of this app.
 * User will get interactive SSO if trying to sign back-in.
 */
private void onSignOutClicked() {
 /* Attempt to get a user and acquireTokenSilent
 * If this fails we do an interactive request
 */
 sampleApp.getAccounts(new PublicClientApplication.AccountsLoadedCallback() {
 @Override
 public void onAccountsLoaded(final List<IAccount> accounts) {

 if (accounts.isEmpty()) {
 /* No accounts to remove */

 } else {
 for (final IAccount account : accounts) {
 sampleApp.removeAccount(
 account,
 new PublicClientApplication.AccountsRemovedCallback() {
 @Override
 public void onAccountsRemoved(Boolean isSuccess) {
 if (isSuccess) {
 /* successfully removed account */
 } else {
 /* failed to remove account */
 }
 }
 });
 }
 }
 updateSignedOutUI();
 }
 });
}
```

#### Call the Microsoft Graph API

Once we have received a token, we can make a request to the [Microsoft Graph API](#). The access token will be inside the `AuthenticationResult` inside the auth callback's `onSuccess()` method. To construct an authorized request, your app will need to add the access token to the HTTP header:

HEADER KEY	VALUE
Authorization	Bearer <access-token>

Add the following two methods inside the `MainActivity` class to call graph and update the UI:

```

/* Use Volley to make an HTTP request to the /me endpoint from MS Graph using an access token */
private void callGraphAPI() {
 Log.d(TAG, "Starting volley request to graph");

 /* Make sure we have a token to send to graph */
 if (authResult.getAccessToken() == null) {return;}

 RequestQueue queue = Volley.newRequestQueue(this);
 JSONObject parameters = new JSONObject();

 try {
 parameters.put("key", "value");
 } catch (Exception e) {
 Log.d(TAG, "Failed to put parameters: " + e.toString());
 }
 JsonObjectRequest request = new JsonObjectRequest(Request.Method.GET, MSGRAPH_URL,
 parameters,new Response.Listener<JSONObject>() {
 @Override
 public void onResponse(JSONObject response) {
 /* Successfully called graph, process data and send to UI */
 Log.d(TAG, "Response: " + response.toString());

 updateGraphUI(response);
 }
 }, new Response.ErrorListener() {
 @Override
 public void onErrorResponse(VolleyError error) {
 Log.d(TAG, "Error: " + error.toString());
 }
 });
 {
 @Override
 public Map<String, String> getHeaders() {
 Map<String, String> headers = new HashMap<>();
 headers.put("Authorization", "Bearer " + authResult.getAccessToken());
 return headers;
 }
 };
}

Log.d(TAG, "Adding HTTP GET to Queue, Request: " + request.toString());

request.setRetryPolicy(new DefaultRetryPolicy(
 3000,
 DefaultRetryPolicy.DEFAULT_MAX_RETRIES,
 DefaultRetryPolicy.DEFAULT_BACKOFF_MULT));
queue.add(request);
}

/* Sets the graph response */
private void updateGraphUI(JSONObject graphResponse) {
 TextView graphText = findViewById(R.id.graphData);
 graphText.setText(graphResponse.toString());
}

```

#### Multi-account applications

This app is built for a single account scenario. MSAL also supports multi-account scenarios, but it requires some additional work from apps. You will need to create UI to help user's select which account they want to use for each action that requires tokens. Alternatively, your app can implement a heuristic to select which account to use via the `getAccounts()` method.

## Test your app

### Run locally

Build and deploy the app to a test device or emulator. You should be able to sign in and get tokens for Azure AD or

personal Microsoft accounts.

After you sign in, the app will display the data returned from the Microsoft Graph `/me` endpoint.

## Consent

The first time any user signs into your app, they will be prompted by Microsoft identity to consent to the permissions requested. While most users are capable of consenting, some Azure AD tenants have disabled user consent which requires admins to consent on behalf of all users. To support this scenario, register your app's scopes in the Azure portal.

## Clean up resources

When no longer needed, delete the app object that you created in the [Register your application](#) step.

## Get help

Visit [Help and support](#) if you have trouble with this tutorial or with the Microsoft identity platform.

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

# Sign in users and call the Microsoft Graph from an iOS or macOS app

11/12/2019 • 13 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to integrate an iOS or macOS app with the Microsoft identity platform. The app will sign in a user, get an access token to call the Microsoft Graph API, and make a request to the Microsoft Graph API.

When you've completed the guide, your application will accept sign-ins of personal Microsoft accounts (including outlook.com, live.com, and others) and work or school accounts from any company or organization that uses Azure Active Directory.

## How this tutorial works

The app in this tutorial will sign in users and get data on their behalf. This data will be accessed via a protected API (Microsoft Graph API in this case) that requires authorization and is protected by the Microsoft identity platform.

More specifically:

- Your app will sign in the user either through a browser or the Microsoft Authenticator.
- The end user will accept the permissions your application has requested.
- Your app will be issued an access token for the Microsoft Graph API.
- The access token will be included in the HTTP request to the web API.
- Process the Microsoft Graph response.

This sample uses the Microsoft Authentication library (MSAL) to implement Authentication. MSAL will automatically renew tokens, deliver single sign-on (SSO) between other apps on the device, and manage the Account(s).

This tutorial is applicable to both iOS and macOS apps. Note that some steps are different between those two platforms.

## Prerequisites

- XCode version 10.x or greater is required to build the app in this guide. You can download XCode from the [iTunes website](#).
- Microsoft Authentication Library ([MSAL.framework](#)). You can use a dependency manager or add the library manually. The instructions below show you how.

This tutorial will create a new project. If you want to download the completed tutorial instead, download the code:

- [iOS sample code](#)
- [macOS sample code](#)

## Create a new project

1. Open Xcode and select **Create a new Xcode project**.
2. For iOS apps, select **iOS > Single view App** and select **Next**.
3. For macOS apps, select **macOS > Cocoa App** and select **Next**.

4. Provide a product name.
5. Set the **Language** to **Swift** and select **Next**.
6. Select a folder to create your app and click **Create**.

## Register your application

1. Go to the [Azure portal](#)
2. Open the [App registrations blade](#) and click **+New registration**.
3. Enter a **Name** for your app and then, without setting a Redirect URI, click **Register**.
4. In the **Manage** section of the pane that appears, select **Authentication**.
5. Click **Try out the new experience** near the top of the screen to open the new app registration experience, and then click **+New registration > + Add a platform > iOS**.
  - Enter your project's Bundle ID. If you downloaded the code, this is `com.microsoft.identitysample.MSALiOS`. If you're creating your own project, select your project in Xcode and open the **General** tab. The bundle identifier appears in the **Identity** section.
  - Note that for macOS you should be also using iOS experience.
6. Click **Configure** and save the **MSAL Configuration** that appears in the **iOS configuration** page so you can enter it when you configure your app later. Click **Done**.

## Add MSAL

Choose one of the following ways to install the MSAL library in your app:

### CocoaPods

1. If you're using [CocoaPods](#), install `MSAL` by first creating an empty file called `podfile` in the same folder as your project's `.xcodeproj` file. Add the following to `podfile`:

```
use_frameworks!

target '<your-target-here>' do
 pod 'MSAL'
end
```

2. Replace `<your-target-here>` with the name of your project.
3. In a terminal window, navigate to the folder that contains the `podfile` you created and run `pod install` to install the MSAL library.
4. Close Xcode and open `<your project name>.xcworkspace` to reload the project in Xcode.

### Carthage

If you're using [Carthage](#), install `MSAL` by adding it to your `Cartfile`:

```
github "AzureAD/microsoft-authentication-library-for-objc" "master"
```

From a terminal window, in the same directory as the updated `Cartfile`, run the following command to have Carthage update the dependencies in your project.

iOS:

```
carthage update --platform ios
```

macOS:

```
carthage update --platform macOS
```

## Manually

You can also use Git Submodule, or check out the latest release to use as a framework in your application.

## Add your app registration

Next, we'll add your app registration to your code.

First, add the following import statement to the top of the `ViewController.swift` and `AppDelegate.swift` files:

```
import MSAL
```

Then Add the following code to `ViewController.swift` prior to `viewDidLoad()`:

```
let kClientID = "Your_Application_Id_Here"

// Additional variables for Auth and Graph API
let kGraphURI = "https://graph.microsoft.com/v1.0/me/" // the Microsoft Graph endpoint
let kScopes: [String] = ["https://graph.microsoft.com/user.read"] // request permission to read the profile of the signed-in user
let kAuthority = "https://login.microsoftonline.com/common" // this authority allows a personal Microsoft account and a work or school account in any organization's Azure AD tenant to sign in
var accessToken = String()
var applicationContext : MSALPublicClientApplication?
var webViewParameters : MSALWebviewParameters?
```

The only value you need to modify above is the value assigned to `kClientID` to be your [Application ID](#). This value is part of the MSAL Configuration data that you saved during the step at the beginning of this tutorial to register the application in the Azure portal.

## For iOS only, configure URL schemes

In this step, you will register `CFBundleURLSchemes` so that the user can be redirected back to the app after sign in. By the way, `LSApplicationQueriesSchemes` also allows your app to make use of Microsoft Authenticator.

In Xcode, open `Info.plist` as a source code file, and add the following inside of the `<dict>` section. Replace `[BUNDLE_ID]` with the value you used in the Azure portal which, if you downloaded the code, is `com.microsoft.identitysample.MSALiOS`. If you're creating your own project, select your project in Xcode and open the **General** tab. The bundle identifier appears in the **Identity** section.

```
<key>CFBundleURLTypes</key>
<array>
<dict>
<key>CFBundleURLSchemes</key>
<array>
<string>msauth.[BUNDLE_ID]</string>
</array>
</dict>
</array>
<key>LSApplicationQueriesSchemes</key>
<array>
<string>msauthv2</string>
<string>msauthv3</string>
</array>
```

## For macOS only, configure App Sandbox

1. Go to your Xcode Project Settings > **Capabilities tab** > **App Sandbox**
2. Select **Outgoing Connections (Client)** checkbox.

## Create your app's UI

Now create a UI that includes a button to call the Microsoft Graph API, another to sign out, and a text view to see some output by adding the following code to the `ViewController` class:

### iOS UI

```

var loggingText: UITextView!
var signOutButton: UIButton!
var callGraphButton: UIButton!

func initUI() {
 // Add call Graph button
 callGraphButton = UIButton()
 callGraphButton.translatesAutoresizingMaskIntoConstraints = false
 callGraphButton.setTitle("Call Microsoft Graph API", for: .normal)
 callGraphButton.setTitleColor(.blue, for: .normal)
 callGraphButton.addTarget(self, action: #selector(callGraphAPI(_:)), for: .touchUpInside)
 self.view.addSubview(callGraphButton)

 callGraphButton.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
 callGraphButton.topAnchor.constraint(equalTo: view.topAnchor, constant: 50.0).isActive = true
 callGraphButton.widthAnchor.constraint(equalToConstant: 300.0).isActive = true
 callGraphButton.heightAnchor.constraint(equalToConstant: 50.0).isActive = true

 // Add sign out button
 signOutButton = UIButton()
 signOutButton.translatesAutoresizingMaskIntoConstraints = false
 signOutButton.setTitle("Sign Out", for: .normal)
 signOutButton.setTitleColor(.blue, for: .normal)
 signOutButton.setTitleColor(.gray, for: .disabled)
 signOutButton.addTarget(self, action: #selector(signOut(_:)), for: .touchUpInside)
 self.view.addSubview(signOutButton)

 signOutButton.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
 signOutButton.topAnchor.constraint(equalTo: callGraphButton.bottomAnchor, constant: 10.0).isActive =
true
 signOutButton.widthAnchor.constraint(equalToConstant: 150.0).isActive = true
 signOutButton.heightAnchor.constraint(equalToConstant: 50.0).isActive = true
 signOutButton.isEnabled = false

 // Add logging textfield
 loggingText = UITextView()
 loggingText.isUserInteractionEnabled = false
 loggingText.translatesAutoresizingMaskIntoConstraints = false

 self.view.addSubview(loggingText)

 loggingText.topAnchor.constraint(equalTo: signOutButton.bottomAnchor, constant: 10.0).isActive = true
 loggingText.leftAnchor.constraint(equalTo: self.view.leftAnchor, constant: 10.0).isActive = true
 loggingText.rightAnchor.constraint(equalTo: self.view.rightAnchor, constant: 10.0).isActive = true
 loggingText.bottomAnchor.constraint(equalTo: self.view.bottomAnchor, constant: 10.0).isActive = true
}

```

## macOS UI

```

var callGraphButton: NSButton!
var loggingText: NSTextView!
var signOutButton: NSButton!

func initUI() {
 // Add call Graph button
 callGraphButton = NSButton()
 callGraphButton.translatesAutoresizingMaskIntoConstraints = false
 callGraphButton.title = "Call Microsoft Graph API"
 callGraphButton.target = self
 callGraphButton.action = #selector(callGraphAPI(_:))
 callGraphButton.bezelStyle = .rounded
 self.view.addSubview(callGraphButton)

 callGraphButton.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
 callGraphButton.topAnchor.constraint(equalTo: view.topAnchor, constant: 30.0).isActive = true
 callGraphButton.heightAnchor.constraint(equalToConstant: 34.0).isActive = true

 // Add sign out button
 signOutButton = NSButton()
 signOutButton.translatesAutoresizingMaskIntoConstraints = false
 signOutButton.title = "Sign Out"
 signOutButton.target = self
 signOutButton.action = #selector(signOut(_:))
 signOutButton.bezelStyle = .texturedRounded
 self.view.addSubview(signOutButton)

 signOutButton.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
 signOutButton.topAnchor.constraint(equalTo: callGraphButton.bottomAnchor, constant: 10.0).isActive =
true
 signOutButton.heightAnchor.constraint(equalToConstant: 34.0).isActive = true
 signOutButton.isEnabled = false

 // Add logging textfield
 loggingText = NSTextView()
 loggingText.translatesAutoresizingMaskIntoConstraints = false

 self.view.addSubview(loggingText)

 loggingText.topAnchor.constraint(equalTo: signOutButton.bottomAnchor, constant: 10.0).isActive = true
 loggingText.leftAnchor.constraint(equalTo: self.view.leftAnchor, constant: 10.0).isActive = true
 loggingText.rightAnchor.constraint(equalTo: self.view.rightAnchor, constant: -10.0).isActive = true
 loggingText.bottomAnchor.constraint(equalTo: self.view.bottomAnchor, constant: -10.0).isActive = true
 loggingText.widthAnchor.constraint(equalToConstant: 500.0).isActive = true
 loggingText.heightAnchor.constraint(equalToConstant: 300.0).isActive = true
}

```

Next, also inside the `ViewController` class, replace the `viewDidLoad()` method with:

```

override func viewDidLoad() {
 super.viewDidLoad()
 initUI()
 do {
 try self.initMSAL()
 } catch let error {
 self.updateLogging(text: "Unable to create Application Context \(error)")
 }
}

```

## Use MSAL

### Initialize MSAL

Add the following `initMSAL` method to the `ViewController` class:

```
func initMSAL() throws {

 guard let authorityURL = URL(string: kAuthority) else {
 self.updateLogging(text: "Unable to create authority URL")
 return
 }

 let authority = try MSALAADAuthority(url: authorityURL)

 let msalConfiguration = MSALPublicClientApplicationConfig(clientId: kClientID, redirectUri: nil,
authority: authority)
 self.applicationContext = try MSALPublicClientApplication(configuration: msalConfiguration)
 self.initWebViewParams()
}
```

Add the following after `initMSAL` method to the `ViewController` class.

#### iOS code:

```
func initWebViewParams() {
 self.webViewParameters = MSALWebviewParameters(parentViewController: self)
}
```

#### macOS code:

```
func initWebViewParams() {
 self.webViewParameters = MSALWebviewParameters()
 self.webViewParameters?.webviewType = .wkWebView
}
```

#### For iOS only, handle the sign-in callback

Open the `AppDelegate.swift` file. To handle the callback after sign-in, add `MSALPublicClientApplication.handleMSALResponse` to the `AppDelegate` class like this:

```
// Inside AppDelegate...
func application(_ app: UIApplication, open url: URL, options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
 return MSALPublicClientApplication.handleMSALResponse(url, sourceApplication:
options[UIApplication.OpenURLOptionsKey.sourceApplication] as? String)
}
```

If you are using Xcode 11, you should place MSAL callback into the `SceneDelegate.swift` instead. If you support both UISceneDelegate and UIApplicationDelegate for compatibility with older iOS, MSAL callback would need to be placed into both files.

```

func scene(_ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>) {

 guard let urlContext = URLContexts.first else {
 return
 }

 let url = urlContext.url
 let sourceApp = urlContext.options.sourceApplication

 MSALPublicClientApplication.handleMSALResponse(url, sourceApplication: sourceApp)
}

```

### Acquire Tokens

Now, we can implement the application's UI processing logic and get tokens interactively through MSAL.

MSAL exposes two primary methods for getting tokens: `acquireTokenSilently()` and `acquireTokenInteractively()`:

- `acquireTokenSilently()` attempts to sign in a user and get tokens without any user interaction as long as an account is present.
- `acquireTokenInteractively()` always shows UI when attempting to sign in the user. It may use session cookies in the browser or an account in the Microsoft authenticator to provide an interactive-SSO experience.

Add the following code to the `ViewController` class:

```

@objc func callGraphAPI(_ sender: AnyObject) {

 guard let currentAccount = self.currentAccount() else {
 // We check to see if we have a current logged in account.
 // If we don't, then we need to sign someone in.
 acquireTokenInteractively()
 return
 }

 acquireTokenSilently(currentAccount)
}

func currentAccount() -> MSALAccount? {

 guard let applicationContext = self.applicationContext else { return nil }

 // We retrieve our current account by getting the first account from cache
 // In multi-account applications, account should be retrieved by home account identifier or username
 instead

 do {
 let cachedAccounts = try applicationContext.allAccounts()
 if !cachedAccounts.isEmpty {
 return cachedAccounts.first
 }
 } catch let error as NSError {
 self.updateLogging(text: "Didn't find any accounts in cache: \(error)")
 }

 return nil
}

```

### Get a token interactively

The code below gets a token for the first time by creating an `MSALInteractiveTokenParameters` object and calling `acquireToken`. Next you will add code that:

- Creates `MSALInteractiveTokenParameters` with scopes.
- Calls `acquireToken()` with the created parameters.
- Handles errors. For more detail, refer to the [MSAL for iOS and macOS error handling guide](#).
- Handles the successful case.

Add the following code to the `ViewController` class.

```
func acquireTokenInteractively() {

 guard let applicationContext = self.applicationContext else { return }
 guard let webViewParameters = self.webViewParameters else { return }

 // #1
 let parameters = MSALInteractiveTokenParameters(scopes: kScopes, webviewParameters: webViewParameters)

 // #2
 applicationContext.acquireToken(with: parameters) { (result, error) in

 // #3
 if let error = error {

 self.updateLogging(text: "Could not acquire token: \(error)")
 return
 }

 guard let result = result else {

 self.updateLogging(text: "Could not acquire token: No result returned")
 return
 }

 // #4
 self.accessToken = result.accessToken
 self.updateLogging(text: "Access token is \(self.accessToken)")
 self.updateSignOutButton(enabled: true)
 self.getContentWithToken()
 }
}
```

#### **Get a token silently**

To acquire an updated token silently, add the following code to the `ViewController` class. It creates an `MSALSilentTokenParameters` object and calls `acquireTokenSilent()`:

```

func acquireTokenSilently(_ account : MSALAccount!) {
 guard let applicationContext = self.applicationContext else { return }
 let parameters = MSALSilentTokenParameters(scopes: kScopes, account: account)

 applicationContext.acquireTokenSilent(with: parameters) { (result, error) in
 if let error = error {
 let nsError = error as NSError
 if (nsError.domain == MSALErrorDomain) {
 if (nsError.code == MSALError.interactionRequired.rawValue) {
 DispatchQueue.main.async {
 self.acquireTokenInteractively()
 }
 }
 return
 }
 }
 self.updateLogging(text: "Could not acquire token silently: \(error)")
 return
 }

 guard let result = result else {
 self.updateLogging(text: "Could not acquire token: No result returned")
 return
 }

 self.accessToken = result.accessToken
 self.updateLogging(text: "Refreshed Access token is \(self.accessToken)")
 self.updateSignOutButton(enabled: true)
 self.getContentWithToken()
}
}

```

## Call the Microsoft Graph API

Once you have a token, your app can use it in the HTTP header to make an authorized request to the Microsoft Graph:

HEADER KEY	VALUE
Authorization	Bearer <access-token>

Add the following code to the `ViewController` class:

```

func getContentWithToken() {
 // Specify the Graph API endpoint
 let url = URL(string: kGraphURI)
 var request = URLRequest(url: url!)

 // Set the Authorization header for the request. We use Bearer tokens, so we specify Bearer + the token
 // we got from the result
 request.setValue("Bearer \((self.accessToken)", forHTTPHeaderField: "Authorization")

 URLSession.shared.dataTask(with: request) { data, response, error in

 if let error = error {
 self.updateLogging(text: "Couldn't get graph result: \(error)")
 return
 }

 guard let result = try? JSONSerialization.jsonObject(with: data!, options: []) else {

 self.updateLogging(text: "Couldn't deserialize result JSON")
 return
 }

 self.updateLogging(text: "Result from Graph: \(result)")

 }.resume()
 }
}

```

See [Microsoft Graph API](#) to learn more about the Microsoft Graph API.

## Use MSAL for Sign-out

Next, add support for sign-out.

### IMPORTANT

Sigining out with MSAL removes all known information about a user from the application, but the user will still have an active session on their device. If the user attempts to sign in again they may see sign-in UI, but may not need to reenter their credentials because the device session is still active.

To add sign-out capability, add the following code inside the `ViewController` class. This method cycles through all accounts and removes them:

```

@objc func signOut(_ sender: AnyObject) {

 guard let applicationContext = self.applicationContext else { return }

 guard let account = self.currentAccount() else { return }

 do {

 /**
 Removes all tokens from the cache for this application for the provided account

 - account: The account to remove from the cache
 */

 try applicationContext.remove(account)
 self.updateLogging(text: "")
 self.updateSignOutButton(enabled: false)
 self.accessToken = ""

 } catch let error as NSError {

 self.updateLogging(text: "Received error signing account out: \(error)")
 }
}

```

## Enable token caching

By default, MSAL caches your app's tokens in the iOS or macOS keychain.

To enable token caching:

1. Ensure your application is properly signed
2. Go to your Xcode Project Settings > **Capabilities tab** > **Enable Keychain Sharing**
3. Click + and enter a following **Keychain Groups** entry: 3.a For iOS, enter `com.microsoft.adalcache` 3.b For macOS enter `com.microsoft.identity.universalstorage`

## Add helper methods

Add the following helper methods to the `ViewController` class to complete the sample.

**iOS UI:**

```

func updateLogging(text : String) {

 if Thread.isMainThread {
 self.loggingText.text = text
 } else {
 DispatchQueue.main.async {
 self.loggingText.text = text
 }
 }
}

func updateSignOutButton(enabled : Bool) {
 if Thread.isMainThread {
 self.signOutButton.isEnabled = enabled
 } else {
 DispatchQueue.main.async {
 self.signOutButton.isEnabled = enabled
 }
 }
}

```

## macOS UI:

```
func updateSignOutButton(enabled : Bool) {
 if Thread.isMainThread {
 self.signOutButton.isEnabled = enabled
 } else {
 DispatchQueue.main.async {
 self.signOutButton.isEnabled = enabled
 }
 }
}

func updateLogging(text : String) {

 if Thread.isMainThread {
 self.loggingText.string = text
 } else {
 DispatchQueue.main.async {
 self.loggingText.string = text
 }
 }
}
```

## Multi-account applications

This app is built for a single account scenario. MSAL also supports multi-account scenarios, but it requires some additional work from apps. You will need to create UI to help users select which account they want to use for each action that requires tokens. Alternatively, your app can implement a heuristic to select which account to use via the `getAccounts()` method.

## Test your app

### Run locally

Build and deploy the app to a test device or simulator. You should be able to sign in and get tokens for Azure AD or personal Microsoft accounts.

The first time a user signs into your app, they will be prompted by Microsoft identity to consent to the permissions requested. While most users are capable of consenting, some Azure AD tenants have disabled user consent, which requires admins to consent on behalf of all users. To support this scenario, register your app's scopes in the Azure portal.

After you sign in, the app will display the data returned from the Microsoft Graph `/me` endpoint.

## Get help

Visit [Help and support](#) if you have trouble with this tutorial or with the Microsoft identity platform.

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

# Call Microsoft Graph API from a Universal Windows Platform application (XAML)

9/25/2019 • 12 minutes to read • [Edit Online](#)

This guide explains how a native Universal Windows Platform (UWP) application can request an access token. The application then calls Microsoft Graph API. The guide also applies to other APIs that require access tokens from the Microsoft identity platform endpoint.

At the end of this guide, your application calls a protected API by using personal accounts. Examples are outlook.com, live.com, and others. Your application also calls work and school accounts from any company or organization that has Azure Active Directory (Azure AD).

## NOTE

This guide requires Visual Studio with Universal Windows Platform development installed. See [Get set up](#) for instructions to download and configure Visual Studio to develop Universal Windows Platform apps.

## How this guide works

This guide creates a sample UWP application that queries Microsoft Graph API. For this scenario, a token is added to HTTP requests by using the Authorization header. Microsoft Authentication Library (MSAL) handles token acquisitions and renewals.

## NuGet packages

This guide uses the following NuGet package:

LIBRARY	DESCRIPTION
<a href="#">Microsoft.Identity.Client</a>	Microsoft Authentication Library

## Set up your project

This section provides step-by-step instructions to integrate a Windows Desktop .NET application (XAML) with Sign-In with Microsoft. Then the application can query Web APIs that require a token, such as Microsoft Graph API.

This guide creates an application that displays a button that queries Graph API and a button to sign out. It also displays text boxes that contain the results of the calls.

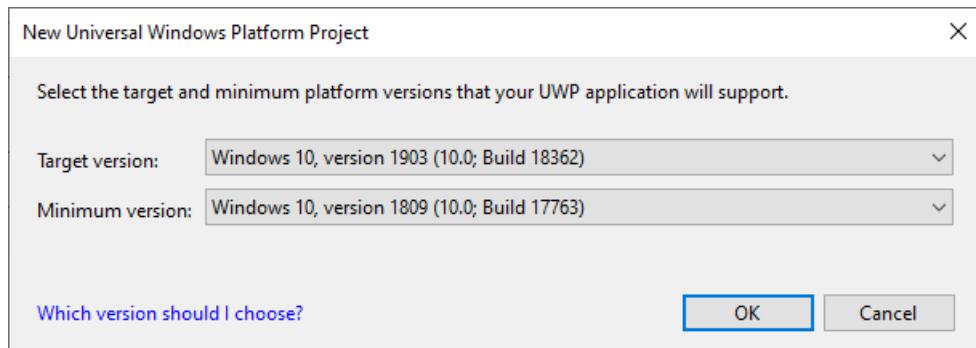
## NOTE

Do you want to download this sample's Visual Studio project instead of creating it? [Download a project](#) and skip to the [application registration](#) step to configure the code sample before it runs.

## Create your application

1. Open Visual Studio and select **Create a new project**.

2. In **Create a new project**, choose **Blank App (Universal Windows)** for C# and select **Next**.
3. In **Configure your new project**, name the app, and select **Create**.
4. If prompted, in **New Universal Windows Platform Project**, select any version for **Target** and **Minimum** versions, and select **OK**.



## Add Microsoft Authentication Library to your project

1. In Visual Studio, select **Tools > NuGet Package Manager > Package Manager Console**.
2. Copy and paste the following command in the **Package Manager Console** window:

```
Install-Package Microsoft.Identity.Client
```

### NOTE

This command installs [Microsoft Authentication Library](#). MSAL acquires, caches, and refreshes user tokens that access APIs protected by Microsoft identity platform.

## Create your application's UI

Visual Studio creates *MainPage.xaml* as a part of your project template. Open this file, and then replace your application's **Grid** node with the following code:

```
<Grid>
 <StackPanel Background="Azure">
 <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
 <Button x:Name="CallGraphButton" Content="Call Microsoft Graph API" HorizontalAlignment="Right" Padding="5" Click="CallGraphButton_Click" Margin="5" FontFamily="Segoe Ui"/>
 <Button x:Name="SignOutButton" Content="Sign-Out" HorizontalAlignment="Right" Padding="5" Click="SignOutButton_Click" Margin="5" Visibility="Collapsed" FontFamily="Segoe Ui"/>
 </StackPanel>
 <TextBlock Text="API Call Results" Margin="2,0,0,-5" FontFamily="Segoe Ui" />
 <TextBox x:Name="ResultText" TextWrapping="Wrap" MinHeight="120" Margin="5" FontFamily="Segoe Ui" />
 <TextBlock Text="Token Info" Margin="2,0,0,-5" FontFamily="Segoe Ui" />
 <TextBox x:Name="TokenInfoText" TextWrapping="Wrap" MinHeight="70" Margin="5" FontFamily="Segoe Ui" />
 </StackPanel>
</Grid>
```

## Use MSAL to get a token for Microsoft Graph API

This section shows how to use MSAL to get a token for Microsoft Graph API. Make changes to the *MainPage.xaml.cs* file.

1. In *MainPage.xaml.cs*, add the following references:

```
using Microsoft.Identity.Client;
using System.Diagnostics;
using System.Threading.Tasks;
```

2. Replace your `MainPage` class with the following code:

```
public sealed partial class MainPage : Page
{
 //Set the API Endpoint to Graph 'me' endpoint
 string graphAPIEndpoint = "https://graph.microsoft.com/v1.0/me";

 //Set the scope for API call to user.read
 string[] scopes = new string[] { "user.read" };

 // Below are the clientId (Application Id) of your app registration and the tenant information.
 // You have to replace:
 // - the content of ClientID with the Application Id for your app registration
 // - the content of Tenant with the information about the accounts allowed to sign in in your
 // application:
 // - for Work or School account in your org, use your tenant ID, or domain
 // - for any Work or School accounts, use organizations
 // - for any Work or School accounts, or Microsoft personal account, use common
 // - for Microsoft Personal account, use consumers
 private const string ClientId = "0b8b0665-bc13-4fdc-bd72-e0227b9fc011";

 public IPublicClientApplication PublicClientApp { get; }

 public MainPage()
 {
 InitializeComponent();

 PublicClientApp = PublicClientApplicationBuilder.Create(ClientId)
 .WithAuthority(AadAuthorityAudience.AzureAdAndPersonalMicrosoftAccount)
 .WithLogging((level, message, containsPii) =>
 {
 Debug.WriteLine($"MSAL: {level} {message} ");
 }, LogLevel.Warning, enablePiiLogging:false,enableDefaultPlatformLogging:true)
 .WithUseCorporateNetwork(true)
 .Build();
 }

 /// <summary>
 /// Call AcquireTokenInteractive - to acquire a token requiring user to sign-in
 /// </summary>
 private async void CallGraphButton_Click(object sender, RoutedEventArgs e)
 {
 AuthenticationResult authResult = null;
 ResultText.Text = string.Empty;
 TokenInfoText.Text = string.Empty;

 // It's good practice to not do work on the UI thread, so use ConfigureAwait(false) whenever
 // possible.
 IEnumerable<IAccount> accounts = await PublicClientApp.GetAccountsAsync().ConfigureAwait(false);
 IAccount firstAccount = accounts.FirstOrDefault();

 try
 {
 authResult = await PublicClientApp.AcquireTokenSilent(scopes, firstAccount)
 .ExecuteAsync();
 }
 catch (MsalUiRequiredException ex)
 {
 // A MsalUiRequiredException happened on AcquireTokenSilent.
 // This indicates you need to call AcquireTokenInteractive to acquire a token
 System.Diagnostics.Debug.WriteLine($"MsalUiRequiredException: {ex.Message}");
 }
 }
}
```

```

try
{
 authResult = await PublicClientApp.AcquireTokenInteractive(scopes)
 .ExecuteAsync()
 .ConfigureAwait(false);
}
catch (MsalException msalex)
{
 await DisplayMessageAsync($"Error Acquiring Token:{System.Environment.NewLine}{msalex}");
}
catch (Exception ex)
{
 await DisplayMessageAsync($"Error Acquiring Token Silently:{System.Environment.NewLine}{ex}");
 return;
}

if (authResult != null)
{
 var content = await GetHttpContentWithToken(graphAPIEndpoint,
 authResult.AccessToken).ConfigureAwait(false);

 // Go back to the UI thread to make changes to the UI
 await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
 {
 ResultText.Text = content;
 DisplayBasicTokenInfo(authResult);
 this.SignOutButton.Visibility = Visibility.Visible;
 });
}
}
}

```

### Get a user token interactively

The `AcquireTokenInteractive` method results in a window that prompts users to sign in. Applications usually require users to sign in interactively the first time to access a protected resource. They might also need to sign in when a silent operation to acquire a token fails. An example is when a user's password has expired.

### Get a user token silently

The `AcquireTokenSilent` method handles token acquisitions and renewals without any user interaction. After `AcquireTokenInteractive` runs for the first time and prompts the user for credentials, use the `AcquireTokenSilent` method to request tokens for later calls. That method acquires tokens silently. MSAL handles token cache and renewal.

Eventually, the `AcquireTokenSilent` method fails. Reasons for failure include a user that signed out or changed their password on another device. When MSAL detects that the issue requires an interactive action, it throws an `MsalUiRequiredException` exception. Your application can handle this exception in two ways:

- Your application calls `AcquireTokenInteractive` immediately. This call results in prompting the user to sign in. Normally, use this approach for online applications where there's no available offline content for the user. The sample generated by this guided setup follows the pattern. You see it in action the first time you run the sample.

Because no user has used the application, `accounts.FirstOrDefault()` contains a null value, and throws an `MsalUiRequiredException` exception.

The code in the sample then handles the exception by calling `AcquireTokenInteractive`. This call results in prompting the user to sign in.

- Your application presents a visual indication to users that they need to sign in. Then they can select the right time to sign in. The application can retry `AcquireTokenSilent` later. Use this approach when users can use other application functionality without disruption. An example is when offline content is available in the

application. In this case, users can decide when they want to sign in. The application can retry `AcquireTokenSilent` after the network was temporarily unavailable.

## Call Microsoft Graph API by using the token you just obtained

Add the following new method to `MainPage.xaml.cs`:

```
/// <summary>
/// Perform an HTTP GET request to a URL using an HTTP Authorization header
/// </summary>
/// <param name="url">The URL</param>
/// <param name="token">The token</param>
/// <returns>String containing the results of the GET operation</returns>
public async Task<string> GetHttpContentWithToken(string url, string token)
{
 var httpClient = new System.Net.Http.HttpClient();
 System.Net.Http.HttpResponseMessage response;
 try
 {
 var request = new System.Net.Http.HttpRequestMessage(System.Net.Http.HttpMethod.Get, url);
 // Add the token in Authorization header
 request.Headers.Authorization =
 new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
 response = await httpClient.SendAsync(request);
 var content = await response.Content.ReadAsStringAsync();
 return content;
 }
 catch (Exception ex)
 {
 return ex.ToString();
 }
}
```

This method makes a `GET` request from Graph API by using an `Authorization` header.

### More information on making a REST call against a protected API

In this sample application, the `GetHttpContentWithToken` method make an HTTP `GET` request against a protected resource that requires a token. Then the method returns the content to the caller. This method adds the acquired token in the **HTTP Authorization** header. For this sample, the resource is the Microsoft Graph API `me` endpoint, which displays the user's profile information.

## Add a method to sign out the user

To sign out the user, add the following method to `MainPage.xaml.cs`:

```

/// <summary>
/// Sign out the current user
/// </summary>
private async void SignOutButton_Click(object sender, RoutedEventArgs e)
{
 IEnumerable<IAccount> accounts = await PublicClientApp.GetAccountsAsync().ConfigureAwait(false);
 IAccount firstAccount = accounts.FirstOrDefault();

 try
 {
 await PublicClientApp.RemoveAsync(firstAccount).ConfigureAwait(false);
 await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
 {
 ResultText.Text = "User has signed out";
 this.CallGraphButton.Visibility = Visibility.Visible;
 this.SignOutButton.Visibility = Visibility.Collapsed;
 });
 }
 catch (MsalException ex)
 {
 ResultText.Text = $"Error signing out user: {ex.Message}";
 }
}

```

#### NOTE

MSAL.NET uses asynchronous methods to acquire tokens or manipulate accounts. You need to support UI actions in the UI thread. This is the reason for the `Dispatcher.RunAsync` call and the precautions to call `ConfigureAwait(false)`.

#### More information about signing out

The `SignOutButton_Click` method removes the user from the MSAL user cache. This method effectively tells MSAL to forget the current user. A future request to acquire a token succeeds only if it's interactive.

The application in this sample supports a single user. MSAL supports scenarios where the user can sign in on more than one account. An example is an email application where a user has several accounts.

#### Display basic token information

Add the following method to `MainPage.xaml.cs` to display basic information about the token:

```

/// <summary>
/// Display basic information contained in the token. Needs to be called from the UI thread.
/// </summary>
private void DisplayBasicTokenInfo(AuthenticationResult authResult)
{
 TokenInfoText.Text = "";
 if (authResult != null)
 {
 TokenInfoText.Text += $"User Name: {authResult.Account.Username}" + Environment.NewLine;
 TokenInfoText.Text += $"Token Expires: {authResult.ExpiresOn.ToLocalTime()}" + Environment.NewLine;
 }
}

```

#### More information

ID tokens acquired by using **OpenID Connect** also contain a small subset of information pertinent to the user.

`DisplayBasicTokenInfo` displays basic information contained in the token. This information includes the user's display name and ID. It also includes the expiration date of the token and the string that represents the access token itself. If you select the **Call Microsoft Graph API** button several times, you'll see that the same token was reused for later requests. You can also see the expiration date extended when MSAL decides it's time to renew the token.

## Display message

Add the following new method to *MainPage.xaml.cs*:

```
/// <summary>
/// Displays a message in the ResultText. Can be called from any thread.
/// </summary>
private async Task DisplayMessageAsync(string message)
{
 await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal,
 () =>
 {
 ResultText.Text = message;
 });
}
```

## Register your application

Now you need to register your application:

1. Sign in to the [Azure portal](#).
2. Select **Azure Active Directory > App registrations**.
3. Select **New registration**. Enter a meaningful application name that will be displayed to users of the app, for example *UWP-App-calling-MSGraph*.
4. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox)**, then select **Register** to continue.
5. On the overview page, find the **Application (client) ID** value and copy it. Go back to Visual Studio, open *MainPage.xaml.cs*, and replace the value of `ClientId` with this value.

Configure authentication for your application:

1. Back in the [Azure portal](#), under **Manage**, select **Authentication**.
2. In the **Redirect URIs** list, for **TYPE**, select **Public client (mobile & desktop)** and enter `urn:ietf:wg:oauth:2.0:oob` for **REDIRECT URI**.
3. Select **Save**.

Configure API permissions for your application:

1. Under **Manage**, select **API permissions**.
2. Select **Add a permission** and then make sure that you've selected **Microsoft APIs**.
3. Select **Microsoft Graph**.
4. Select **Delegated permissions**, search for *User.Read* and verify that **User.Read** is selected.
5. If you made any changes, select **Add permissions** to save them.

## Enable integrated authentication on federated domains (optional)

To enable Windows-Integrated Authentication when it's used with a federated Azure AD domain, the application manifest must enable additional capabilities. Go back to your application in Visual Studio.

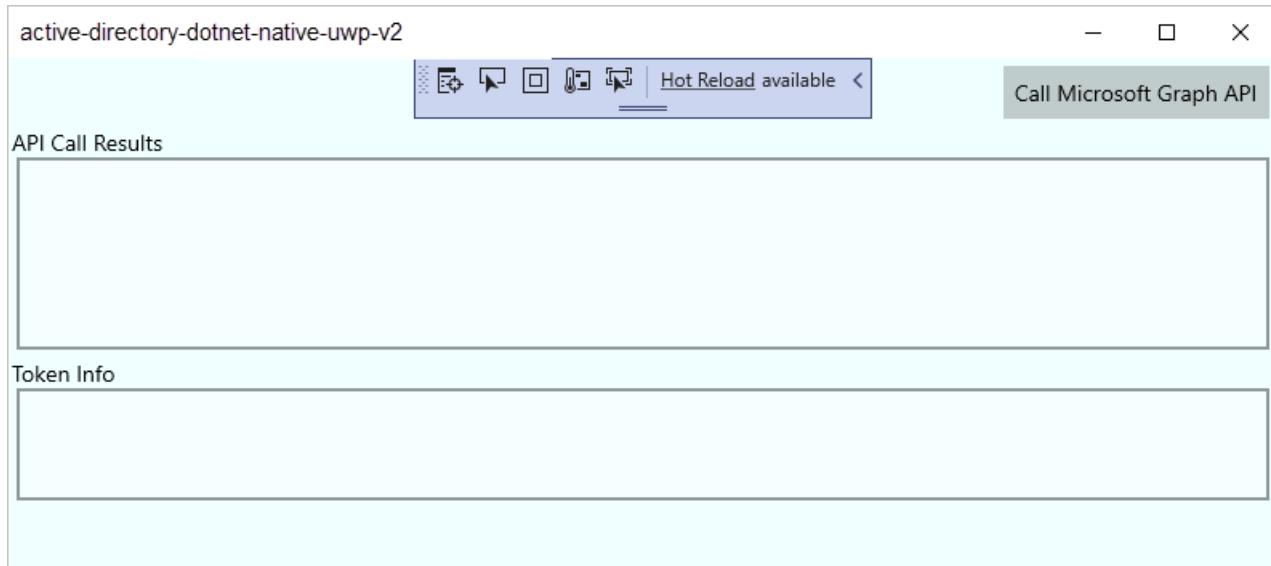
1. Open *Package.appxmanifest*.
2. Select **Capabilities** and enable the following settings:
  - **Enterprise Authentication**
  - **Private Networks (Client & Server)**
  - **Shared User Certificates**

## IMPORTANT

Integrated Windows Authentication is not configured by default for this sample. Applications that request Enterprise Authentication or Shared User Certificates capabilities require a higher level of verification by the Windows Store. Also, not all developers want to perform the higher level of verification. Enable this setting only if you need Windows Integrated Authentication with a federated Azure AD domain.

## Test your code

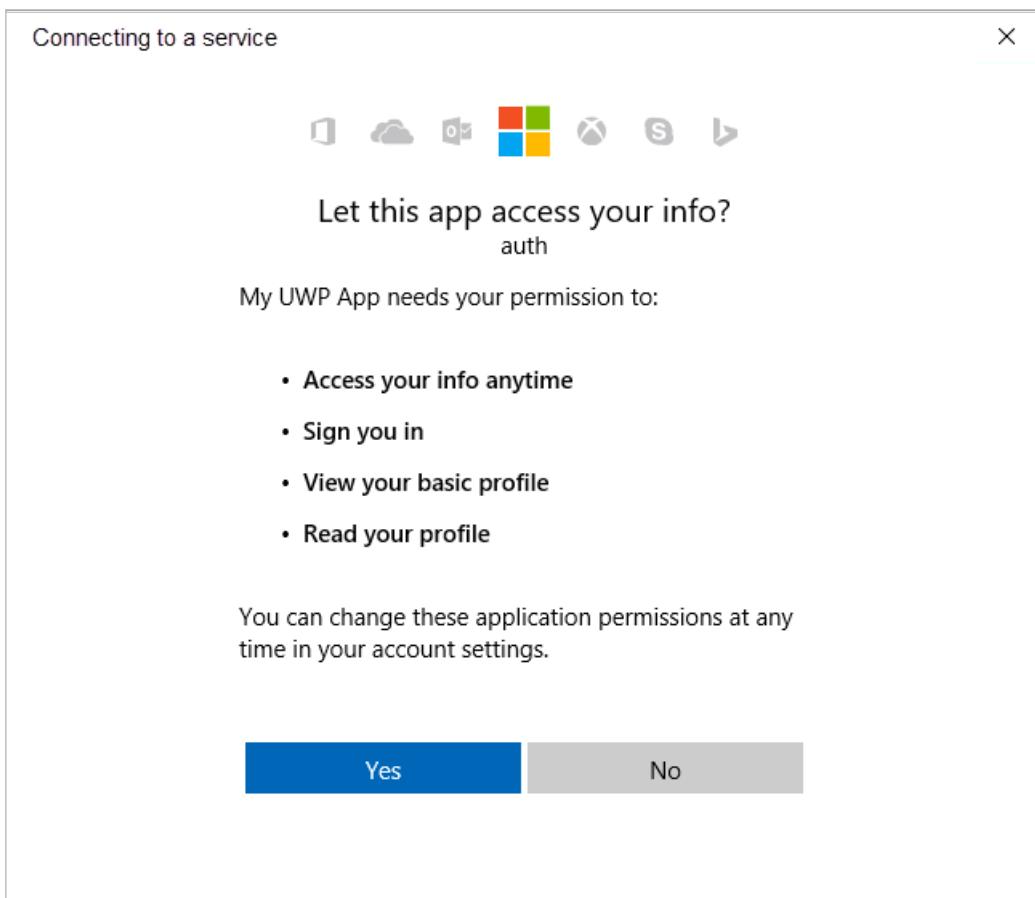
To test your application, select F5 to run your project in Visual Studio. Your main window appears:



When you're ready to test, select **Call Microsoft Graph API**. Then use an Azure AD organizational account or a Microsoft account, such as live.com or outlook.com, to sign in. The first time a user runs this, the application displays a window asking the user to sign in.

### Consent

The first time you sign in to your application, you're presented with a consent screen similar to the following. Select **Yes** to explicitly consent to access:



## Expected results

You see user profile information returned by the Microsoft Graph API call on the **API Call Results** screen:

The screenshot shows the 'active-directory-dotnet-native-uwp-v2' application window. The 'API Call Results' section displays the following JSON response:

```
{"@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users/$entity", "businessPhones": [], "displayName": "Anne Shepard (Contoso)", "givenName": "Anne", "jobTitle": null, "mail": "ashepard@contoso.com", "mobilePhone": null, "officeLocation": "Contoso Main", "preferredLanguage": null, "surname": "Shepard", "userPrincipalName": "ashepard@contoso.com", "id": "753d4b04-d21c-41a6-906e-8051e2ee629e"}
```

The 'Token Info' section shows the following information:

User Name: ashepard@contoso.com
Token Expires: 9/24/2019 10:04:24 AM -07:00

You also see basic information about the token acquired via `AcquireTokenInteractive` or `AcquireTokenSilent` in the **Token Info** box:

PROPERTY	FORMAT	DESCRIPTION
Username	user@domain.com	The username that identifies the user.
Token Expires	DateTime	The time when the token expires. MSAL extends the expiration date by renewing the token as necessary.

[More information about scopes and delegated permissions](#)

Microsoft Graph API requires the `user.read` scope to read a user's profile. This scope is added by default in every application that's registered in the Application Registration Portal. Other APIs for Microsoft Graph and custom APIs for your back-end server might require additional scopes. For instance, Microsoft Graph API requires the `Calendars.Read` scope to list the user's calendars.

To access the user's calendars in the context of an application, add the `Calendars.Read` delegated permission to the application registration information. Then add the `Calendars.Read` scope to the `acquireTokenSilent` call.

#### NOTE

Users might be prompted for additional consents as you increase the number of scopes.

## Known issues

### Issue 1

You receive one of the following error messages when you sign in on your application on a federated Azure AD domain:

- No valid client certificate found in the request.
- No valid certificates found in the user's certificate store.
- Try again choosing a different authentication method.

Cause: Enterprise and certificate capabilities aren't enabled.

Solution: Follow the steps in [Enable integrated authentication on federated domains \(optional\)](#).

### Issue 2

You enable [integrated authentication on federated domains](#) and try to use Windows Hello on a Windows 10 computer to sign in to an environment with multi-factor authentication configured. The list of certificates is presented. However, if you choose to use your PIN, the PIN window is never presented.

Cause: This issue is a known limitation of the web authentication broker in UWP applications that run on Windows 10 desktop. It works fine on Windows 10 Mobile.

Workaround: Select **Sign in with other options**. Then select **Sign in with a username and password**. Select **Provide your password**. Then go through the phone authentication process.

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

### [Help and support for developers](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey:

### [Microsoft identity platform survey](#)

# Call the Microsoft Graph API from a Windows Desktop app

8/15/2019 • 12 minutes to read • [Edit Online](#)

This guide demonstrates how a native Windows Desktop .NET (XAML) application uses an access token to call the Microsoft Graph API. The app can also access other APIs that require access tokens from a Microsoft identity platform for developers v2.0 endpoint. This platform was formerly named Azure AD.

When you've completed the guide, your application will be able to call a protected API that uses personal accounts (including outlook.com, live.com, and others). The application will also use work and school accounts from any company or organization that uses Azure Active Directory.

## NOTE

The guide requires Visual Studio 2015 Update 3, Visual Studio 2017, or Visual Studio 2019. Don't have any of these versions? [Download Visual Studio 2019 for free.](#)

## How the sample app generated by this guide works

The sample application that you create with this guide enables a Windows Desktop application that queries the Microsoft Graph API or a Web API that accepts tokens from a Microsoft identity-platform endpoint. For this scenario, you add a token to HTTP requests via the Authorization header. Microsoft Authentication Library (MSAL) handles token acquisition and renewal.

## Handling token acquisition for accessing protected Web APIs

After the user is authenticated, the sample application receives a token you can use to query Microsoft Graph API or a Web API that's secured by Microsoft identity platform for developers.

APIs such as Microsoft Graph require a token to allow access to specific resources. For example, a token is required to read a user's profile, access a user's calendar, or send email. Your application can request an access token by using MSAL to access these resources by specifying API scopes. This access token is then added to the HTTP Authorization header for every call that's made against the protected resource.

MSAL manages caching and refreshing access tokens for you, so that your application doesn't need to.

## NuGet packages

This guide uses the following NuGet packages:

LIBRARY	DESCRIPTION
<a href="#">Microsoft.Identity.Client</a>	Microsoft Authentication Library (MSAL.NET)

## Set up your project

In this section you create a new project to demonstrate how to integrate a Windows Desktop .NET application

(XAML) with *Sign-In with Microsoft* so that the application can query Web APIs that require a token.

The application that you create with this guide displays a button that's used to call a graph, an area to show the results on the screen, and a sign-out button.

#### NOTE

Prefer to download this sample's Visual Studio project instead? [Download a project](#), and skip to the [Configuration step](#) to configure the code sample before you execute it.

To create your application, do the following:

1. In Visual Studio, select **File > New > Project**.
2. Under **Templates**, select **Visual C#**.
3. Select **WPF App (.NET Framework)**, depending on the version of Visual Studio version you're using.

## Add MSAL to your project

1. In Visual Studio, select **Tools > NuGet Package Manager> Package Manager Console**.
2. In the Package Manager Console window, paste the following Azure PowerShell command:

```
Install-Package Microsoft.Identity.Client -Pre
```

#### NOTE

This command installs Microsoft Authentication Library. MSAL handles acquiring, caching, and refreshing user tokens that are used to access the APIs that are protected by Azure Active Directory v2.0

## Register your application

You can register your application in either of two ways.

### Option 1: Express mode

You can quickly register your application by doing the following:

1. Go to the [Azure portal - Application Registration](#).
2. Enter a name for your application and select **Register**.
3. Follow the instructions to download and automatically configure your new application with just one click.

### Option 2: Advanced mode

To register your application and add your application registration information to your solution, do the following:

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. Navigate to the Microsoft identity platform for developers [App registrations](#) page.
4. Select **New registration**.
  - In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `Win-App-calling-MsGraph`.
  - In the **Supported account types** section, select **Accounts in any organizational directory and**

**personal Microsoft accounts (for example, Skype, Xbox, Outlook.com).**

- Select **Register** to create the application.
5. In the list of pages for the app, select **Authentication**.
- a. In the **Redirect URIs** section, in the Redirect URIs list:
  - b. In the **TYPE** column select **Public client (mobile & desktop)**.
  - c. In the **REDIRECT URI** column, enter `urn:ietf:wg:oauth:2.0:oob`
6. Select **Save**.
7. Go to Visual Studio, open the *App.xaml.cs* file, and then replace `Enter_the_Application_Id_here` in the code snippet below with the application ID that you just registered and copied.

```
private static string ClientId = "Enter_the_Application_Id_here";
```

## Add the code to initialize MSAL

In this step, you create a class to handle interaction with MSAL, such as handling of tokens.

1. Open the *App.xaml.cs* file, and then add the reference for MSAL to the class:

```
using Microsoft.Identity.Client;
```

2. Update the app class to the following:

```
public partial class App : Application
{
 static App()
 {
 _clientApp = PublicClientApplicationBuilder.Create(ClientId)
 .WithAuthority(AzureCloudInstance.AzurePublic, Tenant)
 .Build();
 }

 // Below are the clientId (Application Id) of your app registration and the tenant information.
 // You have to replace:
 // - the content of ClientID with the Application Id for your app registration
 // - the content of Tenant by the information about the accounts allowed to sign-in in your
 // application:
 // - For Work or School account in your org, use your tenant ID, or domain
 // - for any Work or School accounts, use `organizations`
 // - for any Work or School accounts, or Microsoft personal account, use `common`
 // - for Microsoft Personal account, use consumers
 private static string ClientId = "0b8b0665-bc13-4fdc-bd72-e0227b9fc011";

 private static string Tenant = "common";

 private static IPublicClientApplication _clientApp;

 public static IPublicClientApplication PublicClientApp { get { return _clientApp; } }
}
```

## Create the application UI

This section shows how an application can query a protected back-end server such as Microsoft Graph.

A *MainWindow.xaml* file should automatically be created as a part of your project template. Open this file, and then replace your application's `<Grid>` node with the following code:

```
<Grid>
 <StackPanel Background="Azure">
 <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
 <Button x:Name="CallGraphButton" Content="Call Microsoft Graph API" HorizontalAlignment="Right"
 Padding="5" Click="CallGraphButton_Click" Margin="5" FontFamily="Segoe Ui"/>
 <Button x:Name="SignOutButton" Content="Sign-Out" HorizontalAlignment="Right" Padding="5"
 Click="SignOutButton_Click" Margin="5" Visibility="Collapsed" FontFamily="Segoe Ui"/>
 </StackPanel>
 <Label Content="API Call Results" Margin="0,0,0,-5" FontFamily="Segoe Ui" />
 <TextBox x:Name="ResultText" TextWrapping="Wrap" MinHeight="120" Margin="5" FontFamily="Segoe Ui"/>
 <Label Content="Token Info" Margin="0,0,0,-5" FontFamily="Segoe Ui" />
 <TextBox x:Name="TokenInfoText" TextWrapping="Wrap" MinHeight="70" Margin="5" FontFamily="Segoe Ui"/>
 </StackPanel>
</Grid>
```

## Use MSAL to get a token for the Microsoft Graph API

In this section, you use MSAL to get a token for the Microsoft Graph API.

1. In the `MainWindow.xaml.cs` file, add the reference for MSAL to the class:

```
using Microsoft.Identity.Client;
```

2. Replace the `MainWindow` class code with the following:

```

public partial class MainWindow : Window
{
 //Set the API Endpoint to Graph 'me' endpoint
 string graphAPIEndpoint = "https://graph.microsoft.com/v1.0/me";

 //Set the scope for API call to user.read
 string[] scopes = new string[] { "user.read" };

 public MainWindow()
 {
 InitializeComponent();
 }

 /// <summary>
 /// Call AcquireToken - to acquire a token requiring user to sign-in
 /// </summary>
 private async void CallGraphButton_Click(object sender, RoutedEventArgs e)
 {
 AuthenticationResult authResult = null;
 var app = App.PublicClientApp;
 ResultText.Text = string.Empty;
 TokenInfoText.Text = string.Empty;

 var accounts = await app.GetAccountsAsync();
 var firstAccount = accounts.FirstOrDefault();

 try
 {
 authResult = await app.AcquireTokenSilent(scopes, firstAccount)
 .ExecuteAsync();
 }
 catch (MsalUiRequiredException ex)
 {
 // A MsalUiRequiredException happened on AcquireTokenSilent.
 // This indicates you need to call AcquireTokenInteractive to acquire a token
 System.Diagnostics.Debug.WriteLine($"MsalUiRequiredException: {ex.Message}");

 try
 {
 authResult = await app.AcquireTokenInteractive(scopes)
 .WithAccount(accounts.FirstOrDefault())
 .WithPrompt(Prompt.SelectAccount)
 .ExecuteAsync();
 }
 catch (MsalException msalex)
 {
 ResultText.Text = $"Error Acquiring Token:{System.Environment.NewLine}{msalex}";
 }
 }
 catch (Exception ex)
 {
 ResultText.Text = $"Error Acquiring Token Silently:{System.Environment.NewLine}{ex}";
 return;
 }

 if (authResult != null)
 {
 ResultText.Text = await GetHttpContentWithToken(graphAPIEndpoint, authResult.AccessToken);
 DisplayBasicTokenInfo(authResult);
 this.SignOutButton.Visibility = Visibility.Visible;
 }
 }
}

```

## More information

[Get a user token interactively](#)

Calling the `AcquireTokenInteractive` method results in a window that prompts users to sign in. Applications usually require users to sign in interactively the first time they need to access a protected resource. They might also need to sign in when a silent operation to acquire a token fails (for example, when a user's password is expired).

#### Get a user token silently

The `AcquireTokenSilent` method handles token acquisitions and renewals without any user interaction. After `AcquireTokenInteractive` is executed for the first time, `AcquireTokenSilent` is the usual method to use to obtain tokens that access protected resources for subsequent calls, because calls to request or renew tokens are made silently.

Eventually, the `AcquireTokenSilent` method will fail. Reasons for failure might be that the user has either signed out or changed their password on another device. When MSAL detects that the issue can be resolved by requiring an interactive action, it fires an `MsalUiRequiredException` exception. Your application can handle this exception in two ways:

- It can make a call against `AcquireTokenInteractive` immediately. This call results in prompting the user to sign in. This pattern is usually used in online applications where there is no available offline content for the user. The sample generated by this guided setup follows this pattern, which you can see in action the first time you execute the sample.
- Because no user has used the application, `PublicClientApp.Users.FirstOrDefault()` contains a null value, and an `MsalUiRequiredException` exception is thrown.
- The code in the sample then handles the exception by calling `AcquireTokenInteractive`, which results in prompting the user to sign in.
- It can instead present a visual indication to users that an interactive sign-in is required, so that they can select the right time to sign in. Or the application can retry `AcquireTokenSilent` later. This pattern is frequently used when users can use other application functionality without disruption—for example, when offline content is available in the application. In this case, users can decide when they want to sign in to either access the protected resource or refresh the outdated information. Alternatively, the application can decide to retry `AcquireTokenSilent` when the network is restored after having been temporarily unavailable.

## Call the Microsoft Graph API by using the token you just obtained

Add the following new method to your `MainWindow.xaml.cs`. The method is used to make a `GET` request against Graph API by using an Authorize header:

```

/// <summary>
/// Perform an HTTP GET request to a URL using an HTTP Authorization header
/// </summary>
/// <param name="url">The URL</param>
/// <param name="token">The token</param>
/// <returns>String containing the results of the GET operation</returns>
public async Task<string> GetHttpContentWithToken(string url, string token)
{
 var httpClient = new System.Net.Http.HttpClient();
 System.Net.Http.HttpResponseMessage response;
 try
 {
 var request = new System.Net.Http.HttpRequestMessage(System.Net.Http.HttpMethod.Get, url);
 //Add the token in Authorization header
 request.Headers.Authorization = new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
 response = await httpClient.SendAsync(request);
 var content = await response.Content.ReadAsStringAsync();
 return content;
 }
 catch (Exception ex)
 {
 return ex.ToString();
 }
}

```

## More information about making a REST call against a protected API

In this sample application, you use the `GetHttpContentWithToken` method to make an HTTP `GET` request against a protected resource that requires a token and then return the content to the caller. This method adds the acquired token in the HTTP Authorization header. For this sample, the resource is the Microsoft Graph API `me` endpoint, which displays the user's profile information.

## Add a method to sign out a user

To sign out a user, add the following method to your `MainWindow.xaml.cs` file:

```

/// <summary>
/// Sign out the current user
/// </summary>
private async void SignOutButton_Click(object sender, RoutedEventArgs e)
{
 var accounts = await App.PublicClientApp.GetAccountsAsync();

 if (accounts.Any())
 {
 try
 {
 await App.PublicClientApp.RemoveAsync(accounts.FirstOrDefault());
 this.ResultText.Text = "User has signed-out";
 this.CallGraphButton.Visibility = Visibility.Visible;
 this.SignOutButton.Visibility = Visibility.Collapsed;
 }
 catch (MsalException ex)
 {
 ResultText.Text = $"Error signing-out user: {ex.Message}";
 }
 }
}

```

## More information about user sign-out

The `SignOutButton_Click` method removes users from the MSAL user cache, which effectively tells MSAL to forget the current user so that a future request to acquire a token will succeed only if it is made to be interactive.

Although the application in this sample supports single users, MSAL supports scenarios where multiple accounts can be signed in at the same time. An example is an email application where a user has multiple accounts.

## Display basic token information

To display basic information about the token, add the following method to your `MainWindow.xaml.cs` file:

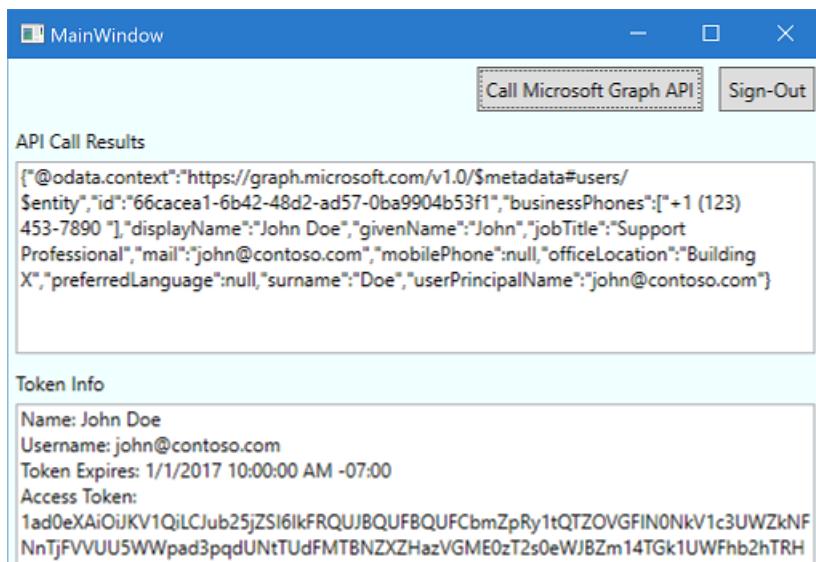
```
/// <summary>
/// Display basic information contained in the token
/// </summary>
private void DisplayBasicTokenInfo(AuthenticationResult authResult)
{
 TokenInfoText.Text = "";
 if (authResult != null)
 {
 TokenInfoText.Text += $"Username: {authResult.Account.Username}" + Environment.NewLine;
 TokenInfoText.Text += $"Token Expires: {authResult.ExpiresOn.ToLocalTime()}" + Environment.NewLine;
 }
}
```

### More information

In addition to the access token that's used to call the Microsoft Graph API, after the user signs in, MSAL also obtains an ID token. This token contain a small subset of information that's pertinent to users. The `DisplayBasicTokenInfo` method displays the basic information that's contained in the token. For example, it displays the user's display name and ID, as well as the token expiration date and the string representing the access token itself. You can select the *Call Microsoft Graph API* button multiple times and see that the same token was reused for subsequent requests. You can also see the expiration date being extended when MSAL decides it is time to renew the token.

## Test your code

To run your project, in Visual Studio, select **F5**. Your application **MainWindow** is displayed, as shown here:



The first time that you run the application and select the **Call Microsoft Graph API** button, you're prompted to sign in. Use an Azure Active Directory account (work or school account) or a Microsoft account (live.com, outlook.com) to test it.

# Test App

Work or school, or personal Microsoft account

[Sign in](#)

[Back](#)

[Can't access your account?](#)

[Other sign in options](#)

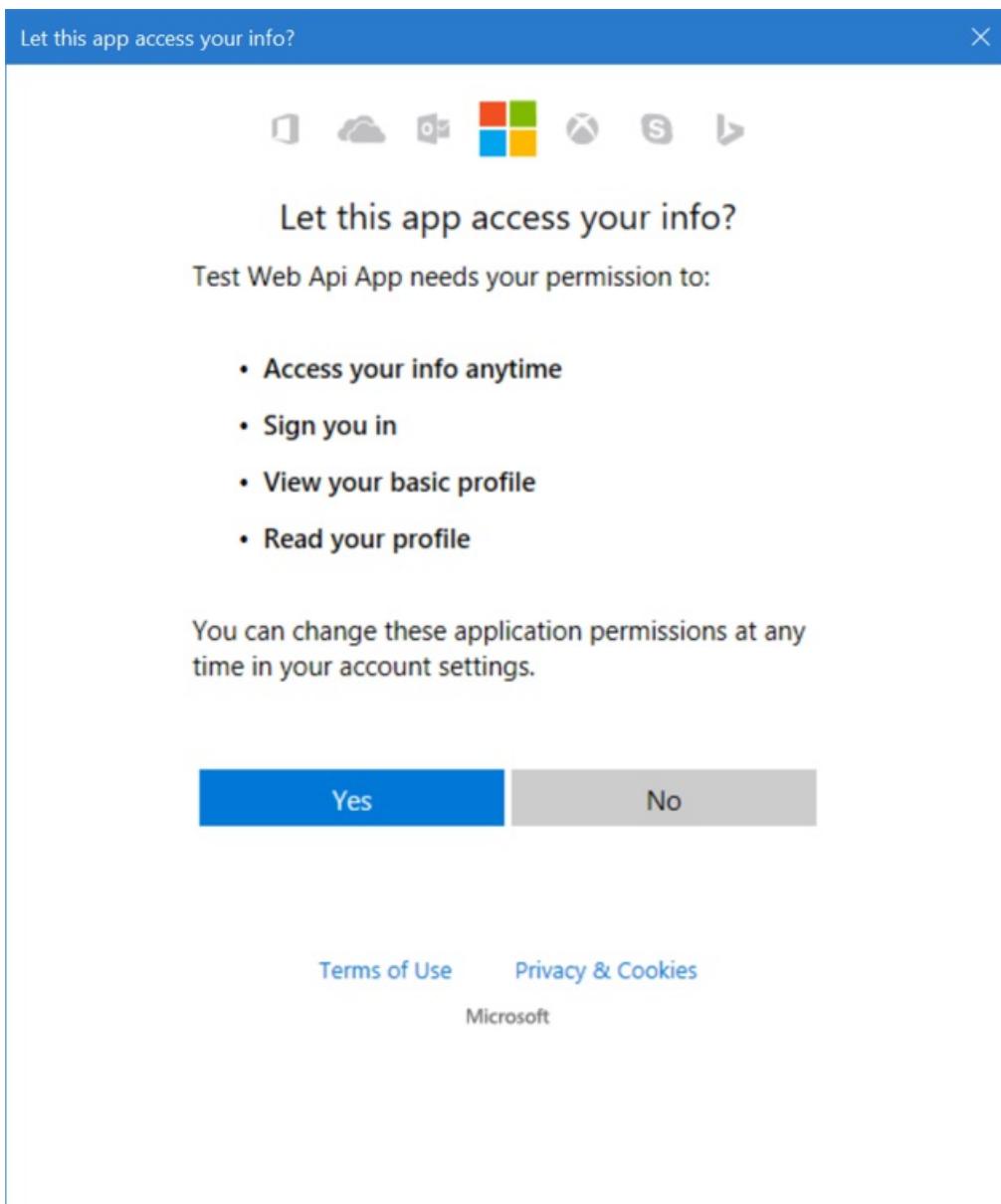
© 2017 Microsoft

[Terms of use](#) [Privacy & Cookies](#)



## Provide consent for application access

The first time that you sign in to your application, you're also prompted to provide consent to allow the application to access your profile and sign you in, as shown here:



## View application results

After you sign in, you should see the user profile information that's returned by the call to the Microsoft Graph API. The results are displayed in the **API Call Results** box. Basic information about the token that was acquired via the call to `AcquireTokenInteractive` or `AcquireTokenSilent` should be visible in the **Token Info** box. The results contain the following properties:

PROPERTY	FORMAT	DESCRIPTION
----------	--------	-------------

|**Username** |user@domain.com |The username that is used to identify the user.| **Token Expires** |DateTime |The time at which the token expires. MSAL extends the expiration date by renewing the token as necessary.|

## More information about scopes and delegated permissions

The Microsoft Graph API requires the `user.read` scope to read a user's profile. This scope is automatically added by default in every application that's registered in the Application Registration Portal. Other APIs for Microsoft Graph, as well as custom APIs for your back-end server, might require additional scopes. The Microsoft Graph API requires the `Calendars.Read` scope to list the user's calendars.

To access the user's calendars in the context of an application, add the `Calendars.Read` delegated permission to the application registration information. Then, add the `Calendars.Read` scope to the `acquireTokenSilent` call.

**NOTE**

The user might be prompted for additional consents as you increase the number of scopes.

## Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

Help us improve the Microsoft identity platform. Tell us what you think by completing a short two-question survey.

[Microsoft identity platform survey](#)

# Microsoft identity platform code samples (v2.0 endpoint)

11/12/2019 • 3 minutes to read • [Edit Online](#)

You can use Microsoft identity platform to:

- Add authentication and authorization to your web applications and web APIs.
- Require an access token to access a protected web API.

This article briefly describes and provides you with links to samples for the Microsoft identity platform endpoint. These samples show you how it's done, and also provide code snippets that you can use in your applications. On the code sample page, you'll find detailed readme topics that help with requirements, installation, and setup. Comments within the code help you understand the critical sections.

## NOTE

If you're interested in v1.0 samples, see [Azure AD code samples \(v1.0 endpoint\)](#).

To understand the basic scenario for each sample type, see [App types for the Microsoft identity platform endpoint](#).

You can also contribute to the samples on GitHub. To learn how, see [Microsoft Azure Active Directory samples and documentation](#).

## Single-page applications

These samples show how to write a single-page application secured with Microsoft identity platform. These samples use one of the flavors of MSAL.js.

PLATFORM	DESCRIPTION	LINK
 JavaScript (msal.js)	Calls Microsoft Graph	<a href="#">javascript-graphapi-web-v2</a>
 JavaScript (msal.js)	Calls B2C	<a href="#">b2c-javascript-msal-singlepageapp</a>
 JavaScript (msal.js)	Calls own web API	<a href="#">javascript-singlepageapp-dotnet-webapi-v2</a>
 JavaScript (MSAL AngularJS)	Calls Microsoft Graph	<a href="#">MsalAngularjsDemoApp</a>

PLATFORM	DESCRIPTION	LINK
 JavaScript (MSAL Angular)	Calls Microsoft Graph	<a href="#">MSALAngularDemoApp</a>

## Web applications

The following samples illustrate web applications that sign in users. Some samples also demonstrate the application calling Microsoft Graph, or your own web API with the user's identity.

PLATFORM	ONLY SIGNS IN USERS	SIGNS IN USERS AND CALLS MICROSOFT GRAPH	
 ASP.NET Core 2.2	<a href="#">ASP.NET Core WebApp signs-in users tutorial</a>	Same sample in the <a href="#">ASP.NET Core Web App calls Microsoft Graph</a> phase	
 ASP.NET	<a href="#">ASP.NET Quickstart dotnet-webapp-openidconnect-v2</a>	<code>dotnet-admin-restricted-scopes-v2</code>	<a href="#">msgraph-training-aspnetmvccapp</a>
		<a href="#">ms-identity-java-webapp</a> : an MSAL4J web app calling Microsoft graph	
		<a href="#">Node.js Quickstart</a>	
		<a href="#">msgraph-training-rubyrailsapp</a>	

## Desktop and mobile public client apps

The following samples show public client applications (desktop or mobile applications) that access the Microsoft Graph API, or your own web API in the name of a user. All these client applications use Microsoft Authentication Library (MSAL).

CLIENT APPLICATION	PLATFORM	FLOW/GRANT	CALLS MICROSOFT GRAPH	CALLS AN ASP.NET CORE 2.0 WEB API
Desktop (WPF)		interactive	<code>dotnet-desktop-msgraph-v2</code>	<code>dotnet-native-aspnetcore-v2</code>

CLIENT APPLICATION	PLATFORM	FLOW/GRANT	CALLS MICROSOFT GRAPH	CALLS AN ASP.NET CORE 2.0 WEB API
Desktop (Console)		Integrated Windows Authentication	dotnet-iwa-v2	
Desktop (Console)		Username/Password	dotnetcore-up-v2	
Mobile (Android, iOS, UWP)		interactive	xamarin-native-v2	
Mobile (iOS)		interactive	ios-swift-objc-native-v2 ios-native-nxoauth2-v2	
Desktop (macOS)	macOS	interactive	macOS-swift-objc-native-v2	
Mobile (Android-Java)		interactive	android-Java	
Mobile (Android-Kotlin)		interactive	android-Kotlin	

## Daemon applications

The following samples show an application that accesses the Microsoft Graph API with its own identity (with no user).

CLIENT APPLICATION	PLATFORM	FLOW/GRANT	CALLS MICROSOFT GRAPH
Console	 ASP.NET	Client Credentials	dotnetcore-daemon-v2
Web app	 ASP.NET	Client Credentials	dotnet-daemon-v2

## Headless applications

The following sample shows a public client application running on a device without a web browser. The app can be a command-line tool, an app running on Linux or Mac, or an IoT application. The sample features an app accessing

the Microsoft Graph API, in the name of a user who signs-in interactively on another device (such as a mobile phone). This client application uses Microsoft Authentication Library (MSAL).

CLIENT APPLICATION	PLATFORM	FLOW/GANT	CALLS MICROSOFT GRAPH
Desktop (Console)		Device code flow	<a href="#">dotnetcore-devicecodeflow-v2</a>

## Web APIs

The following samples show how to protect a web API with the Microsoft identity platform endpoint, and how to call a downstream API from the web API.

PLATFORM	SAMPLE
 ASP.NET Core 2.2	ASP.NET Core web API (service) of <a href="#">dotnet-native-aspnetcore-v2</a>
 ASP.NET MVC	Web API (service) of <a href="#">ms-identity-aspnet-webapi-onbehalfof</a>

## Azure Functions as web APIs

The following samples show how to protect an Azure Function using HttpTrigger and exposing a web API with the Microsoft identity platform endpoint, and how to call a downstream API from the web API.

PLATFORM	SAMPLE
 ASP.NET Core 2.2	ASP.NET Core web API (service) Azure Function of <a href="#">dotnet-native-aspnetcore-v2</a>
 NodeJS	Web API (service) of <a href="#">NodeJS and passport-azure-ad</a>
 Python	Web API (service) of <a href="#">Python</a>

PLATFORM	SAMPLE
 NodeJS	Web API (service) of <a href="#">NodeJS</a> and <a href="#">passport-azure-ad</a> using on behalf of

## Other Microsoft Graph samples

To learn about [samples](#) and tutorials that demonstrate different usage patterns for the Microsoft Graph API, including authentication with Azure AD, see [Microsoft Graph Community samples & tutorials](#).

## See also

- [Azure Active Directory \(v1.0\) developer's guide](#)
- [Azure AD Graph API conceptual and reference](#)
- [Azure AD Graph API Helper Library](#)

# Authentication basics

11/12/2019 • 11 minutes to read • [Edit Online](#)

## What is authentication

This article covers many of the authentication concepts you'll need to understand to create protected web apps, web APIs, or apps calling protected Web APIs.

**Authentication** is the process of proving you are who you say you are. Authentication is sometimes shortened to AuthN.

**Authorization** is the act of granting an authenticated party permission to do something. It specifies what data you're allowed to access and what you can do with that data. Authorization is sometimes shortened to AuthZ.

Instead of creating apps that each maintain their own username and password information, which incurs a high administrative burden when you need to add or remove users across multiple apps, apps can delegate that responsibility to a centralized identity provider.

Azure Active Directory (Azure AD) is a centralized identity provider in the cloud. Delegating authentication and authorization to it enables scenarios such as Conditional Access policies that require a user to be in a specific location, the use of multi-factor authentication, as well as enabling a user to sign in once and then be automatically signed in to all of the web apps that share the same centralized directory. This capability is referred to as Single Sign On (SSO).

A centralized identity provider is even more important for apps that have users located around the globe that don't necessarily sign in from the enterprise's network. Azure AD authenticates users and provides access tokens. An access token is a security token that is issued by an authorization server. It contains information about the user and the app for which the token is intended, which can be used to access Web APIs and other protected resources.

The Microsoft identity platform simplifies authentication for application developers by providing identity as a service, with support for industry-standard protocols such as OAuth 2.0 and OpenID Connect, as well as open-source libraries for different platforms to help you start coding quickly. 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. For more information, see [Evolution of Microsoft identity platform](#).

### Tenants

A cloud identity provider serves many organizations. To keep users from different organizations separate, Azure AD is partitioned into tenants, with one tenant per organization.

Tenants keep track of users and their associated apps. The Microsoft identity platform also supports users that sign in with personal Microsoft accounts.

Azure AD also provides Azure Active Directory B2C so that organizations can sign in users, typically customers, using social identities like a Google account. For more information, see [Azure Active Directory B2C documentation](#).

### Security tokens

Security tokens contain information about users and apps. Azure AD uses JSON based tokens (JWTs) that contain claims. A claim provides assertions about one entity to another. Applications can use claims for various tasks such as:

- Validating the token

- Identifying the subject's directory tenant
- Displaying user information
- Determining the subject's authorization

A claim consists of key-value pairs that provide information such as:

- the Security Token Server that generated the token.
- the date when the token was generated.
- the subject, such as the user (except for daemons).
- the audience, which is the app for which the token was generated.
- the app (the client) that asked for the token. In the case of web apps, this may be the same as the audience.

For more detailed claim information, see the [access tokens](#) and [ID tokens](#).

It's up to the app for which the token was generated, the web app that signed-in the user, or the Web API being called, to validate the token. The token is signed by the Security Token Server (STS) with a private key. The STS publishes the corresponding public key. To validate a token, the app verifies the signature by using the STS public key to validate that the signature was created using the private key.

Tokens are only valid for a limited amount of time. Usually the STS provides a pair of tokens: an access token to access the application or protected resource, and a refresh token used to refresh the access token when the access token is close to expiring.

Access tokens are passed to a Web API as the bearer token in the `Authorization` header. An app can provide a refresh token to the STS, and if the user access to the app wasn't revoked, it will get back a new access token and a new refresh token. This is how the scenario of someone leaving the enterprise is handled. When the STS receives the refresh token, it won't issue another valid access token if the user is no longer authorized.

## Application model

Applications can sign in users themselves or delegate sign-in to an identity provider. See [Authentication flows and app scenarios](#) to learn about sign-in scenarios supported by Azure AD.

For an identity provider to know that a user has access to a particular app, both the user and the application must be registered with the identity provider. When you register your application with Azure AD, you are providing an identity configuration for your application that allows it to integrate with Azure AD. Registering the app also allows you to:

- customize the branding of your application in the sign-in dialog. This is important because this is the first experience a user will have with your app.
- decide if you want to let users sign in only if they belong to your organization. This is a single tenant application. Or allow users to sign in using any work or school account. This is a multi-tenant application. You can also allow personal Microsoft accounts, or a social account from LinkedIn, Google, and so on.
- request scope permissions. For example, you can request the "user.read" scope, which grants permission to read the profile of the signed-in user.
- define scopes that define access to your Web API. Typically, when an app wants to access your API, it will need to request permissions to the scopes you define.
- share a secret with Azure AD that proves the app's identity to Azure AD. This is relevant in the case where the app is a confidential client application. A confidential client application is an application that can hold credentials securely. They require a trusted backend server to store the credentials.

Once registered, the application will be given a GUID that the app shares with Azure AD when it requests tokens. If the app is a confidential client application, it will also share the secret or the public key, depending on whether certificates or secrets were used.

The Microsoft identity platform represents applications using a model that fulfills two main functions:

Identify the app by the authentication protocols it supports and provide all the identifiers, URLs, secrets, and related information that are needed to authenticate. The Microsoft identity platform:

- Holds all the data required to support authentication at runtime.
- Holds all the data for deciding what resources an app might need to access, and under what circumstances a given request should be fulfilled.
- Provides infrastructure for implementing app provisioning within the app developer's tenant, and to any other Azure AD tenant.
- Handles user consent during token request time and facilitate the dynamic provisioning of apps across tenants

Consent is the process of a resource owner granting authorization for a client application to access protected resources, under specific permissions, on behalf of the resource owner. The Microsoft identity platform:

- Enables users and administrators to dynamically grant or deny consent for the app to access resources on their behalf.
- Enables administrators to ultimately decide what apps are allowed to do and which users can use specific apps, and how the directory resources are accessed.

In the Microsoft identity platform, an **application object** describes an application as an abstract entity. At deployment time, the Microsoft identity platform uses the application object as a blueprint to create a **service principal**, which represents a concrete instance of an application within a directory or tenant. The service principal defines what the app can actually do in a specific target directory, who can use it, what resources it has access to, and so on. The Microsoft identity platform creates a service principal from an application object through **consent**.

The following diagram shows a simplified Microsoft identity platform provisioning flow driven by consent. It shows two tenants (A and B). Tenant A owns the application. Tenant B is instantiating the application via a service principal.

In this provisioning flow:

1. A user from tenant B attempts to sign in with the app, the authorization endpoint requests a token for the application.
2. The user credentials are acquired and verified for authentication.
3. The user is prompted to provide consent for the app to gain access to tenant B.
4. The Microsoft identity platform uses the application object in tenant A as a blueprint for creating a service principal in tenant B.
5. The user receives the requested token.

You can repeat this process for additional tenants. Tenant A retains the blueprint for the app (application object). Users and admins of all the other tenants where the app is given consent keep control over what the application is allowed to do via the corresponding service principal object in each tenant. For more information, see [Application and service principal objects in Microsoft identity platform](#).

## Web app sign-in flow with Azure AD

When a user navigates in the browser to a web app, the following happens:

- The web app determines whether the user is authenticated.
- If the user isn't authenticated, the web app delegates to Azure AD to sign in the user. That sign in will be compliant with the policy of the organization, which may mean asking the user to enter their credentials,

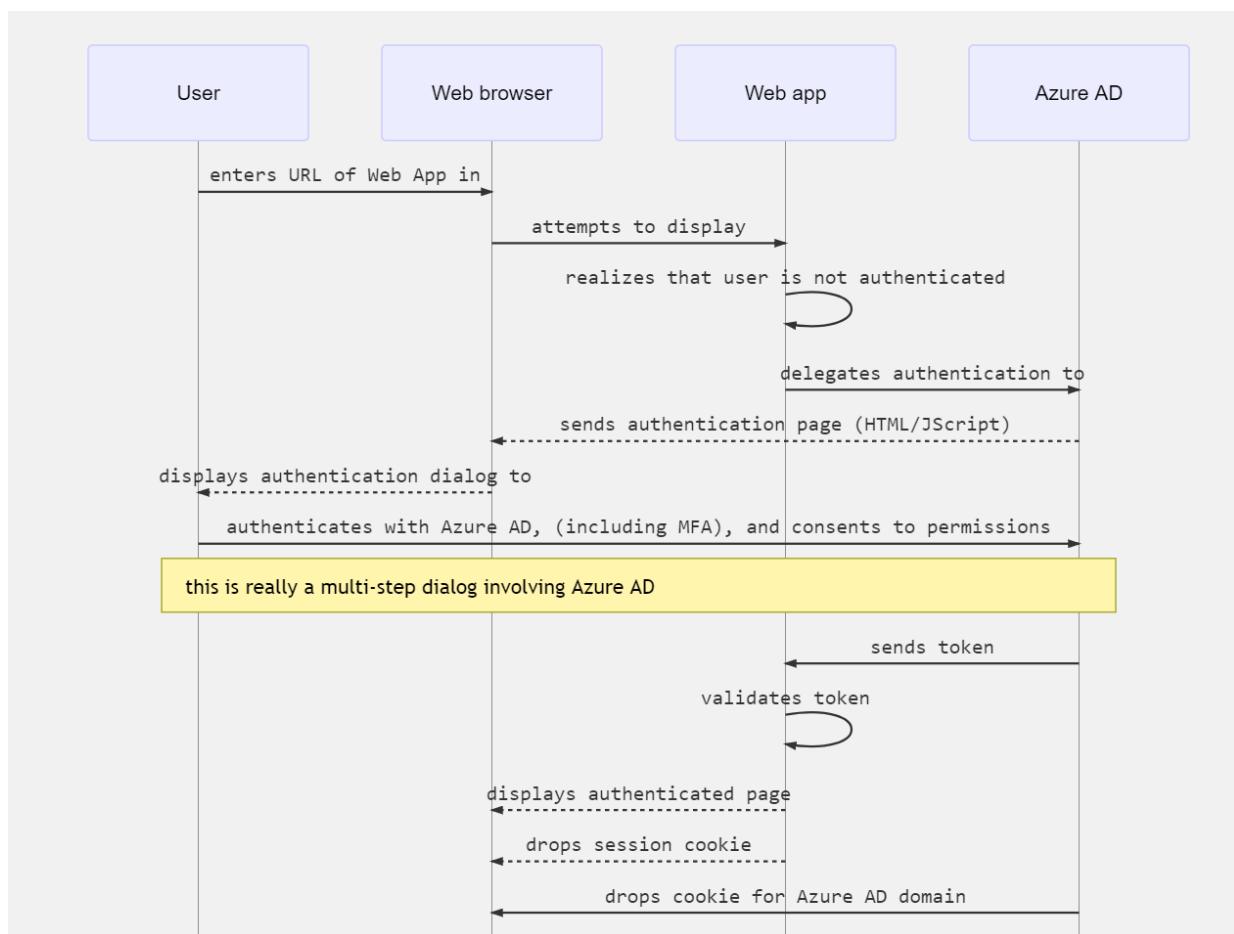
using multi-factor-authentication, or not using a password at all (for example using Windows Hello).

- The user is asked to consent to the access that the client app needs. This is why client apps need to be registered with Azure AD, so that Azure AD can deliver tokens representing the access that the user has consented to.

When the user has successfully authenticated:

- Azure AD sends a token to the web app.
- A cookie is saved, associated with Azure AD's domain, that contains the identity of the user in the browser's cookie jar. The next time an app uses the browser to navigate to the Azure AD authorization end point, the browser presents the cookie so that the user doesn't have to sign in again. This is also the way that SSO is achieved. The cookie is produced by Azure AD and can only be understood by Azure AD.
- The web app then validates the token. If the validation succeeds, the web app displays the protected page and saves a session cookie in the browser's cookie jar. When the user navigates to another page, the web app knows that the user is authenticated based on the session cookie.

The following sequence diagram summarizes this interaction:



### How a web app determines if the user is authenticated

Web app developers can indicate whether all or only certain pages require authentication. For example, in ASP.NET/ASP.NET Core, this is done by adding the `[Authorize]` attribute to the controller actions.

This attribute causes ASP.NET to check for the presence of a session cookie containing the identity of the user. If a cookie isn't present, ASP.NET redirects authentication to the specified identity provider. If the identity provider is Azure AD, the web app redirects authentication to <https://login.microsoftonline.com>, which displays a sign-in dialog.

### How a web app delegates sign-in to Azure AD and obtains a token

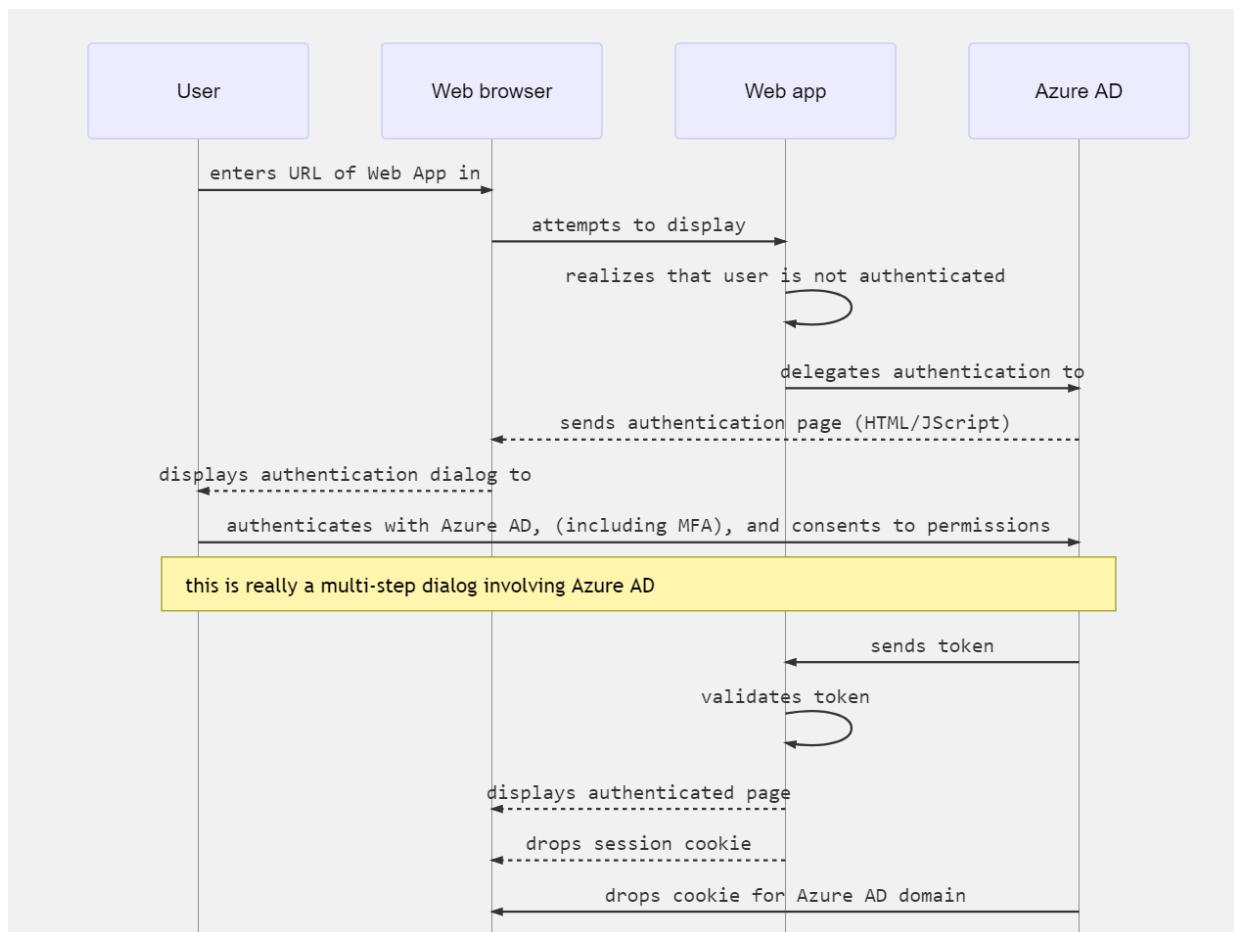
User authentication happens via the browser. The OpenID protocol uses standard HTTP protocol messages.

- The web app sends an HTTP 202 (redirect) to the browser to use Azure AD.
- When the user is authenticated, Azure AD sends the token to the web app by using a redirect through the browser.
- The redirect is provided by the web app in the form of a redirect URI. This redirect URI is registered with the Azure AD application object. There can be several redirect URIs because the application may be deployed at several URLs. So the web app will also need to specify the redirect URI to use.
- Azure AD verifies that the redirect URI sent by the web app is one of the registered redirect URIs for the app.

## Desktop and mobile app sign-in flow with Azure AD

The flow described above applies, with slight differences, to desktop and mobile applications.

Desktop and mobile applications can use an embedded Web control, or a system browser, for authentication. The following diagram shows how a Desktop or mobile app uses the Microsoft authentication library (MSAL) to acquire access tokens and call web APIs.



MSAL uses a browser to get tokens, and as with web apps, delegates authentication to Azure AD.

Because Azure AD saves the same identity cookie in the browser as it does for web apps, if the native or mobile app uses the system browser it will immediately get SSO with the corresponding web app.

By default, MSAL uses the system browser except for .NET Framework desktop applications where an embedded control is used to provide a more integrated user experience.

## Next steps

See the [Microsoft identity platform developer glossary](#) to get familiar with common terms. See [Authentication flows and app scenarios](#) to learn more about other scenarios for authenticating users supported by the Microsoft identity platform. See [MSAL libraries](#) to learn about the Microsoft libraries that help you develop

applications that work with Microsoft Accounts, Azure AD accounts, and Azure AD B2C users all in a single, streamlined programming model.

# Authentication flows and application scenarios

11/4/2019 • 10 minutes to read • [Edit Online](#)

The Microsoft identity platform (v2.0) endpoint supports authentication for different kinds of modern application architectures. All of the architectures are based on the industry-standard protocols [OAuth 2.0](#) and [OpenID Connect](#). Using the [authentication libraries](#), applications authenticate identities and acquire tokens to access protected APIs.

This article describes the different authentication flows and the application scenarios they're used in. This article also provides lists of:

- [Application scenarios and supported authentication flows](#).
- [Application scenarios and supported platforms and languages](#).

## Application categories

Tokens can be acquired from several types of applications including:

- Web apps
- Mobile apps
- Desktop apps
- Web APIs

They can also be acquired from apps running on devices that don't have a browser or are running on IoT.

Applications can be categorized as in the following list:

- [Protected resources vs. client applications](#): Some scenarios are about protecting resources like web apps or web APIs. Other scenarios are about acquiring a security token to call a protected web API.
- [With users or without users](#): Some scenarios involve a signed-in user, but others like daemon scenarios don't involve a user.
- [Single-page, public client, and confidential client applications](#): These are three large categories of application types. Each is used with different libraries and objects.
- [Sign-in audience](#): The available authentication flows differ depending on the sign-in audience. Some flows are available only for work or school accounts. And some are available both for work or school accounts and for personal Microsoft accounts. The allowed audience depends on the authentication flows.
- [Supported OAuth 2.0 flows](#): Authentication flows are used to implement the application scenarios that are requesting tokens. There isn't a one-to-one mapping between application scenarios and authentication flows.
- [Supported platforms](#): Not all application scenarios are available for every platform.

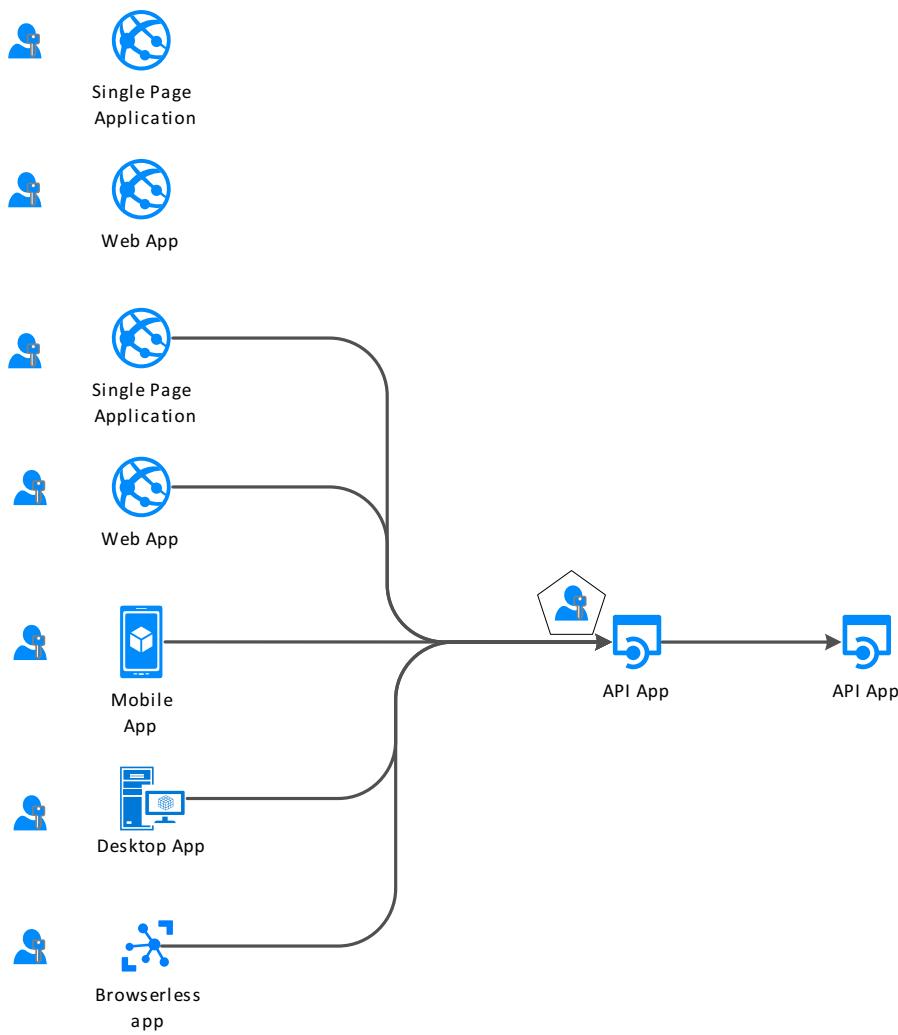
### Protected resources vs. client applications

Authentication scenarios involve two activities:

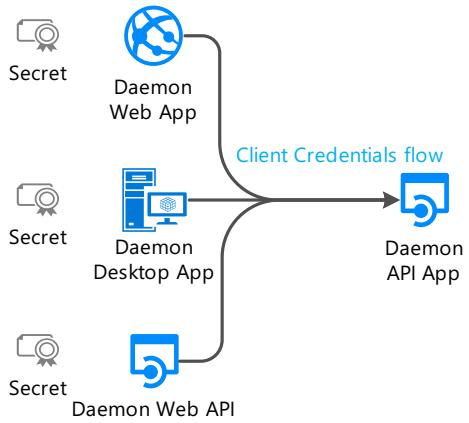
- **Acquiring security tokens for a protected web API**: Microsoft recommends that you use [authentication libraries](#) to acquire tokens, in particular the Microsoft Authentication Library (MSAL) family.
- **Protecting a web API or a web app**: One challenge of protecting a web API or web app resource is validating the security token. On some platforms, Microsoft offers [middleware libraries](#).

### With users or without users

Most authentication scenarios acquire tokens on behalf of signed-in users.



However, there are also daemon-app scenarios, in which applications acquire tokens on behalf of themselves with no user.



### Single-page, public client, and confidential client applications

The security tokens can be acquired from multiple types of applications. These applications tend to be separated into three categories:

- **Single-page applications:** Also known as SPAs, these are web apps in which tokens are acquired from a JavaScript or TypeScript app running in the browser. Many modern apps have a single-page application front end that is primarily written in JavaScript. The application often uses a framework like Angular, React, or Vue. MSAL.js is the only Microsoft authentication library that supports single-page applications.
- **Public client applications:** These applications always sign in users:
  - Desktop apps calling web APIs on behalf of the signed-in user

- Mobile apps
  - Apps running on devices that don't have a browser, like those running on IoT
- These apps are represented by the MSAL [PublicClientApplication](#) class.

- **Confidential client applications:**

- Web apps calling a web API
- Web APIs calling a web API
- Daemon apps, even when implemented as a console service like a Linux daemon or a Windows service

These types of apps use the [ConfidentialClientApplication](#) class.

## Application scenarios

The Microsoft identity platform endpoint supports authentication for different kinds of app architectures:

- Single-page apps
- Web apps
- Web APIs
- Mobile apps
- Native apps
- Daemon apps
- Server-side apps

Applications use the different authentication flows to sign in users and get tokens to call protected APIs.

### A single-page application

Many modern web apps are built as client-side single-page applications written using JavaScript or an SPA framework like Angular, Vue.js, and React.js. These applications run in a web browser. Their authentication characteristics differ from those of traditional server-side web apps. By using the Microsoft identity platform, single-page applications can sign in users and get tokens to access back-end services or web APIs.



For more information, see [Single-page applications](#).

### A web app that is signing in a user



To protect a web app that is signing in a user:

- If you develop in .NET, you use ASP.NET or ASP.NET Core with the ASP.NET Open ID Connect middleware. Protecting a resource involves validating the security token, which is done by the [IdentityModel extensions for .NET](#) library and not MSAL libraries.
- If you develop in Node.js, you use Passport.js.

For more information, see [Web app that signs in users](#).

### A web app that signs in a user and calling a web API on behalf of the user



To call a web API from a web app on behalf of a user, use the MSAL **ConfidentialClientApplication** class. You use the Authorization code flow and store the acquired tokens in the token cache. When needed, MSAL refreshes tokens and the controller silently acquires tokens from the cache.

For more information, see [A web app calling web APIs](#).

### A desktop app calling a web API on behalf of a signed-in user

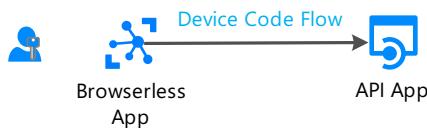
For a desktop app to call a web API that signs in users, use the interactive token-acquisition methods of the MSAL **PublicClientApplication** class. With these interactive methods, you can control the sign-in UI experience. MSAL uses a web browser for this interaction.

- Authorization code flow (with PKCE)
- Integrated Windows Authentication
- Username/Password



There's another possibility for Windows-hosted applications on computers joined either to a Windows domain or by Azure Active Directory (Azure AD). These applications can silently acquire a token by using [Integrated Windows Authentication](#).

Applications running on a device without a browser can still call an API on behalf of a user. To authenticate, the user must sign in on another device that has a web browser. This scenario requires that you use the [Device Code flow](#).



Though we don't recommend you use it, the [Username/Password flow](#) is available in public client applications. This flow is still needed in some scenarios like DevOps.

But using this flow imposes constraints on your applications. For instance, applications using this flow can't sign in a user who needs to perform multi-factor authentication or Conditional Access. Your applications also don't benefit from single sign-on.

Authentication with the Username/Password flow goes against the principles of modern authentication and is provided only for legacy reasons.

In desktop apps, if you want the token cache to be persistent, you should [customize the token cache serialization](#). By implementing [dual token cache serialization](#), you can use backward-compatible and forward-compatible token caches with previous generations of authentication libraries. Specific libraries include Azure AD Authentication Library for .NET (ADAL.NET) version 3 and version 4.

For more information, see [Desktop app that calls web APIs](#).

### A mobile app calling a web API on behalf of an interactive user

Similar to a desktop app, a mobile app calls the interactive token-acquisition methods of the MSAL **PublicClientApplication** class to acquire a token for calling a web API.



MSAL iOS and MSAL Android use the system web browser by default. However, you can direct them to use the embedded Web View instead. There are specificities that depend on the mobile platform: Universal Windows Platform (UWP), iOS, or Android.

Some scenarios, like those that involve Conditional Access related to a device ID or a device enrollment, require a [broker](#) to be installed on the device. Examples of brokers are Microsoft Company Portal on Android and Microsoft Authenticator on Android and iOS. Also, MSAL can now interact with brokers.

#### **NOTE**

Your mobile app that uses MSAL.iOS, MSAL.Android, or MSAL.NET on Xamarin can have app protection policies applied to it. For instance, the policies might prevent a user from copying protected text. The mobile app is [managed by Intune](#) and recognized by Intune as a managed app. The [Intune App SDK](#) is separate from MSAL libraries and interacts with Azure AD on its own.

For more information, see [Mobile app that calls web APIs](#).

#### **A protected web API**

You can use the Microsoft Identity Platform endpoint to secure web services like your app's RESTful web API. A protected web API is called with an access token to secure the API's data and to authenticate incoming requests. The caller of a web API appends an access token in the authorization header of an HTTP request.

If you want to protect your ASP.NET or ASP.NET Core Web API, you need to validate the access token. For this validation, you use the ASP.NET JWT middleware. The validation is done by the [IdentityModel extensions for .NET](#) library and not by MSAL.NET.

For more information, see [Protected web API](#).

#### **A web API calling another web API on behalf of a user**

For your ASP.NET or ASP.NET Core protected Web API to call another web API on behalf of a user, your app needs to acquire a token for the downstream web API. It does so by calling the **ConfidentialClientApplication** class's [AcquireTokenOnBehalfOf](#) method. Such calls are also named service-to-services calls. The web APIs that call other web APIs need to provide custom cache serialization.

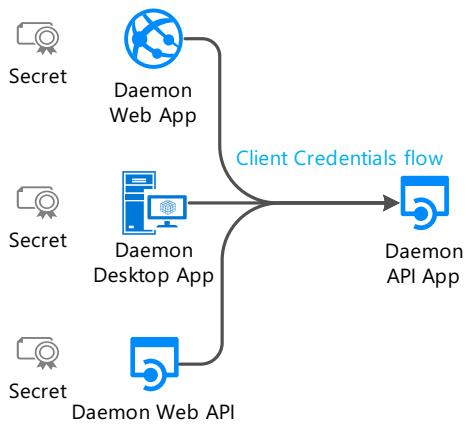


For more information, see [Web API that calls web APIs](#).

#### **A daemon app calling a web API in the daemon's name**

Apps that have long-running processes or that operate without user interaction also need a way to access secure web APIs. Such an app can authenticate and get tokens by using the app's identity rather than a user's delegated identity. The app proves its identity by using a client secret or certificate.

You can write such daemon apps that acquire a token for the calling app by using the MSAL **ConfidentialClientApplication** class's [client credentials](#) acquisition methods. These methods require that the calling app has registered a secret with Azure AD. The app then shares the secret with the called daemon. Examples of such secrets include application passwords, certificate assertion, or client assertion.



For more information, see [Daemon application that calls web APIs](#).

## Scenarios and supported authentication flows

Scenarios that involve acquiring tokens also map to OAuth 2.0 authentication flows, as detailed in [Microsoft identity platform protocols](#).

SCENARIO	DETAILED SCENARIO WALK-THROUGH	OAUTH 2.0 FLOW AND GRANT	AUDIENCE
   Single Page Application	Single-page app	Implicit	Work or school accounts, personal accounts, and Microsoft Azure Active Directory B2C (Azure AD B2C)
  Web App	A web app that signs in users	Authorization Code	Work or school accounts, personal accounts, and Azure AD B2C
   Web App	A web app that calls web APIs	Authorization Code	Work or school accounts, personal accounts, and Azure AD B2C
   - Authorization code flow (with PKCE) - Integrated Windows Authentication - Username/Password	A desktop app that calls web APIs  Interactive by using <a href="#">Authorization Code</a> with PKCE	Interactive by using <a href="#">Authorization Code</a> with PKCE	Work or school accounts, personal accounts, and Azure AD B2C
		Integrated Windows Auth	Work or school accounts
		Resource Owner Password	Work or school accounts and Azure AD B2C
   Browserless App	Device code	Device code	Work or school accounts
   Mobile App	A mobile app that calls web APIs  Interactive by using <a href="#">Authorization Code</a> with PKCE	Interactive by using <a href="#">Authorization Code</a> with PKCE	Work or school accounts, personal accounts, and Azure AD B2C
		Resource Owner Password	Work or school accounts and Azure AD B2C

SCENARIO	DETAILED SCENARIO WALK-THROUGH	OAuth 2.0 FLOW AND GRANT	AUDIENCE
	A daemon app that calls web APIs	Client credentials	App-only permissions with no user and used only in Azure AD organizations
	A web API that calls web APIs	On Behalf Of	Work or school accounts and personal accounts

## Scenarios and supported platforms and languages

Microsoft Authentication libraries support multiple platforms:

- JavaScript
- .NET Framework
- .NET Core
- Windows 10/UWP
- Xamarin.iOS
- Xamarin.Android
- Native iOS
- macOS
- Native Android
- Java
- Python

You can also use various languages to build your applications. Note that some application types aren't available on every platform.

In the Windows column of the following table, each time .NET Core is mentioned, .NET Framework is also possible. The latter is omitted to avoid cluttering the table.

SCENARIO	WINDOWS	LINUX	MAC	IOS	ANDROID
	MSAL.js	MSAL.js	MSAL.js	MSAL.js	MSAL.js
	ASP.NET Core	ASP.NET Core	ASP.NET Core		

SCENARIO	WINDOWS	LINUX	MAC	IOS	ANDROID
Web App that calls web APIs	   <i>Authorization code flow</i> Web App → API App	 ASP.NET Core + MSAL.NET  MSAL Java  Flask + MSAL Python	 ASP.NET Core + MSAL.NET  MSAL Java  Flask + MSAL Python		
Desktop app that calls web APIs	   <i>Authorization code flow (with PKCE)</i> <i>Integrated Windows Authentication</i> <i>Username/Password</i> Desktop App → API App	 MSAL.NET  MSAL Java  MSAL Python	 MSAL.NET  MSAL Java  MSAL Python	 MSAL.NET  MSAL Java  MSAL Python	
	   <i>Device Code Flow</i> Browserless App → API App			iOS	MSAL.objc
Mobile app that calls web APIs	   <i>Authorization code flow (with PKCE)</i> Mobile App → API App	 MSAL.NET  MSAL.NET		iOS	MSAL.Android
Daemon app	     <i>Client Credentials flow</i> Secret → Daemon Web App → Daemon Desktop App → Daemon API App	 MSAL.NET  MSAL Java  MSAL Python	 MSAL.NET  MSAL Java  MSAL Python		
Web API that calls web APIs	  <i>On behalf of flow</i> API App → API App	 ASP.NET Core + MSAL.NET  MSAL Java  MSAL Python	 ASP.NET Core + MSAL.NET  MSAL Java  MSAL Python		

See also [Microsoft-supported libraries by OS / language](#).

## Next steps

Learn more about [authentication basics](#) and [access tokens](#).

# Microsoft identity platform best practices and recommendations

10/31/2019 • 6 minutes to read • [Edit Online](#)

This article highlights best practices, recommendations, and common oversights when integrating with the Microsoft identity platform. This checklist will guide you to a high-quality and secure integration. Review this list on a regular basis to make sure you maintain the quality and security of your app's integration with the identity platform. The checklist isn't intended to review your entire application. The contents of the checklist are subject to change as we make improvements to the platform.

If you're just getting started, check out the [Microsoft identity platform documentation](#) to learn about authentication basics, application scenarios in the Microsoft identity platform, and more.

Use the following checklist to ensure that your application is effectively integrated with the [Microsoft identity platform](#).

## Basics



Read and understand the [Microsoft Platform Policies](#). Ensure that your application adheres to the terms outlined as they're designed to protect users and the platform.

## Ownership



Make sure the information associated with the account you used to register and manage apps is up-to-date.

## Branding



Adhere to the [Branding guidelines for applications](#).



Provide a meaningful name and logo for your application. This information appears on your [application's consent prompt](#). Make sure your name and logo are representative of your company/product so that users can make informed decisions. Ensure that you're not violating any trademarks.

## Privacy



Provide links to your app's terms of service and privacy statement.

# Security

<input type="checkbox"/>	<p>Manage your redirect URLs:</p> <ul style="list-style-type: none"><li>• Maintain ownership of all your redirect URLs and keep the DNS records for them up-to-date.</li><li>• Don't use wildcards (*) in your URLs.</li><li>• For web apps, make sure all URLs are secure and encrypted (for example, using https schemes).</li><li>• For public clients, use platform-specific redirect URLs if applicable (mainly for iOS and Android). Otherwise, use redirect URLs with a high amount of randomness to prevent collisions when calling back to your app.</li><li>• If your app is being used from an isolated web agent, you may use <a href="https://login.microsoftonline.com/common/oauth2/nativeclient">https://login.microsoftonline.com/common/oauth2/nativeclient</a>.</li><li>• Review and trim all unused or unnecessary redirect URLs on a regular basis.</li></ul>
<input type="checkbox"/>	If your app is registered in a directory, minimize and manually monitor the list of app registration owners.
<input type="checkbox"/>	Don't enable support for the <a href="#">OAuth2 implicit grant flow</a> unless explicitly required. Learn about the valid scenario <a href="#">here</a> .
<input type="checkbox"/>	Move beyond username/password. Don't use <a href="#">resource owner password credential flow (ROPC)</a> , which directly handles users' passwords. This flow requires a high degree of trust and user exposure and should only be used when other, more secure, flows can't be used. This flow is still needed in some scenarios (like DevOps), but beware that using it will impose constraints on your application. For more modern approaches, read <a href="#">Authentication flows and application scenarios</a> .
<input type="checkbox"/>	Protect and manage your confidential app credentials for web apps, web APIs and daemon apps. Use <a href="#">certificate credentials</a> , not password credentials (client secrets). If you must use a password credential, don't set it manually. Don't store credentials in code or config, and never allow them to be handled by humans. If possible, use <a href="#">managed identities for Azure resources</a> or <a href="#">Azure Key Vault</a> to store and regularly rotate your credentials.
<input type="checkbox"/>	Make sure your application requests the least privilege permissions. Only ask for permissions that your application absolutely needs, and only when you need them. Understand the different <a href="#">types of permissions</a> . Only use application permissions if necessary; use delegated permissions where possible. For a full list of Microsoft Graph permissions, see this <a href="#">permissions reference</a> .
<input type="checkbox"/>	If you're securing an API using the Microsoft identity platform, carefully think through the permissions it should expose. Consider what's the right granularity for your solution and which permission(s) require admin consent. Check for expected permissions in the incoming tokens before making any authorization decisions.

# Implementation

<input type="checkbox"/>	<p>Use modern authentication solutions (OAuth 2.0, <a href="#">OpenID Connect</a>) to securely sign in users.</p>
<input type="checkbox"/>	<p>Don't program directly against protocols such as OAuth 2.0 and Open ID. Instead, leverage the <a href="#">Microsoft Authentication Library (MSAL)</a>. The MSAL libraries securely wrap security protocols in an easy-to-use library, and you get built-in support for <a href="#">Conditional Access</a> scenarios, device-wide <a href="#">single sign-on (SSO)</a>, and built-in token caching support. For more info, see the list of Microsoft supported <a href="#">client libraries</a> and <a href="#">middleware libraries</a> and the list of <a href="#">compatible third-party client libraries</a>.</p> <p>If you must hand code for the authentication protocols, you should follow a methodology such as <a href="#">Microsoft SDL</a>. Pay close attention to the security considerations in the standards specifications for each protocol.</p>
<input type="checkbox"/>	<p>Migrate existing apps from <a href="#">Azure Active Directory Authentication Library (ADAL)</a> to <a href="#">Microsoft Authentication Library</a>. MSAL is Microsoft's latest identity platform solution and is preferred to ADAL. It is available on .NET, JavaScript, Android, iOS, macOS and is also in public preview for Python and Java. Read more about migrating <a href="#">ADAL.NET</a>, <a href="#">ADAL.js</a>, and <a href="#">ADAL.NET and iOS broker apps</a>.</p>
<input type="checkbox"/>	<p>For mobile apps, configure each platform using the application registration experience. In order for your application to take advantage of the Microsoft Authenticator or Microsoft Company Portal for single sign-in, your app needs a "broker redirect URI" configured. This allows Microsoft to return control to your application after authentication. When configuring each platform, the app registration experience will guide you through the process. Use the quickstart to download a working example. On iOS, use brokers and system webview whenever possible.</p>
<input type="checkbox"/>	<p>In web apps or web APIs, keep one token cache per account. For web apps, the token cache should be keyed by the account ID. For web APIs, the account should be keyed by the hash of the token used to call the API. MSAL.NET provides custom token cache serialization in the .NET Framework and .NET Core subplatforms. For security and performance reasons, our recommendation is to serialize one cache per user. For more information, read about <a href="#">token cache serialization</a>.</p>
<input type="checkbox"/>	<p>If the data your app requires is available through <a href="#">Microsoft Graph</a>, request permissions for this data using the Microsoft Graph endpoint rather than the individual API.</p>

<input type="checkbox"/>	Don't look at the access token value, or attempt to parse it as a client. They can change values, formats, or even become encrypted without warning - always use the id_token if your client needs to learn something about the user, or call Microsoft Graph. Only web APIs should parse access tokens (since they are the ones defining the format and setting the encryption keys).
--------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## End-user experience

<input type="checkbox"/>	<a href="#">Understand the consent experience</a> and configure the pieces of your app's consent prompt so that end users and admins have enough information to determine if they trust your app.
<input type="checkbox"/>	Minimize the number of times a user needs to enter login credentials while using your app by attempting silent authentication (silent token acquisition) before interactive flows.
<input type="checkbox"/>	Don't use "prompt=consent" for every sign-in. Only use prompt=consent if you've determined that you need to ask for consent for additional permissions (for example, if you've changed your app's required permissions).
<input type="checkbox"/>	Where applicable, enrich your application with user data. Using the <a href="#">Microsoft Graph API</a> is an easy way to do this. The <a href="#">Graph Explorer</a> tool that can help you get started.
<input type="checkbox"/>	Register the full set of permissions that your app requires so admins can grant consent easily to their tenant. Use <a href="#">incremental consent</a> at run time to help users understand why your app is requesting permissions that may concern or confuse users when requested on first start.
<input type="checkbox"/>	Implement a <a href="#">clean single sign-out experience</a> . It's a privacy and a security requirement, and makes for a good user experience.

## Testing

<input type="checkbox"/>	Test for <a href="#">Conditional Access policies</a> that may affect your users' ability to use your application.
<input type="checkbox"/>	Test your application with all possible accounts that you plan to support (for example, work or school accounts, personal Microsoft accounts, child accounts, and sovereign accounts).

## Additional resources

Explore in-depth information about v2.0:

- [Microsoft identity platform \(v2.0 overview\)](#)

- [Microsoft identity platform protocols reference](#)
- [Access tokens reference](#)
- [ID tokens reference](#)
- [Authentication libraries reference](#)
- [Permissions and consent in the Microsoft identity platform](#)
- [Microsoft Graph API](#)

# Supported account types

11/12/2019 • 2 minutes to read • [Edit Online](#)

This article explains what accounts types (sometimes named audiences) are supported in applications

## Supported accounts types in Microsoft Identity platform applications

In the Microsoft Azure public Cloud, most types of apps can sign in users with any audience:

- If you're writing a Line of Business (LOB) application, you can sign in users in your own organization. Such an application is sometimes named **single tenant**.
- If you're an ISV, you can write an application which signs-in users:
  - In any organization. Such an application is named a **multi-tenant** web application. You'll sometimes read that it signs-in users with their work or school accounts.
  - With their work or school or personal Microsoft account.
  - With only personal Microsoft account.

### NOTE

Currently the Microsoft identity platform supports personal Microsoft accounts only by registering an app for **work or school or Microsoft personal accounts**, and then, restrict sign-in in the code for the application by specifying an Azure AD authority, when building the application, such as

`https://login.microsoftonline.com/consumers`

- If you're writing a business to consumers application, you can also sign in users with their social identities, using Azure AD B2C.

## Certain authentication flows don't support all the account types

Some account types can't be used with certain authentication flows. For instance, in desktop, UWP applications, or daemon applications:

- Daemon applications can only be used with Azure Active Directory organizations. It doesn't make sense to attempt to use daemon applications to manipulate Microsoft personal accounts (the admin consent will never be granted).
- You can only use the Integrated Windows Authentication flow with work or school accounts (in your organization or any organization). Indeed, Integrated Windows Authentication works with domain accounts, and requires the machines to be domain joined or Azure AD joined. This flow doesn't make sense for personal Microsoft Accounts.
- The [Resource Owner Password Grant](#) (Username/Password), can't be used with personal Microsoft accounts. Indeed, personal Microsoft accounts require that the user consents to accessing personal resources at each sign-in session. That's why, this behavior isn't compatible with non-interactive flows.
- Device code flow doesn't yet work with personal Microsoft accounts.

## Supported account types in national clouds

Apps can also sign in users in [national clouds](#). However, Microsoft personal accounts aren't supported in these clouds (by definition of these clouds). That's why the supported account types are reduced, for these clouds, to

your organization (single tenant) or any organizations (multi-tenant applications).

## Next steps

- Learn more about [Tenancy in Azure Active Directory](#)
- Learn more about [National Clouds](#)

# Scenario: Single-page application

8/7/2019 • 2 minutes to read • [Edit Online](#)

Learn all you need to build a single-page application (SPA).

## Prerequisites

Before reading this article, you should be familiar with the following concepts or read the following articles:

- [Microsoft identity platform overview](#)
- [Authentication basics](#)
- [Audiences](#)
- [Application and service principals](#)
- [Permissions and consent](#)
- [ID tokens and access tokens](#)

## Getting started

You can create your first application by following the JavaScript SPA quickstart:

[Quickstart: Single-page application](#)

## Overview

Many modern web applications are built as client-side single-page applications written using JavaScript or a SPA framework such as Angular, Vue.js, and React.js. These applications run in a web browser and have different authentication characteristics than traditional server-side web applications. The Microsoft identity platform enables single-page applications to sign in users and get tokens to access backend services or web APIs using the [OAuth 2.0 implicit flow](#). The implicit flow allows the application to get ID tokens to represent the authenticated user and also access tokens needed to call protected APIs.



This authentication flow does not include application scenarios using cross-platform JavaScript frameworks such as Electron, React-Native and so on. since they require further capabilities for interaction with the native platforms.

## Specifics

The following aspects are required to enable this scenario for your application:

- Application registration with Azure AD involves enabling the implicit flow and setting a redirect URI to which tokens are returned.
- Application configuration with the registered application properties such as Application ID.
- Using MSAL library to do the auth flow to sign in and acquire tokens.

## Next steps

[App registration](#)

# Single-page application - app registration

5/6/2019 • 2 minutes to read • [Edit Online](#)

This page explains the app registration specifics for a single-page application (SPA).

Follow the steps to [register a new application with Microsoft identity platform](#), and select the supported accounts for your application. The SPA scenario can support authentication with accounts in your organization or any organization and personal Microsoft accounts.

Next, learn the specific aspects of application registration that apply to single-page applications.

## Register a redirect URI

The implicit flow sends the tokens in a redirect to the single-page application running in a web browser. Therefore, it's an important requirement to register a redirect URI where your application can receive the tokens. Please ensure that the redirect URI matches exactly with the URI for your application.

In the [Azure portal](#), navigate to your registered application, on the **Authentication** page of the application, select the **Web** platform and enter the value of the redirect URI for your application in the **Redirect URI** field.

## Enable the implicit flow

On the same **Authentication** page, under **Advanced settings**, you must also enable the **Implicit grant**. If your application is only performing sign in of users and getting ID tokens, it's sufficient to enable **ID tokens** checkbox.

If your application also needs to get access tokens to call APIs, make sure to enable the **Access tokens** checkbox as well. For more information, see [ID tokens](#) and [Access tokens](#).

## API permissions

Single-page applications can call APIs on behalf of the signed-in user. They need to request delegated permissions. For details, see [Add permissions to access web APIs](#)

## Next steps

[App's code configuration](#)

# Single-page application - code configuration

8/21/2019 • 2 minutes to read • [Edit Online](#)

Learn how to configure the code for your single-page application (SPA).

## MSAL libraries supporting implicit flow

Microsoft identity platform provides MSAL.js library to support the implicit flow using the industry recommended secure practices.

The libraries supporting implicit flow are:

MSAL LIBRARY	DESCRIPTION
 MSAL.js	Plain JavaScript library for use in any client side web app built using JavaScript or SPA frameworks such as Angular, Vue.js, React.js, etc.
 MSAL Angular	Wrapper of the core MSAL.js library to simplify use in single page apps built with the Angular framework. This library is in preview and has <a href="#">known issues</a> with certain Angular versions and browsers.

## Application code configuration

In MSAL library, the application registration information is passed as configuration during the library initialization.

### JavaScript

```
// Configuration object constructed.
const config = {
 auth: {
 clientId: 'your_app_id',
 redirectUri: "your_app_redirect_uri" //defaults to application start page
 }
}

// create UserAgentApplication instance
const userAgentApplication = new UserAgentApplication(config);
```

For more details on the configurable options available, see [Initializing application with MSAL.js](#).

### Angular

```
//In app.module.ts
import { MsalModule } from '@azure/msal-angular';

@NgModule({
 imports: [MsalModule.forRoot({
 clientID: 'your_app_id'
 })]
})
export class AppModule { }
```

## Next steps

[Sign-in and sign-out](#)

# Single-page application - sign in

8/15/2019 • 3 minutes to read • [Edit Online](#)

Learn how to add sign in to the code for your single-page application.

Before you can get tokens to access APIs in your application, you will need an authenticated user context. You can sign in users to your application in MSAL.js in two ways:

- [Sign in with a pop-up window](#) using `loginPopup` method
- [Sign in with redirect](#) using `loginRedirect` method

You can also optionally pass the scopes of the APIs for which you need the user to consent at the time of sign in.

## NOTE

If your application already has access to an authenticated user context or id token, you can skip the login step and directly acquire tokens. For more details, see [sso without msal.js login](#).

## Choosing between a pop-up or redirect experience

You cannot use a combination of both the pop-up and redirect methods in your application. The choice between a pop-up or redirect experience depends on your application flow.

- If you don't want the user to navigate away from your main application page during authentication, it's recommended to use the pop-up methods. Since the authentication redirect happens in a pop-up window, the state of the main application is preserved.
- There are certain cases where you may need to use the redirect methods. If users of your application have browser constraints or policies where pop-ups windows are disabled, you can use the redirect methods. Use the redirect methods with Internet Explorer browser since there are certain [known issues with Internet Explorer](#) when handling pop-up windows.

## Sign in with a pop-up window

### JavaScript

```
const loginRequest = {
 scopes: ["https://graph.microsoft.com/User.ReadWrite"]
}

userAgentApplication.loginPopup(loginRequest).then(function (loginResponse) {
 //login success
 let idToken = loginResponse.idToken;
}).catch(function (error) {
 //login failure
 console.log(error);
});
```

### Angular

The MSAL Angular wrapper allows you to secure specific routes in your application by just adding the `MsalGuard` to the route definition. This guard will invoke the method to sign in when that route is accessed.

```
// In app.routes.ts
{ path: 'product', component: ProductComponent, canActivate : [MsalGuard],
 children: [
 { path: 'detail/:id', component: ProductDetailComponent }
]
},
{ path: 'myProfile' ,component: MsGraphComponent, canActivate : [MsalGuard] },
```

For a pop-up window experience, enable the `popUp` config option. You can also pass the scopes that require consent as follows:

```
//In app.module.ts
@NgModule({
 imports: [MsalModule.forRoot({
 clientID: 'your_app_id',
 popUp: true,
 consentScopes: ["https://graph.microsoft.com/User.ReadWrite"]
 })]
})
```

## Sign in with redirect

### JavaScript

The redirect methods do not return a promise due to the navigation away from the main app. To process and access the returned tokens, you will need to register success and error callbacks before calling the redirect methods.

```
function authCallback(error, response) {
 //handle redirect response
}

userAgentApplication.handleRedirectCallback(authCallback);

const loginRequest = {
 scopes: ["https://graph.microsoft.com/User.ReadWrite"]
}

userAgentApplication.loginRedirect(loginRequest);
```

### Angular

The code here is the same as described above under the sign in with a pop-up window section. The default flow is redirect.

#### NOTE

The ID token doesn't contain the consented scopes and only represents the authenticated user. The consented scopes are returned in the access token which you will acquire in the next step.

## Sign out

The MSAL library provides a `logout` method that will clear the cache in the browser storage and sends a sign out request to Azure AD. After sign out, it redirects back to the application start page by default.

You can configure the URI to which it should redirect after sign out by setting the `postLogoutRedirectUri`. This URI should also be registered as the Logout URI in your application registration.

## JavaScript

```
const config = {

 auth: {
 clientID: 'your_app_id',
 redirectUri: "your_app_redirect_uri", //defaults to application start page
 postLogoutRedirectUri: "your_app_logout_redirect_uri"
 }
}

const userAgentApplication = new UserAgentApplication(config);
userAgentApplication.logout();
```

## Angular

```
//In app.module.ts
@NgModule({
 imports: [MsalModule.forRoot({
 clientID: 'your_app_id',
 postLogoutRedirectUri: "your_app_logout_redirect_uri"
 })]
})

// In app.component.ts
this.authService.logout();
```

## Next steps

[Acquiring a token for the app](#)

# Single-page application - acquire a token to call an API

8/29/2019 • 3 minutes to read • [Edit Online](#)

The pattern for acquiring tokens for APIs with MSAL.js is to first attempt a silent token request using the `acquireTokenSilent` method. When this method is called, the library first checks the cache in the browser storage to see if a valid token exists and returns it. When there is no valid token in the cache, it sends a silent token request to Azure Active Directory (Azure AD) from a hidden iframe. This method also allows the library to renew tokens. For more information about single sign-on session and token lifetime values in Azure AD, see [token lifetimes](#).

The silent token requests to Azure AD may fail for some reasons such as an expired Azure AD session or a password change. In that case, you can invoke one of the interactive methods(which will prompt the user) to acquire tokens.

- [Acquire token with a pop-up window](#) using `acquireTokenPopup`
- [Acquire token with redirect](#) using `acquireTokenRedirect`

## Choosing between a pop-up or redirect experience

You cannot use a combination of both the pop-up and redirect methods in your application. The choice between a pop-up or redirect experience depends on your application flow.

- If you don't want the user to navigate away from your main application page during authentication, it's recommended to use the pop-up methods. Since the authentication redirect happens in a pop-up window, the state of the main application is preserved.
- There are certain cases where you may need to use the redirect methods. If users of your application have browser constraints or policies where pop-ups windows are disabled, you can use the redirect methods. It's also recommended to use the redirect methods with Internet Explorer browser since there are certain [known issues with Internet Explorer](#) when handling pop-up windows.

You can set the API scopes that you want the access token to include when building the access token request. Note, that all requested scopes may not be granted in the access token and depends on the user's consent.

## Acquire token with a pop-up window

### JavaScript

The above pattern using the methods for a pop-up experience:

```

const accessTokenRequest = {
 scopes: ["user.read"]
}

userAgentApplication.acquireTokenSilent(accessTokenRequest).then(function(accessTokenResponse) {
 // Acquire token silent success
 // call API with token
 let accessToken = accessTokenResponse.accessToken;
}).catch(function (error) {
 //Acquire token silent failure, send an interactive request.
 if (error.errorMessage.indexOf("interaction_required") !== -1) {
 userAgentApplication.acquireTokenPopup(accessTokenRequest).then(function(accessTokenResponse) {
 // Acquire token interactive success
 }).catch(function(error) {
 // Acquire token interactive failure
 console.log(error);
 });
 }
 console.log(error);
});

```

## Angular

The MSAL Angular wrapper provides the convenience of adding the HTTP interceptor, which will automatically acquire access tokens silently and attach them to the HTTP requests to APIs.

You can specify the scopes for APIs in the `protectedResourceMap` config option, which the `MsalInterceptor` will request when automatically acquiring tokens.

```

//In app.module.ts
@NgModule({
 imports: [MsalModule.forRoot({
 clientID: 'your_app_id',
 protectedResourceMap: {"https://graph.microsoft.com/v1.0/me": ["user.read", "mail.send"]}
 })
]

providers: [ProductService, {
 provide: HTTP_INTERCEPTORS,
 useClass: MsalInterceptor,
 multi: true
}
],

```

For success and failure of the silent token acquisition, MSAL Angular provides callbacks you can subscribe to. It's also important to remember to unsubscribe.

```

// In app.component.ts
ngOnInit() {
 this.subscription= this.broadcastService.subscribe("msal:acquireTokenFailure", (payload) => {
 });
}

ngOnDestroy() {
 this.broadcastService.getMSALSubject().next(1);
 if(this.subscription) {
 this.subscription.unsubscribe();
 }
}

```

Alternatively, you can also explicitly acquire tokens using the acquire token methods as described in the core `MSAL.js` library.

# Acquire token with redirect

## JavaScript

The pattern is as described above but shown with a redirect method to acquire token interactively. You will need to register the redirect callback as mentioned above.

```
function authCallback(error, response) {
 //handle redirect response
}

userAgentApplication.handleRedirectCallback(authCallback);

const accessTokenRequest: AuthenticationParameters = {
 scopes: ["user.read"]
}

userAgentApplication.acquireTokenSilent(accessTokenRequest).then(function(accessTokenResponse) {
 // Acquire token silent success
 // call API with token
 let accessToken = accessTokenResponse.accessToken;
}).catch(function (error) {
 //Acquire token silent failure, send an interactive request.
 console.log(error);
 if (error.errorMessage.indexOf("interaction_required") !== -1) {
 userAgentApplication.acquireTokenRedirect(accessTokenRequest);
 }
});
```

## Request for optional claims

You can request optional claims in your app to specify which additional claims to include in tokens for your application. In order to request optional claims in the id\_token, you can send a stringified claims object to the claimsRequest field of the AuthenticationParameters.ts class.

You can use optional claims for following purpose:

- To include additional claims in tokens for your application.
- Change the behavior of certain claims that Azure AD returns in tokens.
- Add and access custom claims for your application.

## JavaScript

```
"optionalClaims": [
 {
 "idToken": [
 {
 "name": "auth_time",
 "essential": true
 }
],
 var request = {
 scopes: ["user.read"],
 claimsRequest: JSON.stringify(claims)
 };
 myMSALObj.acquireTokenPopup(request);
 }
];
```

To learn more about optional claims, checkout [optional claims](#)

## Angular

This is the same as described above.

## Next steps

[Calling a Web API](#)

# Single-page application - call a web API

5/10/2019 • 2 minutes to read • [Edit Online](#)

We recommend that you call the `acquireTokenSilent` method to acquire or renew an access token before calling a web API. Once you have a token, you can call a protected web API.

## Call a web API

### JavaScript

Use the acquired access token as a bearer in an HTTP request to call any web API such as Microsoft Graph API. For example:

```
var headers = new Headers();
var bearer = "Bearer " + access_token;
headers.append("Authorization", bearer);
var options = {
 method: "GET",
 headers: headers
};
var graphEndpoint = "https://graph.microsoft.com/v1.0/me";

fetch(graphEndpoint, options)
 .then(function (response) {
 //do something with response
 })
```

### Angular

As mentioned in the [acquiring tokens section](#), the MSAL Angular wrapper leverages the HTTP interceptor to automatically acquire access tokens silently and attach them to the HTTP requests to APIs.

## Next steps

[Move to production](#)

# Single-page application - move to production

5/6/2019 • 2 minutes to read • [Edit Online](#)

Now that you know how to acquire a token to call Web APIs, learn how to move to production.

## Improve your app

Follow the steps needed to make your app production ready.

- [Enable logging](#) in your application.

## Test your integration

- Test your integration by following the [Microsoft identity platform integration checklist](#).

## Next steps

Here are a few other samples/tutorials:

- For a deep dive of the quickstart sample which explains the code for how to sign in users and get an access token to call the MS Graph API using MSAL.js  
[JavaScript SPA tutorial](#)

- Sample demonstrating how to get tokens for your own backend web API using MSAL.js  
[SPA with an ASP.NET backend](#)

- Sample to show how to use MSAL.js to sign in users in an app registered with Azure AD B2C  
[SPA with Azure AD B2C](#)

# Scenario: Web app that signs in users

10/18/2019 • 2 minutes to read • [Edit Online](#)

Learn all you need to build a web app that signs-in users with the Microsoft identity platform.

## Prerequisites

Before reading this article, you should be familiar with the following concepts or read the following articles:

- [Microsoft identity platform overview](#)
- [Authentication basics](#)
- [Audiences](#)
- [Application and service principals](#)
- [Permissions and consent](#)
- [ID tokens and access tokens](#)

## Getting started

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

If you want to create your first portable (ASP.NET Core) web apps that sign in users, follow this quickstart:

[Quickstart: ASP.NET Core web app that signs-in users](#)

## Overview

You add authentication to your web app, so that it can sign in users. Adding authentication enables your web app to access limited profile information, and, for instance customize the experience you offer to its users. Web apps authenticate a user in a web browser. In this scenario, the web application directs the user's browser to sign them in to Azure AD. Azure AD returns a sign-in response through the user's browser, which contains claims about the user in a security token. Signing-in users leverage the [Open ID Connect](#) standard protocol itself simplified by the use of middleware [libraries](#).



Web App

As a second phase you can also enable your application to call Web APIs on behalf of the signed-in user. This next phase is a different scenario, which you'll find in [Web App calls Web APIs](#)

### NOTE

Adding sign-in to a web app is about protecting the web app, and validating a user token, which is what **middleware** libraries do. In the case of .NET, this scenario does not require yet the Microsoft Authentication Libraries (MSAL), which are about acquiring a token to call protected APIs. The authentication libraries will only be introduced in the follow-up scenario when the web app needs to call web APIs.

## Specifics

- During the application registration, you'll need to provide one, or several (if you deploy your app to several locations) Reply URLs. In some cases (ASP.NET/ASP.NET Core), you'll need to enable the ID token. Finally you'll want to set up a sign-out URI so that your application reacts to users signing-out.
- In the code for your application, you'll need to provide the authority to which you web app delegates sign-in. You might want to customize token validation (in particular in ISV scenarios).
- Web applications support any account types. For more info, see [Supported account types](#).

## Next steps

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

[App registration](#)

# Web app that signs in users - app registration

10/30/2019 • 4 minutes to read • [Edit Online](#)

This page explains the app registration specifics for a web app that signs-in users.

To register your application, you can use:

- The [web app quickstarts](#) - In addition to being a great first experience with creating an application, quickstarts in the Azure portal contain a button named **Make this change for me**. You can use this button to set the properties you need, even for an existing app. You'll need to adapt the values of these properties to your own case. In particular, the web API URL for your app is probably going to be different from the proposed default, which will also impact the sign-out URI.
- The Azure portal to [register your application manually](#)
- PowerShell and command-line tools

## Register an app using the QuickStarts

If you navigate to this link, you can create bootstrap the creation of your web application:

- [ASP.NET Core](#)
- [ASP.NET](#)

## Register an app using Azure portal

### NOTE

the portal to use is different depending on if your application runs in the Microsoft Azure public cloud or in a national or sovereign cloud. For more information, see [National Clouds](#)

1. Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account. Alternatively, sign in to the national cloud Azure portal of choice.
2. If your account gives you access to more than one tenant, select your account in the top-right corner, and set your portal session to the desired Azure AD tenant.
3. In the left-hand navigation pane, select the **Azure Active Directory** service, and then select **App registrations** > **New registration**.
  - [ASP.NET Core](#)
  - [ASP.NET](#)
  - [Java](#)
  - [Python](#)
4. When the **Register an application** page appears, enter your application's registration information:
  - a. choose the supported account types for your application (See [Supported Account types](#))
  - b. In the **Name** section, enter a meaningful application name that will be displayed to users of the app, for example `AspNetCore-WebApp`.
  - c. In **Redirect URI**, add the type of application and the URI destination that will accept returned token responses after successfully authenticating. For example, `https://localhost:44321/`. Select **Register**.
5. Select the **Authentication** menu, and then add the following information:
  - a. In **Reply URL**, add `https://localhost:44321/signin-oidc` of type "Web".

- b. In the **Advanced settings** section, set **Logout URL** to `https://localhost:44321/signout-oidc`.
- c. Under **Implicit grant**, check **ID tokens**.
- d. Select **Save**.

## Register an app using PowerShell

### NOTE

Currently Azure AD PowerShell only creates applications with the following supported account types:

- MyOrg (Accounts in this organizational directory only)
- AnyOrg (Accounts in any organizational directory).

If you want to create an application that signs-in users with their personal Microsoft Accounts (e.g. Skype, Xbox, Outlook.com), you can first create a multi-tenant application (Supported account types = Accounts in any organizational directory), and then change the `signInAudience` property in the application manifest from the Azure portal. This is explained in details in the step [1.3](#) of the ASP.NET Core tutorial (and can be generalized to web apps in any language).

## Next steps

[App's code configuration](#)

# Web app that signs-in users - code configuration

10/30/2019 • 9 minutes to read • [Edit Online](#)

Learn how to configure the code for your Web app that signs-in users.

## Libraries used to protect Web Apps

The libraries used to protect a Web App (and a Web API) are:

PLATFORM	LIBRARY	DESCRIPTION
	<a href="#">Identity model extensions for .NET</a>	Used directly by ASP.NET and ASP.NET Core, Microsoft Identity Extensions for .NET proposes a set of DLLs running both on .NET Framework and .NET Core. From an ASP.NET/ASP.NET Core Web app, you can control token validation using the <b>TokenValidationParameters</b> class (in particular in some ISV scenarios)
	<a href="#">MSAL Java</a>	MSAL for Java - currently in public preview
	<a href="#">MSAL Python</a>	MSAL for Python - currently in public preview

Select the tab corresponding to the platform you're interested in:

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

Code snippets in this article and the following are extracted from the [ASP.NET Core Web app incremental tutorial, chapter 1](#).

You might want to refer to this tutorial for full implementation details.

## Configuration files

Web applications that sign in users with the Microsoft identity platform are usually configured through configuration files. The settings that you need to fill in are:

- the cloud `instance` if you want your app to run (for instance in national clouds)
- the audience in `tenantId`
- the `clientId` for your application, as copied from the Azure portal.

Sometimes, applications can be parametrized by the `authority`, which is the concatenation of the `instance` and the `tenantId`

- [ASP.NET Core](#)

- [ASP.NET](#)
- [Java](#)
- [Python](#)

In ASP.NET Core, these settings are located in the [appsettings.json](#) file, in the "AzureAD" section.

```
{
 "AzureAd": {
 // Azure Cloud instance among:
 // "https://login.microsoftonline.com/" for Azure Public cloud.
 // "https://login.microsoftonline.us/" for Azure US government.
 // "https://login.microsoftonline.de/" for Azure AD Germany
 // "https://login.chinacloudapi.cn/" for Azure AD China operated by 21Vianet
 "Instance": "https://login.microsoftonline.com/",

 // Azure AD Audience among:
 // - the tenant Id as a GUID obtained from the azure portal to sign-in users in your organization
 // - "organizations" to sign-in users in any work or school accounts
 // - "common" to sign-in users with any work and school account or Microsoft personal account
 // - "consumers" to sign-in users with Microsoft personal account only
 "TenantId": "[Enter the tenantId here]",

 // Client Id (Application ID) obtained from the Azure portal
 "ClientId": "[Enter the Client Id]",
 "CallbackPath": "/signin-oidc",
 "SignedOutCallbackPath": "/signout-callback-oidc"
 }
}
```

In ASP.NET Core, there's another file [properties\launchSettings.json](#) that contains the URL (`applicationUrl`) and the SSL Port (`sslPort`) for your application and various profiles.

```
{
 "iisSettings": {
 "windowsAuthentication": false,
 "anonymousAuthentication": true,
 "iisExpress": {
 "applicationUrl": "http://localhost:3110/",
 "sslPort": 44321
 }
 },
 "profiles": {
 "IIS Express": {
 "commandName": "IISExpress",
 "launchBrowser": true,
 "environmentVariables": {
 "ASPNETCORE_ENVIRONMENT": "Development"
 }
 },
 "webApp": {
 "commandName": "Project",
 "launchBrowser": true,
 "environmentVariables": {
 "ASPNETCORE_ENVIRONMENT": "Development"
 },
 "applicationUrl": "http://localhost:3110/"
 }
 }
}
```

In the Azure portal, the reply URIs that you need to register in the **Authentication** page for your application needs to match these URLs; that is, for the two configuration files above, they would be

`https://localhost:44321/signin-oidc` as the applicationUrl is `http://localhost:3110` but the `sslPort` is specified (44321), and the `CallbackPath` is `/signin-oidc` as defined in the `appsettings.json`.

In the same way, the sign-out URI would be set to `https://localhost:44321/signout-callback-oidc`.

## Initialization code

The initialization code is different depending on the platform. For ASP.NET Core and ASP.NET, signing in users is delegated to the OpenIDConnect middleware. Today the ASP.NET / ASP.NET Core template generate web applications for the Azure AD v1.0 endpoint. Therefore, a bit of configuration is required to adapt them to the Microsoft identity platform (v2.0) endpoint. In the case of Java, it's handled by Spring with the cooperation of the application.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

In ASP.NET Core Web Apps (and Web APIs), the application is protected because you have a `[Authorize]` attribute on the controllers or the controller actions. This attribute checks that the user is authenticated. The code doing the application initialization is located in the `Startup.cs` file, and, to add authentication with the Microsoft identity platform (formerly Azure AD v2.0), you'll need to add the following code. The comments in the code should be self-explanatory.

### NOTE

If you start your project with default ASP.NET core web project within Visual studio or using `dotnet new mvc` the method `AddAzureAD` is available by default because the related packages are automatically loaded. However if you build a project from scratch and are trying to use the below code we suggest you to add the NuGet Package "**Microsoft.AspNetCore.Authentication.AzureAD.UI**" to your project to make the `AddAzureAD` method available.

The following code is available from [Startup.cs#L33-L34](#)

```
public class Startup
{
 ...
 // This method gets called by the runtime. Use this method to add services to the container.
 public void ConfigureServices(IServiceCollection services)
 {
 ...
 // Sign-in users with the Microsoft identity platform
 services.AddMicrosoftIdentityPlatformAuthentication(Configuration);

 services.AddMvc(options =>
 {
 var policy = new AuthorizationPolicyBuilder()
 .RequireAuthenticatedUser()
 .Build();
 options.Filters.Add(new AuthorizeFilter(policy));
 })
 .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
 }
}
```

The `AddMicrosoftIdentityPlatformAuthentication` is an extension method defined in [Microsoft.Identity.Web/WebAppServiceCollectionExtensions.cs#L23](#). It:

- adds the authentication service
- configure options to read the config file
- configures the OpenID connect options so that the used authority is the Microsoft identity platform (formerly Azure AD v2.0) endpoint
- the issuer of the token is validated
- the claims corresponding to name is mapped from the "preferred\_username" claim in the ID Token

In addition to the configuration, you can specify, when calling `AddMicrosoftIdentityPlatformAuthentication` :

- the name of the configuration section (by default `AzureAD`)
- if you want to trace the OpenIdConnect middleware events, which can help you troubleshooting your Web application if authentication doesn't work: setting `subscribeToOpenIdConnectMiddlewareDiagnosticsEvents` to `true` will show you how information gets elaborated by the set of ASP.NET Core middleware as it progresses from the HTTP response to the identity of the user in the `HttpContext.User`.

```

/// <summary>
/// Add authentication with Microsoft identity platform.
/// This method expects the configuration file will have a section named "AzureAd" with the necessary settings
/// to initialize authentication options.
/// </summary>
/// <param name="services">Service collection to which to add this authentication scheme</param>
/// <param name="configuration">The Configuration object</param>
/// <param name="subscribeToOpenIdConnectMiddlewareDiagnosticsEvents">
/// Set to true if you want to debug, or just understand the OpenIdConnect events.
/// </param>
/// <returns></returns>
public static IServiceCollection AddMicrosoftIdentityPlatformAuthentication(
 this IServiceCollection services,
 IConfiguration configuration,
 string configSectionName = "AzureAd",
 bool subscribeToOpenIdConnectMiddlewareDiagnosticsEvents = false)
{
 services.AddAuthentication(AzureADDefaults.AuthenticationScheme)
 .AddAzureAD(options => configuration.Bind(configSectionName, options));
 services.Configure<AzureADOptions>(options => configuration.Bind(configSectionName, options));

 services.Configure<OpenIdConnectOptions>(AzureADDefaults.OpenIdScheme, options =>
 {
 // Per the code below, this application signs in users in any Work and School
 // accounts and any Microsoft Personal Accounts.
 // If you want to direct Azure AD to restrict the users that can sign-in, change
 // the tenant value of the appsettings.json file in the following way:
 // - only Work and School accounts => 'organizations'
 // - only Microsoft Personal accounts => 'consumers'
 // - Work and School and Personal accounts => 'common'
 // If you want to restrict the users that can sign-in to only one tenant
 // set the tenant value in the appsettings.json file to the tenant ID
 // or domain of this organization
 options.Authority = options.Authority + "/v2.0/";

 // If you want to restrict the users that can sign-in to several organizations
 // Set the tenant value in the appsettings.json file to 'organizations', and add the
 // issuers you want to accept to options.TokenValidationParameters.ValidIssuers collection
 options.TokenValidationParameters.IssuerValidator =
 AadIssuerValidator.GetIssuerValidator(options.Authority).Validate;

 // Set the nameClaimType to be preferred_username.
 // This change is needed because certain token claims from Azure AD V1 endpoint
 // (on which the original .NET core template is based) are different than Microsoft identity platform
 // endpoint.
 // For more details see [ID Tokens](https://docs.microsoft.com/azure/active-directory/develop/id-tokens)
 // and [Access Tokens](https://docs.microsoft.com/azure/active-directory/develop/access-tokens)
 options.TokenValidationParameters.NameClaimType = "preferred_username";

 // ...

 if (subscribeToOpenIdConnectMiddlewareDiagnosticsEvents)
 {
 OpenIdConnectMiddlewareDiagnostics.Subscribe(options.Events);
 }
 });
 return services;
}
...

```

The `AadIssuerValidator` class enables that the issuer of the token is validated in many cases (v1.0 or v2.0 token, single-tenant, or multi-tenant application or application that signs in users with their personal Microsoft accounts, in the Azure public cloud, or national clouds). It's available from

[Microsoft.Identity.Web/Resource/AadIssuerValidator.cs](#)

## Next steps

In the next article, you'll learn how to trigger the sign-in and sign-out.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

[Sign in and sign out](#)

# Web app that signs in users - sign in and sign out

10/30/2019 • 7 minutes to read • [Edit Online](#)

Learn how to add sign-in to the code for your web app that signs-in users, and then, how to let them sign out

## Sign-in

Sign-in is brought by two parts:

- the sign-in button in the HTML page
- the sign-in action in the code behind in the controller

### Sign-in button

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

In ASP.NET Core, the sign-in button is exposed in `Views\Shared\_LoginPartial.cshtml` and only displayed when there's no authenticated account (that is when the user hasn't yet signed-in, or has signed-out).

```
@using Microsoft.Identity.Web
@if (User.Identity.IsAuthenticated)
{
 // Code omitted code for clarity
}
else
{
 <ul class="nav navbar-nav navbar-right">
 <a asp-area="AzureAD" asp-controller="Account" asp-action="SignIn">Sign in

}
```

### Login action of the controller

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

In ASP.NET, pressing the **Sign-in** button on the web app triggers the `SignIn` action on the `AccountController` controller. In previous versions of the ASP.NET core templates, the `Account` controller was embedded with the web app, but it's no longer the case as it's now part of the ASP.NET Core framework itself.

The code for the `AccountController` is available from the ASP.NET core repository from [AccountController.cs](#). The account control challenges the user by redirecting to the Microsoft identity platform endpoint. For details see the `SignIn` method provided as part of ASP.NET Core.

Once the user has signed-in to your app, you probably want to enable them to sign out.

## Sign out

Signing out from a web app is about more than removing the information about the signed-in account from the

web app's state. The web app must also redirect the user to the Microsoft identity platform `logout` endpoint to sign out. When your web app redirects the user to the `logout` endpoint, this endpoint clears the user's session from the browser. If your app didn't go to the `logout` endpoint, the user would reauthenticate to your app without entering their credentials again, because they would have a valid single sign-in session with the Microsoft identity platform endpoint.

To learn more, see the [Send a sign-out request](#) section in the [Microsoft identity platform and the OpenID Connect protocol](#) conceptual documentation.

## Application registration

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

During the application registration, you'll have registered a **post logout URI**. In our tutorial, you registered `https://localhost:44321/signout-oidc` in the **Logout URL** field of the **Advanced Settings** section in the **Authentication** page. For details see, [Register the webApp app](#)

## Sign-out button

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

In ASP.NET Core, the sign-out button is exposed in `Views\Shared\_LoginPartial.cshtml` and only displayed when there's an authenticated account (that is when the user has previously signed in).

```
@using Microsoft.Identity.Web
@if (User.Identity.IsAuthenticated)
{
 <ul class="nav navbar-nav navbar-right">
 <li class="navbar-text">Hello @User.GetDisplayName()!
 <a asp-area="AzureAD" asp-controller="Account" asp-action="SignOut">Sign out

}
else
{
 <ul class="nav navbar-nav navbar-right">
 <a asp-area="AzureAD" asp-controller="Account" asp-action="SignIn">Sign in

}
```

### Signout action of the controller

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

In ASP.NET, pressing the **Sign-out** button on the web app triggers the `SignOut` action on the `AccountController` controller. In previous versions of the ASP.NET core templates, the `Account` controller was embedded with the web app, but it's no longer the case as it's now part of the ASP.NET Core framework itself.

The code for the `AccountController` is available from the ASP.NET core repository at from [AccountController.cs](#).  
The account control:

- Sets an OpenID redirect URI to `/Account/SignedOut` so that the controller is called back when Azure AD has completed the sign-out
- Calls `Signout()`, which lets the OpenIdConnect middleware contact the Microsoft identity platform `logout` endpoint which:
  - Clears the session cookie from the browser, and
  - finally calls back the **logout URL**, which, by default, displays the signed out view page `SignedOut.html` also provided as part of ASP.NET Core.

### Intercepting the call to the `logout` endpoint

The post logout URI enables applications to participate to the global sign-out.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

The ASP.NET Core OpenIdConnect middleware enables your app to intercept the call to the Microsoft identity platform `logout` endpoint by providing an OpenIdConnect event named `OnRedirectToIdentityProviderForSignOut`.

See [Microsoft.Identity.Web/WebAppServiceCollectionExtensions.cs#L151-L156](#) for an example of how to subscribe to this event (to clear the token cache)

```
// Handling the global sign-out
options.Events.OnRedirectToIdentityProviderForSignOut = async context =>
{
 // Forget about the signed-in user
};
```

## Protocol

If you want to learn more about sign-out, read the protocol documentation, that is available from [Open ID Connect](#).

## Next steps

[Move to production](#)

# Web app that signs in users - move to production

9/18/2019 • 2 minutes to read • [Edit Online](#)

Now that you know how to acquire a token to call web APIs, learn how to move it to production.

Make your application great:

- Enable [logging](#).
- Enable telemetry.
- Enable [proxies](#) and customize HTTP clients.

Test your integration:

- Use the [Microsoft identity platform integration checklist](#).

## Next steps

### **Calling web APIs scenario**

Once your web app signs-in users, it can call web APIs on behalf of the signed-in users. Calling web APIs from the web app is the object of the following scenario:

#### [Web app that calls web APIs](#)

### **Deep dive - ASP.NET Core web app tutorial**

Learn about other ways of sign-in users with the ASP.NET Core tutorial: [ms-identity-aspnetcore-webapp-tutorial](#).

This sample is a progressive tutorial with production ready code for a web app including how to add sign in with accounts in:

- your organization,
- multiple organizations,
- work or school accounts or personal Microsoft account,
- with [Azure AD B2C](#),
- or in national clouds.

### **Sample code - Java web app**

Learn more about the Java web app from the sample on GitHub: [A Java Web application that signs in users with the Microsoft identity platform and calls Microsoft Graph](#)

# Scenario: Web app that calls web APIs

5/6/2019 • 2 minutes to read • [Edit Online](#)

Learn how to build a web app signing-in users on the Microsoft identity platform and that calls web APIs on behalf of the signed-in user.

## Prerequisites

Before reading this article, you should be familiar with the following concepts or read the following articles:

- [Microsoft identity platform overview](#)
- [Authentication basics](#)
- [Audiences](#)
- [Application and service principals](#)
- [Permissions and consent](#)
- [ID tokens and access tokens](#)

This scenario supposes that you've gone through the following scenario:

[Web app that signs-in users](#)

## Overview

You add authentication to your Web App, which can therefore sign in users and calls a web API on behalf of the signed-in user.



Web Apps that calls web APIs:

- are confidential client applications.
- that's why they've registered a secret (application password or certificate) with Azure AD. This secret is passed-in during the call to Azure AD to get a token

## Specifics

### NOTE

Adding sign-in to a Web App does not use the MSAL libraries as this is about protecting the Web App. Protecting libraries is achieved by libraries named Middleware. This was the object of the previous scenario [Sign-in users to a Web App](#)

When calling web APIs from a Web App, you will need to get access tokens for these web APIs. You can use MSAL libraries to acquire these tokens.

The end to end experience of developers for this scenario has, therefore, specific aspects as:

- During the [Application registration](#), you'll need to provide one, or several (if you deploy your app to several locations) Reply URIs, secrets, or certificates need to be shared with Azure AD.
- The [Application configuration](#) needs to provide client credentials as shared with Azure AD during the

application registration

## Next steps

[App registration](#)

# Web app that calls web APIs - app registration

5/6/2019 • 2 minutes to read • [Edit Online](#)

A Web app calling web APIs has the same registration as a Web App signing-in users. You'll therefore need to follow the instructions in [Web app that signs-in users - app registration](#)

However since the Web App now calls web APIs, it becomes a confidential client application. That's why there is a bit of extra registration required: it needs to share secrets (client credentials) with the Microsoft identity platform.

## Registration of secrets or certificates

Like for any confidential client application, you need to register a secret or certificate. You can register your application secrets either through the interactive experience in the [Azure portal](#), or using command-line tools (like PowerShell)

### Registering client secrets using the application registration portal

The management of client credentials happens in the **certificates & secrets** page for an application:

The screenshot shows the Azure portal interface. On the left, the navigation menu is visible with various service icons like App Services, Function Apps, and Azure Active Directory. The main content area is titled 'ContosoApp\_1 - Certificates & secrets'. A sidebar on the left lists 'Manage' options: Overview, Quickstart, Branding, Authentication, Certificates & secrets (which is selected and highlighted in blue), API permissions, Expose an API, Owners, Manifest, Support + Troubleshooting, Troubleshooting, and New support request. The main pane displays two sections: 'Certificates' and 'Client secrets'. The 'Certificates' section contains a note about using certificates for identifying the application and a button to 'Upload certificate'. The 'Client secrets' section contains a note about using a secret string for identity proofing and a button to 'New client secret'. Both sections have tables with columns like THUMBPRINT, START DATE, EXPIRES, DESCRIPTION, and VALUE.

- the application secret (also named client secret) is generated by Azure AD, during the registration of the confidential client application. This generation happens when you select **New client secret**. At that point, you must copy the secret string in the clipboard for use in your app, before selecting **Save**. This string won't be presented any longer.
- the certificate is uploaded in the application registration using the **Upload certificate** button. Azure AD only supports certificates that are directly registered on the application, and do not follow certificate chains.

For details, see [Quickstart: Configure a client application to access web APIs | Add credentials to your application](#)

### Registering client secrets using PowerShell

Alternatively, you can register your application with Azure AD using command-line tools. The [active-directory-](#)

[dotnetcore-daemon-v2](#) sample shows how to register an application secret or a certificate with an Azure AD application:

- For details on how to register an application secret, see [AppCreationScripts/Configure.ps1](#)
- For details on how to register a certificate with the application, see [AppCreationScripts-withCert/Configure.ps1](#)

## API permissions

Web applications call APIs on behalf of the signed-in user. They need to request delegated permissions. For details see [Add permissions to access web APIs](#)

## Next steps

[App's code configuration](#)

# Web app that calls web APIs - code configuration

10/30/2019 • 14 minutes to read • [Edit Online](#)

As seen in the [Web app signs-in users scenario](#), the web app uses the OAuth2.0 [authorization code flow](#) to sign in the user. This flow is in two parts:

1. Request an authorization code. This part delegates to the Microsoft identity platform a private dialog with the user. The user signs in and consents to the use of web APIs. When this private dialog ends successfully, the application receives an authorization code on its redirect URI.
2. Request an access token for the API by redeeming the authorization code.

[Web app signs-in users scenario](#) was only doing the first leg. Learn here, how to modify your web API that signs-in users, so that it now calls web APIs.

## Libraries supporting Web App scenarios

The libraries supporting the authorization code flow for web Apps are:

MSAL LIBRARY	DESCRIPTION
 MSAL.NET	Supported platforms are .NET Framework and .NET Core platforms (not UWP, Xamarin.iOS, and Xamarin.Android as those platforms are used to build public client applications)
 MSAL Python	Development in progress - in public preview
 MSAL Java	Development in progress - in public preview

Select the tab corresponding to the platform you're interested in:

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

Given that letting a user sign in is delegated to the Open ID connect (OIDC) middleware, you want to hook-up in the OIDC process. The way to do that is different depending on the framework you use. In the case of ASP.NET Core, you'll subscribe to middleware OIDC events. The principle is that:

- You'll let ASP.NET core request an authorization code, through the Open ID connect middleware. By doing this ASP.NET/ASP.NET core will let the user sign in and consent,
- You'll subscribe to the reception of the authorization code by the Web app. This subscription is done through a C# delegate.
- When the auth code is received, you'll use MSAL libraries to redeem the code and the resulting access tokens

and refresh tokens are stored in the token cache. From there, the cache can be used in other parts of the application, for instance in controllers, to acquire other tokens silently.

Code snippets in this article and the following are extracted from the [ASP.NET Core Web app incremental tutorial](#), chapter 2. You might want to refer to this tutorial for full implementation details.

#### NOTE

To understand fully the code snippets below, you need to be familiar with [ASP.NET Core fundamentals](#), and in particular [dependency injection](#) and [options](#)

## Code that redeems the authorization code

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

### Startup.cs

In ASP.NET Core, the principle is that in the `Startup.cs` file. You'll want to subscribe to the `OnAuthorizationCodeReceived` open ID connect event, and from this event, call MSAL.NET's method `AcquireTokenFromAuthorizationCode`, which has the effect of storing in the token cache, the access token for the requested `scopes`, and a refresh token that will be used to refresh the access token when it's close to expiry, or to get a token on behalf of the same user, but for a different resource.

In practice the [ASP.NET Core Web app tutorial](#) attempts to provide you with reusable code for your web apps.

Here is the `Startup.cs#L40-L42` code, featuring the call to the `AddMicrosoftIdentityPlatformAuthentication` method which adds authentication to the web app, and `AddMsal` which adds the capability of calling Web APIs. The call to `AddInMemoryTokenCaches` is about choosing a token cache implementation among the ones which are possible:

```
public class Startup
{
 // Code not show here

 public void ConfigureServices(IServiceCollection services)
 {
 // Token acquisition service based on MSAL.NET
 // and chosen token cache implementation
 services.AddMicrosoftIdentityPlatformAuthentication(Configuration)
 .AddMsal(Configuration, new string[] { Constants.ScopeUserRead })
 .AddInMemoryTokenCaches();
 }

 // Code not show here
}
```

`Constants.ScopeUserRead` is defined in `Constants.cs#L5`

```
public static class Constants
{
 public const string ScopeUserRead = "User.Read";
}
```

You've already studied the content of `AddMicrosoftIdentityPlatformAuthentication` in [Web app that signs-in users - code configuration](#)

## The AddMsal method

The code for `AddMsal` is located in [Microsoft.Identity.Web/WebAppServiceCollectionExtensions.cs#L108-L159](#).

```
/// <summary>
/// Extensions for IServiceCollection for startup initialization.
/// </summary>
public static class WebAppServiceCollectionExtensions
{
 // Code omitted here

 /// <summary>
 /// Add MSAL support to the Web App or Web API
 /// </summary>
 /// <param name="services">Service collection to which to add authentication</param>
 /// <param name="initialScopes">Initial scopes to request at sign-in</param>
 /// <returns></returns>
 public static IServiceCollection AddMsal(this IServiceCollection services, IConfiguration configuration,
 IEnumerable<string> initialScopes, string configSectionName = "AzureAd")
 {
 // Ensure that configuration options for MSAL.NET, HttpContext accessor and the Token acquisition
 // service
 // (encapsulating MSAL.NET) are available through dependency injection
 services.Configure<ConfidentialClientApplicationOptions>(options =>
 configuration.Bind(configSectionName, options));
 services.AddHttpContextAccessor();
 services.AddTokenAcquisition();

 services.Configure<OpenIdConnectOptions>(AzureADDefaults.OpenIdScheme, options =>
 {
 // Response type
 options.ResponseType = OpenIdConnectResponseType.CodeIdToken;

 // This scope is needed to get a refresh token when users sign-in with their Microsoft personal
 // accounts
 // (it's required by MSAL.NET and automatically provided when users sign-in with work or school
 // accounts)
 options.Scope.Add("offline_access");
 if (initialScopes != null)
 {
 foreach (string scope in initialScopes)
 {
 if (!options.Scope.Contains(scope))
 {
 options.Scope.Add(scope);
 }
 }
 }

 // Handling the auth redemption by MSAL.NET so that a token is available in the token cache
 // where it will be usable from Controllers later (through the TokenAcquisition service)
 var handler = options.Events.OnAuthorizationCodeReceived;
 options.Events.OnAuthorizationCodeReceived = async context =>
 {
 var tokenAcquisition = context.HttpContext.RequestServices.GetRequiredService<ITokenAcquisition>()
 ;
 await tokenAcquisition.AddAccountToCacheFromAuthorizationCodeAsync(context,
 options.Scope).ConfigureAwait(false);
 await handler(context).ConfigureAwait(false);
 };
 });
 return services;
 }
}
```

The `AddMsal` method ensures that:

- the ASP.NET Core Web app requests both an IDToken for the user, and an authentication code ( `options.ResponseType = OpenIdConnectResponseType.CodeIdToken` )
- the `offline_access` scope is added. It's needed so that the user consents to let the application get a refresh token.
- the app subscribes to the OIDC `OnAuthorizationCodeReceived` event, and redeems the call using MSAL.NET, which is here encapsulated into a reusable component implementing `ITokenAcquisition`.

### The `TokenAcquisition.AddAccountToCacheFromAuthorizationCodeAsync` method

The `TokenAcquisition.AddAccountToCacheFromAuthorizationCodeAsync` method is located in [Microsoft.Identity.Web/TokenAcquisition.cs#L101-L145](#). It ensures that:

- ASP.NET does not attempt to redeem the authentication code in parallel to MSAL.NET ( `context.HandleCodeRedemption();` )
- The claims in the IDToken are available for MSAL to compute a token cache key for the account of the user
- the MSAL.NET application is instantiated if needed
- the code is redeemed by the MSAL.NET application
- The new ID token is shared with ASP.NET Core (during the call to `context.HandleCodeRedemption(null, result.IdToken);`). The access token is not shared with ASP.NET Core. It remains in the MSAL.NET token cache associated with the user, where it's ready to be used in ASP.NET Core controllers.

```

public class TokenAcquisition : ITokenAcquisition
{
 string[] scopesRequestedByMsalNet = new string[]{"openid", "profile", "offline_access"};

 // Code omitted here for clarity

 public async Task AddAccountToCacheFromAuthorizationCodeAsync(AuthorizationCodeReceivedContext context,
 IEnumerable<string> scopes)
 {
 // Code omitted here for clarity

 try
 {
 // As AcquireTokenByAuthorizationCodeAsync is asynchronous we want to tell ASP.NET core that we are
 // handing the code
 // even if it's not done yet, so that it does not concurrently call the Token endpoint. (otherwise there
 // will be a
 // race condition ending-up in an error from Azure AD telling "code already redeemed")
 context.HandleCodeRedemption();

 // The cache will need the claims from the ID token.
 // If they are not yet in the HttpContext.User's claims, so adding them here.
 if (!context.HttpContext.User.Claims.Any())
 {
 (context.HttpContext.User.Identity as ClaimsIdentity).AddClaims(context.Principal.Claims);
 }

 var application = GetOrBuildConfidentialClientApplication();

 // Do not share the access token with ASP.NET Core otherwise ASP.NET will cache it and will not send the
 // OAuth 2.0 request in
 // case a further call to AcquireTokenByAuthorizationCodeAsync in the future is required for incremental
 // consent (getting a code requesting more scopes)
 // Share the ID Token though
 var result = await application
 .AcquireTokenByAuthorizationCode(scopes.Except(_scopesRequestedByMsalNet),
 context.ProtocolMessage.Code)
 .ExecuteAsync()
 .ConfigureAwait(false);

 context.HandleCodeRedemption(null, result.IdToken);
 }
 catch (MsalException ex)
 {
 Debug.WriteLine(ex.Message);
 throw;
 }
 }
}

```

### The TokenAcquisition.BuildConfidentialClientApplication method

In ASP.NET Core, building the confidential client application uses information that is in the `HttpContext`. Accessed through the `CurrentHttpContext` property, the `HttpContext`, associated with the request, knows about the URL for the Web App, and the signed-in user (in a `ClaimsPrincipal`). The `BuildConfidentialClientApplication` also uses the ASP.NET Core configuration, which has an "AzureAD" section, and which is bound both to:

- the `_applicationOptions` data structure of type `ConfidentialClientApplicationOptions`
  - the `azureAdOptions` instance of type `AzureAdOptions` defined in ASP.NET Core `Authentication.AzureAD.UI`.
- Finally the application needs to maintain token caches. You'll learn more about this in the next section.

The code for the `GetOrBuildConfidentialClientApplication()` method is in

[Microsoft.Identity.Web/TokenAcquisition.cs#L290-L333](#). It uses members that were injected by dependency injection (passed in the constructor of `TokenAcquisition` in [Microsoft.Identity.Web/TokenAcquisition.cs#L47-L59](#))

```

public class TokenAcquisition : ITokenAcquisition
{
 // Code omitted here for clarity

 // Members
 private IConfidentialClientApplication application;
 private HttpContext CurrentHttpContext => _httpContextAccessor.HttpContext;

 // The following members are set by dependency injection in the TokenAcquisition constructor
 private readonly AzureADOptions _azureAdOptions;
 private readonly ConfidentialClientApplicationOptions _applicationOptions;
 private readonly IMsalAppTokenCacheProvider _appTokenCacheProvider;
 private readonly IMsalUserTokenCacheProvider _userTokenCacheProvider;
 private readonly IHttpContextAccessor _httpContextAccessor;

 /// <summary>
 /// Creates an MSAL Confidential client application if needed
 /// </summary>
 private IConfidentialClientApplication GetOrBuildConfidentialClientApplication()
 {
 if (application == null)
 {
 application = BuildConfidentialClientApplication();
 }
 return application;
 }

 /// <summary>
 /// Creates an MSAL Confidential client application
 /// </summary>
 /// <param name="claimsPrincipal"></param>
 /// <returns></returns>
 private IConfidentialClientApplication BuildConfidentialClientApplication()
 {
 var request = CurrentHttpContext.Request;
 var azureAdOptions = _azureAdOptions;
 var applicationOptions = _applicationOptions;
 string currentUri = UriHelper.BuildAbsolute(
 request.Scheme,
 request.Host,
 request.PathBase,
 azureAdOptions.CallbackPath ?? string.Empty);

 string authority = $"{applicationOptions.Instance}{applicationOptions.TenantId}/";

 var app = ConfidentialClientApplicationBuilder
 .CreateWithApplicationOptions(applicationOptions)
 .WithRedirectUri(currentUri)
 .WithAuthority(authority)
 .Build();

 // Initialize token cache providers
 _appTokenCacheProvider?.InitializeAsync(app.AppTokenCache);
 _userTokenCacheProvider?.InitializeAsync(app.UserTokenCache);

 return app;
 }
}

```

## Summary

To sum-up, `AcquireTokenByAuthorizationCode` really redeems the authorization code requested by ASP.NET and gets the tokens that are added to MSAL.NET user token cache. From there, they're then, used, in the ASP.NET Core controllers.

Instead of a client secret, the confidential client application can also prove its identity using a client certificate, or a

client assertion. Using client assertions is an advanced scenario, detailed in [Client assertions](#)

## Token cache

### IMPORTANT

In web apps (or web APIs as a matter of fact), the token cache implementation is different from the Desktop applications token cache implementations (which are often [file based](#)). It's important, for security and performance reasons, to ensure is that for web Apps and web APIs, there should be one token cache per user (per account). You need to serialize the token cache for each account.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

The ASP.NET core tutorial uses dependency injection to let you decide the token cache implementation in the Startup.cs file for your application. Microsoft.Identity.Web comes with a number of pre-built token cache serializers described in [Token cache serialization](#). An interesting possibility is to choose ASP.NET Core [distributed memory caches](#):

```
// or use a distributed Token Cache by adding
services.AddMicrosoftIdentityPlatformAuthentication(Configuration)
 .AddMsal(new string[] { scopesToRequest })
 .AddDistributedTokenCaches();

// and then choose your implementation

// For instance the distributed in memory cache (not cleared when you stop the app)
services.AddDistributedMemoryCache()

// Or a Redis cache
services.AddStackExchangeRedisCache(options =>
{
 options.Configuration = "localhost";
 options.InstanceName = "SampleInstance";
});

// Or even a SQL Server token cache
services.AddDistributedSqlServerCache(options =>
{
 options.ConnectionString = _config["DistCache_ConnectionString"];
 options.SchemaName = "dbo";
 options.TableName = "TestCache";
});
```

For details about the token cache providers, see also the [ASP.NET Core Web app tutorials | Token caches](#) phase of the tutorial

## Next steps

At this point, when the user signs-in a token is stored in the token cache. Let's see how it's then used in other parts of the Web app.

[Sign in to the Web App](#)

# Web app that calls web APIs - sign in

10/18/2019 • 2 minutes to read • [Edit Online](#)

You already know how to add sign-in to your web app. You learn that in [Web app that signs-in users - add sign-in](#).

What is different here, is that when the user has signed out, from this application, or from any application, you want to remove from the token cache, the tokens associated with the user.

## Intercepting the callback after sign-out - Single Sign Out

Your application can intercept the after `logout` event, for instance to clear the entry of the token cache associated with the account that signed out. The web app will store access tokens for the user in a cache. Intercepting the after `logout` callback enables your web application to remove the user from the token cache.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

This mechanism is illustrated in the `AddMsal()` method of [WebAppServiceCollectionExtensions.cs#L151-L157](#)

The **Logout Url** that you've registered for your application enables you to implement single sign-out. The Microsoft identity platform `logout` endpoint will call the **Logout URL** registered with your application. This call happens if the sign-out was initiated from your web app, or from another web app or the browser. For more information, see [Single sign-out](#).

```
public static class WebAppServiceCollectionExtensions
{
 public static IServiceCollection AddMsal(this IServiceCollection services, IConfiguration configuration,
 IEnumerable<string> initialScopes, string configSectionName = "AzureAd")
 {
 // Code omitted here

 services.Configure<OpenIdConnectOptions>(AzureADDefaults.OpenIdScheme, options =>
 {
 // Code omitted here

 // Handling the sign-out: removing the account from MSAL.NET cache
 options.Events.OnRedirectToIdentityProviderForSignOut = async context =>
 {
 // Remove the account from MSAL.NET token cache
 var tokenAcquisition = context.HttpContext.RequestServices.GetRequiredService<ITokenAcquisition>();
 await tokenAcquisition.RemoveAccountAsync(context).ConfigureAwait(false);
 };
 });
 return services;
 }
}
```

The code for `RemoveAccountAsync` is available from [Microsoft.Identity.Web/TokenAcquisition.cs#L264-L288](#).

## Next steps

- [ASP.NET Core](#)

- [ASP.NET](#)
- [Java](#)
- [Python](#)

[Acquiring a token for the web app](#)

# Web app that calls web APIs - acquire a token for the app

10/30/2019 • 2 minutes to read • [Edit Online](#)

Now that you have built your client application object, you'll use it to acquire a token to call a web API. In ASP.NET or ASP.NET Core, calling a web API is then done in the controller. It's about:

- Getting a token for the web API using the token cache. To get this token, you call `AcquireTokenSilent`.
- Calling the protected API with the access token.
- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

The controller methods are protected by an `[Authorize]` attribute that forces users being authenticated to use the Web App. Here is the code that calls Microsoft Graph.

```
[Authorize]
public class HomeController : Controller
{
 readonly ITokenAcquisition tokenAcquisition;

 public HomeController(ITokenAcquisition tokenAcquisition)
 {
 this.tokenAcquisition = tokenAcquisition;
 }

 // Code for the controller actions(see code below)

}
```

The `ITokenAcquisition` service is injected by ASP.NET through dependency injection.

Here is a simplified code of the action of the HomeController, which gets a token to call the Microsoft Graph.

```
public async Task<IActionResult> Profile()
{
 // Acquire the access token
 string[] scopes = new string[]{"user.read"};
 string accessToken = await tokenAcquisition.GetAccessTokenOnBehalfOfUserAsync(scopes);

 // use the access token to call a protected web API
 HttpClient client = new HttpClient();
 client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", accessToken);
 string json = await client.GetStringAsync(url);
}
```

To understand more thoroughly the code required for this scenario, see the phase 2 ([2-1-Web App Calls Microsoft Graph](#)) step of the [ms-identity-aspnetcore-webapp-tutorial](#) tutorial.

There are many additional complexities, such as:

- Calling several APIs,

- processing incremental consent and Conditional Access.

These advanced steps are processed in chapter 3 of the tutorial [3-WebApp-multi-APIs](#)

# Web app that calls web APIs - call a web API

10/30/2019 • 2 minutes to read • [Edit Online](#)

Now that you have a token, you can call a protected web API.

- [ASP.NET Core](#)
- [Java](#)
- [Python](#)

Here is a simplified code of the action of the `HomeController`. This code gets a token to call the Microsoft Graph.

This time code was added, showing how to call Microsoft Graph as a REST API. The URL for the graph API is provided in the `appsettings.json` file and read in a variable named `webOptions`:

```
{
 "AzureAd": {
 "Instance": "https://login.microsoftonline.com/",
 ...
 },
 ...
 "GraphApiUrl": "https://graph.microsoft.com"
}
```

```

public async Task<IActionResult> Profile()
{
 var application = BuildConfidentialClientApplication(HttpContext, HttpContext.User);
 string accountIdentifier = claimsPrincipal.GetMsalAccountId();
 string loginHint = claimsPrincipal.GetLoginHint();

 // Get the account
 IAccount account = await application.GetAccountAsync(accountIdentifier);

 // Special case for guest users as the Guest iod / tenant id are not surfaced.
 if (account == null)
 {
 var accounts = await application.GetAccountsAsync();
 account = accounts.FirstOrDefault(a => a.Username == loginHint);
 }

 AuthenticationResult result;
 result = await application.AcquireTokenSilent(new []{"user.read"}, account)
 .ExecuteAsync();
 var accessToken = result.AccessToken;

 // Calls the web API (here the graph)
 HttpClient httpClient = new HttpClient();
 httpClient.DefaultRequestHeaders.Authorization =
 new AuthenticationHeaderValue(Constants.BearerAuthorizationScheme, accessToken);
 var response = await httpClient.GetAsync($"{webOptions.GraphApiUrl}/beta/me");

 if (response.StatusCode == HttpStatusCode.OK)
 {
 var content = await response.Content.ReadAsStringAsync();

 dynamic me = JsonConvert.DeserializeObject(content);
 return me;
 }

 ViewData["Me"] = me;
 return View();
}

```

#### **NOTE**

You can use the same principle to call any web API.

Most Azure web APIs provide an SDK that simplifies calling it. This is also the case of the Microsoft Graph. You'll learn in the next article where to find a tutorial illustrating these aspects.

## Next steps

[Move to production](#)

# Web app that calls web APIs - move to production

5/6/2019 • 2 minutes to read • [Edit Online](#)

Now that you know how to acquire a token to call Web APIs, learn how to move to production.

Make your application great:

- Enable [logging](#).
- Enable telemetry.
- Enable [proxies and customize HTTP clients](#).

Test your integration:

- Use the [Microsoft identity platform integration checklist](#).

## Next steps

Learn more by trying out the full ASP.NET Core web app progressive tutorial, which shows:

- How to sign in users with multiple audiences, national clouds, or with social identities
- Calls Microsoft Graph
- Calls several Microsoft APIs
- Handles incremental consent
- Calls your own Web API

[ASP.NET Core web app tutorial](#)

# Scenario: Protected web API

10/23/2019 • 2 minutes to read • [Edit Online](#)

In this scenario, we'll show you how you can expose a web API and how you can protect it so that only authenticated users can access the API. You'll want to enable authenticated users with both work and school accounts, or personal Microsoft personal accounts to use your web API.

## Prerequisites

Before reading this article, you should be familiar with the following concepts or read the following articles:

- [Microsoft identity platform overview](#)
- [Authentication basics](#)
- [Audiences](#)
- [Application and service principals](#)
- [Permissions and consent](#)
- [ID tokens and access tokens](#)

## Specifics

Here are some specifics you need to know to protect web APIs:

- Your app registration must expose at least one scope. The token version accepted by your web API depends on the sign in audience.
- The configuration of the code for the web API must validate the token that's used when calling the web API.

## Next steps

[App registration](#)

# Protected web API: App registration

10/23/2019 • 5 minutes to read • [Edit Online](#)

This article explains the specifics of app registration for a protected web API.

See [Quickstart: Register an application with the Microsoft identity platform](#) for the common steps for registering an app.

## Accepted token version

The Microsoft identity platform endpoint can issue two types of tokens: v1.0 tokens and v2.0 tokens. For more information about these tokens, see [Access tokens](#). The accepted token version depends on the **Supported account types** you chose when you created your application:

- If the value of **Supported account types** is **Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)**, the accepted token version will be v2.0.
- Otherwise, the accepted token version will be v1.0.

After you create the application, you can determine or change the accepted token version by following these steps:

1. In the Azure portal, select your app and then select the **Manifest** for your app.
2. In the manifest, search for "**accessTokenAcceptedVersion**". Note that its value is **2**. This property specifies to Azure Active Directory (Azure AD) that the web API accepts v2.0 tokens. If the value is **null**, the accepted token version is v1.0.
3. If you've changed the token version, select **Save**.

### NOTE

The web API specifies which token version (v1.0 or v2.0) it accepts. When clients request a token for your web API from the Microsoft identity platform (v2.0) endpoint, they'll get a token that indicates which version is accepted by the web API.

## No redirect URI

Web APIs don't need to register a redirect URI because no user is signed in interactively.

## Expose an API

Another setting specific to web APIs is the exposed API and the exposed scopes.

### Resource URI and scopes

Scopes are usually in the form `resourceURI/scopedName`. For Microsoft Graph, the scopes have shortcuts like `User.Read`. This string is a shortcut for `https://graph.microsoft.com/user.read`.

During app registration, you'll need to define these parameters:

- The resource URI. By default, the application registration portal recommends that you to use `api://{clientId}`. This resource URI is unique, but it's not human readable. You can change it, but make sure the new value is unique.
- One or more *scopes*. (To client applications, they'll show up as *delegated permissions* for your web API.)
- One or more *app roles*. (To client applications, they'll show up as *application permissions* for your web API.)

The scopes are also displayed on the consent screen that's presented to end users of your app. So you'll need to provide the corresponding strings that describe the scope:

- As seen by the end user.
- As seen by the tenant admin, who can grant admin consent.

### Exposing delegated permissions (scopes)

1. Select the **Expose an API** section in the application registration.
2. Select **Add a scope**.
3. If prompted, accept the proposed Application ID URI (`api://{clientId}`) by selecting **Save and Continue**.
4. Enter these parameters:
  - For **Scope name**, use `access_as_user`.
  - For **Who can consent**, make sure **Admins and users** is selected.
  - In **Admin consent display name**, enter **Access TodoListService as a user**.
  - In **Admin consent description**, enter **Accesses the TodoListService Web API as a user**.
  - In **User consent display name**, enter **Access TodoListService as a user**.
  - In **User consent description**, enter **Accesses the TodoListService Web API as a user**.
  - Keep **State** set to **Enabled**.
  - Select **Add scope**.

### If your web API is called by a daemon app

In this section, you'll learn how to register your protected web API so it can be securely called by daemon apps.

- You'll need to expose *application permissions*. You'll declare only application permissions because daemon apps don't interact with users, so delegated permissions wouldn't make sense.
- Tenant admins can require Azure Active Directory (Azure AD) to issue tokens for your web API only to applications that have registered to access one of the web API's application permissions.

### Exposing application permissions (app roles)

To expose application permissions, you'll need to edit the manifest.

1. In the application registration for your application, select **Manifest**.
2. Edit the manifest by locating the `appRoles` setting and adding one or more application roles. The role definition is provided in the following sample JSON block. Leave the `allowedMemberTypes` set to `"Application"` only. Make sure the `id` is a unique GUID and that `displayName` and `value` don't contain spaces.
3. Save the manifest.

The following sample shows the contents of `appRoles`. (The `id` can be any unique GUID.)

```
"appRoles": [
 {
 "allowedMemberTypes": ["Application"],
 "description": "Accesses the TodoListService-Cert as an application.",
 "displayName": "access_as_application",
 "id": "ccf784a6-fd0c-45f2-9c08-2f9d162a0628",
 "isEnabled": true,
 "lang": null,
 "origin": "Application",
 "value": "access_as_application"
 }
,
```

### Ensuring that Azure AD issues tokens for your web API to only allowed clients

The web API checks for the app role. (That's the developer way to expose application permissions.) But you can also configure Azure AD to issue a token for your web API only to apps that are approved by the tenant admin to

access your API. To add this increased security:

1. On the app **Overview** page for your app registration, select the link with the name of your app under **Managed application in local directory**. The title for this field might be truncated. You might, for example, see **Managed application in ...**

**NOTE**

When you select this link, you'll go to the **Enterprise Application Overview** page associated with the service principal for your application in the tenant where you created it. You can navigate back to the app registration page by using the back button of your browser.

2. Select the **Properties** page in the **Manage** section of the Enterprise application pages.
3. If you want Azure AD to allow access to your web API from only certain clients, set **User assignment required?** to **Yes**.

**IMPORTANT**

If you set **User assignment required?** to **Yes**, Azure AD will check the app role assignments of clients when they request an access token for the web API. If the client isn't assigned to any app roles, Azure AD will return the error

```
invalid_client: AADSTS501051: Application <application name> is not assigned to a role for the <web API>
```

If you keep **User assignment required?** set to **No**, Azure AD won't check the app role assignments when a client requests an access token for your web API. Any daemon client (that is, any client using the client credentials flow) will be able to obtain an access token for the API just by specifying its audience. Any application will be able to access the API without having to request permissions for it. But your web API can always, as explained in the previous section, verify that the application has the right role (which is authorized by the tenant admin). The API performs this verification by validating that the access token has a roles claim and that the value for this claim is correct. (In our case, the value is `access_as_application`.)

4. Select **Save**.

## Next steps

[App's code configuration](#)

# Protected web API: Code configuration

11/8/2019 • 4 minutes to read • [Edit Online](#)

To configure the code for your protected web API, you need to understand what defines APIs as protected, how to configure a bearer token, and how to validate the token.

## What defines ASP.NET/ASP.NET Core APIs as protected?

Like web apps, the ASP.NET/ASP.NET Core web APIs are "protected" because their controller actions are prefixed with the `[Authorize]` attribute. So the controller actions can be called only if the API is called with an identity that's authorized.

Consider the following questions:

- How does the web API know the identity of the app that calls it? (Only an app can call a web API.)
- If the app called the web API on behalf of a user, what's the user's identity?

## Bearer token

The information about the identity of the app, and about the user (unless the web app accepts service-to-service calls from a daemon app), is held in the bearer token that's set in the header when the app is called.

Here's a C# code example that shows a client calling the API after it acquires a token with Microsoft Authentication Library for .NET (MSAL.NET):

```
var scopes = new[] {"api://.../access_as_user"};
var result = await app.AcquireToken(scopes)
 .ExecuteAsync();

httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", result.AccessToken);

// Call the web API.
HttpResponseMessage response = await _httpClient.GetAsync(apiUri);
```

### IMPORTANT

The bearer token was requested by a client application to the Microsoft identity platform endpoint *for the web API*. The web API is the only application that should verify the token and view the claims it contains. Client apps should never try to inspect the claims in tokens. (The web API could require, in the future, that the token be encrypted. This requirement would prevent access for client apps that can view access tokens.)

## JwtBearer configuration

This section describes how to configure a bearer token.

### Config file

```
{
 "AzureAd": {
 "Instance": "https://login.microsoftonline.com/",
 "ClientId": "[Client_id-of-web-api-eg-2ec40e65-ba09-4853-bcde-bcb60029e596]",
 /*
 You need specify the TenantId only if you want to accept access tokens from a single tenant
 (line-of-business app).
 Otherwise, you can leave them set to common.
 This can be:
 - A GUID (Tenant ID = Directory ID)
 - 'common' (any organization and personal accounts)
 - 'organizations' (any organization)
 - 'consumers' (Microsoft personal accounts)
 */
 "TenantId": "common"
 },
 "Logging": {
 "LogLevel": {
 "Default": "Warning"
 }
 },
 "AllowedHosts": "*"
}
```

## Code initialization

When an app is called on a controller action that holds an `[Authorize]` attribute, ASP.NET/ASP.NET Core looks at the bearer token in the Authorization header of the calling request and extracts the access token. The token is then forwarded to the JwtBearer middleware, which calls Microsoft IdentityModel Extensions for .NET.

In ASP.NET Core, this middleware is initialized in the Startup.cs file:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
```

The middleware is added to the web API by this instruction:

```
services.AddAzureAdBearer(options => Configuration.Bind("AzureAd", options));
```

Currently, the ASP.NET Core templates create Azure Active Directory (Azure AD) web APIs that sign in users within your organization or any organization, not with personal accounts. But you can easily change them to use the Microsoft identity platform endpoint by adding this code to the Startup.cs file:

```
services.Configure<JwtBearerOptions>(AzureADDefaults.JwtBearerAuthenticationScheme, options =>
{
 // This is a Microsoft identity platform web API.
 options.Authority += "/v2.0";

 // The web API accepts as audiences both the Client ID (options.Audience) and api://{ClientID}.
 options.TokenValidationParameters.ValidAudiences = new []
 {
 options.Audience,
 $"api://{options.Audience}"
 };

 // Instead of using the default validation (validating against a single tenant,
 // as we do in line-of-business apps),
 // we inject our own multitenant validation logic (which even accepts both v1 and v2 tokens).
 options.TokenValidationParameters.IssuerValidator =
 AadIssuerValidator.GetIssuerValidator(options.Authority).Validate;;
});
```

This code snippet is extracted from the ASP.NET Core Web API incremental tutorial in [Microsoft.Identity.Web/WebApiServiceCollectionExtensions.cs#L50-L63](#). The `AddProtectedWebApi` method, which does a lot more, is called from the `Startup.cs`

## Token validation

The `JwtBearer` middleware, like the OpenID Connect middleware in web apps, is directed by `TokenValidationParameters` to validate the token. The token is decrypted (as needed), the claims are extracted, and the signature is verified. The middleware then validates the token by checking for this data:

- It's targeted for the web API (audience).
- It was issued for an app that's allowed to call the web API (sub).
- It was issued by a trusted security token service (STS) (issuer).
- Its lifetime is in range (expiry).
- It wasn't tampered with (signature).

There can also be special validations. For example, it's possible to validate that signing keys (when embedded in a token) are trusted and that the token isn't being replayed. Finally, some protocols require specific validations.

### Validators

The validation steps are captured in validators, which are all in the [Microsoft IdentityModel Extensions for .NET](#) open-source library, in one source file: [Microsoft.IdentityModel.Tokens/Validators.cs](#).

The validators are described in this table:

VALIDATOR	DESCRIPTION
<code>ValidateAudience</code>	Ensures the token is for the application that validates the token (for me).
<code>ValidateIssuer</code>	Ensures the token was issued by a trusted STS (from someone I trust).
<code>ValidateIssuerSigningKey</code>	Ensures the application validating the token trusts the key that was used to sign the token. (Special case where the key is embedded in the token. Not usually required.)
<code>ValidateLifetime</code>	Ensures the token is still (or already) valid. The validator checks if the lifetime of the token ( <code>notbefore</code> and <code>expires</code> claims) is in range.
<code>ValidateSignature</code>	Ensures the token hasn't been tampered with.
<code>ValidateTokenReplay</code>	Ensures the token isn't replayed. (Special case for some onetime use protocols.)

The validators are all associated with properties of the `TokenValidationParameters` class, themselves initialized from the ASP.NET/ASP.NET Core configuration. In most cases, you won't have to change the parameters. There's one exception, for apps that aren't single tenants. (That is, web apps that accept users from any organization or from personal Microsoft accounts.) In this case, the issuer must be validated.

## Token validation in Azure Functions

It's also possible to validate incoming access tokens in Azure functions. You can find examples of validating tokens in Azure functions in [Dotnet](#), [NodeJS](#), and [Python](#).

## Next steps

[Verify scopes and app roles in your code](#)

# Protected web API: Verify scopes and app roles

10/30/2019 • 4 minutes to read • [Edit Online](#)

This article describes how you can add authorization to your web API. This protection ensures that the API is called only by:

- Applications on behalf of users who have the right scopes.
- Daemon apps that have the right application roles.

## NOTE

The code snippets from this article are extracted from the following samples, which are fully functional

- [ASP.NET Core Web API incremental tutorial](#) on GitHub
- [ASP.NET Web API sample](#)

To protect an ASP.NET/ASP.NET Core web API, you'll need to add the `[Authorize]` attribute on one of these:

- The controller itself, if you want all the actions of the controller to be protected
- The individual controller action for your API

```
[Authorize]
public class TodoListController : Controller
{
 ...
}
```

But this protection isn't enough. It guarantees only that ASP.NET/ASP.NET Core will validate the token. Your API needs to verify that the token used to call your web API was requested with the claims it expects, in particular:

- The *scopes*, if the API is called on behalf of a user.
- The *app roles*, if the API can be called from a daemon app.

## Verifying scopes in APIs called on behalf of users

If your API is called by a client app on behalf of a user, it needs to request a bearer token that has specific scopes for the API. (See [Code configuration | Bearer token](#).)

```
[Authorize]
public class TodoListController : Controller
{
 /// <summary>
 /// The web API will accept only tokens 1) for users, 2) that have the `access_as_user` scope for
 /// this API.
 /// </summary>
 const string scopeRequiredByAPI = "access_as_user";

 // GET: api/values
 [HttpGet]
 public IEnumerable<TodoItem> Get()
 {
 VerifyUserHasAnyAcceptedScope(scopeRequiredByAPI);
 // Do the work and return the result.
 ...
 }
 ...
}
```

The `VerifyUserHasAnyAcceptedScope` method would do something like the following:

- Verify that there's a claim named `http://schemas.microsoft.com/identity/claims/scope` or `scp`.
- Verify that the claim has a value that contains the scope expected by the API.

```
/// <summary>
/// When applied to a <see cref="HttpContext"/>, verifies that the user authenticated in the
/// web API has any of the accepted scopes.
/// If the authenticated user doesn't have any of these <paramref name="acceptedScopes"/>, the
/// method throws an HTTP Unauthorized error with a message noting which scopes are expected in the token.
/// </summary>
/// <param name="acceptedScopes">Scopes accepted by this API</param>
/// <exception cref="HttpRequestException"/> with a <see cref="HttpResponse.StatusCode"/> set to
/// <see cref="HttpStatusCode.Unauthorized"/>
public static void VerifyUserHasAnyAcceptedScope(this HttpContext context,
 params string[] acceptedScopes)
{
 if (acceptedScopes == null)
 {
 throw new ArgumentNullException(nameof(acceptedScopes));
 }
 Claim scopeClaim = HttpContext?.User
 ?.FindFirst("http://schemas.microsoft.com/identity/claims/scope");
 if (scopeClaim == null || !scopeClaim.Value.Split(' ').Intersect(acceptedScopes).Any())
 {
 context.Response.StatusCode = (int) HttpStatusCode.Unauthorized;
 string message = $"The 'scope' claim does not contain scopes '{string.Join(", ", acceptedScopes)}' or was not found";
 throw new HttpRequestException(message);
 }
}
```

This [sample code](#) is for ASP.NET Core. For ASP.NET, just replace `HttpContext.User` with `ClaimsPrincipal.Current`, and replace the claim type `"http://schemas.microsoft.com/identity/claims/scope"` with `"scp"`. (See also the code snippet later in this article.)

## Verifying app roles in APIs called by daemon apps

If your web API is called by a [daemon app](#), that app should require an application permission to your web API. We've seen in [Exposing application permissions \(app roles\)](#) that your API exposes such permissions (for example, the `access_as_application` app role). You now need to have your APIs verify that the token it received contains the

`roles` claim and that this claim has the value it expects. The code doing this verification is similar to the code that verifies delegated permissions, except that, instead of testing for `scopes`, your controller action will test for `roles`:

```
[Authorize]
public class TodoListController : ApiController
{
 public IEnumerable<TodoItem> Get()
 {
 ValidateAppRole("access_as_application");
 ...
 }
}
```

The `ValidateAppRole()` method can be something like this:

```
private void ValidateAppRole(string appRole)
{
 //
 // The `role` claim tells you what permissions the client application has in the service.
 // In this case, we look for a `role` value of `access_as_application`.
 //
 Claim roleClaim = ClaimsPrincipal.Current.FindFirst("roles");
 if (roleClaim == null || !roleClaim.Value.Split(' ').Contains(appRole))
 {
 throw new HttpResponseException(new HttpResponseMessage
 {
 StatusCode = HttpStatusCode.Unauthorized,
 ReasonPhrase = $"The 'roles' claim does not contain '{appRole}' or was not found"
 });
 }
}
```

This time, the code snippet is for ASP.NET. For ASP.NET Core, just replace `ClaimsPrincipal.Current` with `HttpContext.User`, and replace the `"roles"` claim name with `"http://schemas.microsoft.com/identity/claims/roles"`. (See also the code snippet earlier in this article.)

### Accepting app-only tokens if the web API should be called only by daemon apps

The `roles` claim is also used for users in user assignment patterns. (See [How to: Add app roles in your application and receive them in the token](#).) So just checking roles will allow apps to sign in as users and the other way around, if the roles are assignable to both. We recommend that you declare different roles for users and apps to prevent this confusion.

If you want to allow only daemon apps to call your web API, add a condition, when you validate the app role, that the token is an app-only token:

```
string oid = ClaimsPrincipal.Current.FindFirst("oid")?.Value;
string sub = ClaimsPrincipal.Current.FindFirst("sub")?.Value;
bool isAppOnlyToken = oid == sub;
```

Checking the inverse condition will allow only apps that sign in a user to call your API.

## Next steps

[Move to production](#)

# Protected web API - move to production

10/23/2019 • 2 minutes to read • [Edit Online](#)

Now that you know how to protect your web API, here's how you can move it to production.

Make your application great:

- Enable [logging](#).
- Enable telemetry.
- Enable [proxies](#) and customize HTTP clients.

Test your integration:

- Use the [Microsoft identity platform integration checklist](#).

## Next steps

Learn how to call downstream APIs:

[Scenario - Web API calls downstream APIS](#)

Learn more with tutorials and samples:

[ASP.NET Core web API Tutorial](#)

[ASP.NET web API sample](#)

# Scenario: Web API that calls web APIs

8/7/2019 • 2 minutes to read • [Edit Online](#)

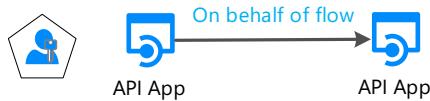
Learn all you need to build a web API that calls web APIs.

## Prerequisites

This scenario, protected web API that calls web APIs, builds on top of the "Protect a web API" scenario. To learn more about this foundational scenario, see [Protected Web API - Scenario](#) first.

## Overview

- A client (web, desktop, mobile, or single-page application) - not represented on the diagram below - calls a protected web API and provides a JWT bearer token in its "Authorization" Http header.
- The protected web API validates the token and uses the MSAL `AcquireTokenOnBehalfOf` method to request (from Azure AD) another token so that it can, itself, call a second web API (named the downstream web API) on behalf of the user.
- The protected web API uses this token to call a downstream API. It can also call `AcquireTokenSilent` later to request tokens for other downstream APIs (but still on behalf of the same user). `AcquireTokenSilent` refreshes the token when needed.



## Specifics

The part of app registration related to the API permissions is classical. The application configuration involves using the OAuth 2.0 on-behalf-of flow to exchange the JWT bearer token against a token for a downstream API. This token is added to the token cache, where it's available in the web API's controllers, and can acquire a token silently to call downstream APIs.

## Next steps

[App registration](#)

# Web API that calls web APIs - app registration

5/6/2019 • 2 minutes to read • [Edit Online](#)

A web API that calls downstream web APIs has the same registration as a protected web API. Therefore, you'll need to follow the instructions in [Protected Web API - app registration](#).

However, since the web app now calls web APIs, it becomes a confidential client application. That's why there's extra registration info that's required: the app needs to share secrets (client credentials) with the Microsoft identity platform.

## Registration of secrets or certificates

Like for any confidential client application, you need to register a secret or certificate. You can register your application secrets either through the interactive experience in the [Azure portal](#), or using command-line tools (like PowerShell)

### Registering client secrets using the application registration portal

The management of client credentials happens in the **certificates & secrets** page for an application:

The screenshot shows the Azure portal interface. The left sidebar contains a navigation menu with options like 'Create a resource', 'All services', 'FAVORITES' (which includes 'Dashboard', 'All resources', 'Resource groups', 'App Services', 'Function Apps', 'SQL databases', 'Azure Cosmos DB', 'Virtual machines', 'Load balancers', 'Storage accounts', 'Virtual networks', 'Azure Active Directory', 'Monitor', 'Advisor', 'Security Center', 'Cost Management + Billing', and 'Help + support'). The top navigation bar includes a search bar, account information ('isabell@contoso... CONTOSO ENTERPRISES'), and various icons for settings and help. The main content area is titled 'ContosoApp\_1 - Certificates & secrets' and shows two sections: 'Certificates' and 'Client secrets'. The 'Certificates' section has a button to 'Upload certificate'. The 'Client secrets' section has a button to '+ New client secret'. Both sections have tables with columns like 'THUMBPRINT', 'START DATE', 'EXPIRES', 'DESCRIPTION', and 'VALUE'.

- the application secret (also named client secret) is generated by Azure AD, during the registration of the confidential client application. This generation happens when you select **New client secret**. At that point, you must copy the secret string in the clipboard for use in your app, before selecting **Save**. This string won't be presented any longer.
- the certificate is uploaded in the application registration using the **Upload certificate** button. Azure AD only supports certificates that are directly registered on the application, and do not follow certificate chains.

For details, see [Quickstart: Configure a client application to access web APIs | Add credentials to your application](#)

### Registering client secrets using PowerShell

Alternatively, you can register your application with Azure AD using command-line tools. The [active-directory-dotnetcore-daemon-v2](#) sample shows how to register an application secret or a certificate with an Azure AD application:

- For details on how to register an application secret, see [AppCreationScripts/Configure.ps1](#)
- For details on how to register a certificate with the application, see [AppCreationScripts-withCert/Configure.ps1](#)

## API permissions

Web applications call APIs on behalf of the user for whom the bearer token was received. They need to request delegated permissions. For details, see [Add permissions to access web APIs](#).

## Next steps

[App's code configuration](#)

# Web API that calls web APIs - code configuration

11/12/2019 • 3 minutes to read • [Edit Online](#)

After you've registered your web API, you can configure the code for the application.

The code to configure your web API so that it calls downstream web APIs builds on top of the code used to protect a web API. For more info, see [Protected web API - app configuration](#).

## Code subscribed to OnTokenValidated

On top of the code configuration for any protected web APIs, you need to subscribe to the validation of the bearer token that's received when your API is called:

```
/// <summary>
/// Protects the web API with Microsoft Identity Platform (a.k.a AAD v2.0)
/// This supposes that the configuration files have a section named "AzureAD"
/// </summary>
/// <param name="services">Service collection to which to add authentication</param>
/// <param name="configuration">Configuration</param>
/// <returns></returns>
public static IServiceCollection AddProtectedApiCallsWebApis(this IServiceCollection services,
 IConfiguration configuration,
 IEnumerable<string> scopes)
{
 services.AddTokenAcquisition();
 services.Configure<JwtBearerOptions>(AzureADDefaults.JwtBearerAuthenticationScheme, options =>
 {
 // When an access token for our own web API is validated, we add it
 // to MSAL.NET's cache so that it can be used from the controllers.
 options.Events = new JwtBearerEvents();

 options.Events.OnTokenValidated = async context =>
 {
 context.Success();

 // Adds the token to the cache, and also handles the incremental consent
 // and claim challenges
 AddAccountToCacheFromJwt(context, scopes);
 await Task.FromResult(0);
 };
 });
 return services;
}
```

## On-behalf-of flow

The AddAccountToCacheFromJwt() method needs to:

- Instantiate an MSAL confidential client application.
- Call `AcquireTokenOnBehalf` to exchange the bearer token that was acquired by the client for the web API, against a bearer token for the same user, but for our API to call a downstream API.

### Instantiate a confidential client application

This flow is only available in the confidential client flow so the protected web API provides client credentials (client secret or certificate) to the `ConfidentialClientApplicationBuilder` via the `WithClientSecret` or `WithCertificate` methods, respectively.

**IConfidentialClientApplication**  
Interface  
↳ IClientApplicationBase

Methods

- ⌚ AcquireTokenByAuthorizationCode(IEnumerable<string> scopes, string authorizationCode) : AcquireTokenByAuthorizationCodeParameterBuilder
- ⌚ AcquireTokenForClient(IEnumerable<string> scopes) : AcquireTokenForClientParameterBuilder
- ⌚ AcquireTokenOnBehalfOf(IEnumerable<string> scopes, UserAssertion userAssertion) : AcquireTokenOnBehalfOfParameterBuilder
- ⌚ GetAuthorizationRequestUrl(IEnumerable<string> scopes) : GetAuthorizationRequestUrlParameterBuilder

```
IConfidentialClientApplication app;

#if !VariationWithCertificateCredentials
app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
 .WithClientSecret(config.ClientSecret)
 .Build();
#else
// Building the client credentials from a certificate
X509Certificate2 certificate = ReadCertificate(config.CertificateName);
app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
 .WithCertificate(certificate)
 .Build();
#endif
```

Finally, instead of a client secret or a certificate, confidential client applications can also prove their identity using client assertions. This advanced scenario is detailed in [Client assertions](#)

## How to call on-behalf-of

The on-behalf-of (OBO) call is done by calling the [AcquireTokenOnBehalf](#) method on the [IConfidentialClientApplication](#) interface.

The [UserAssertion](#) is built from the bearer token received by the web API from its own clients. There are [two constructors](#), one that takes a JWT bearer token, and one that takes any kind of user assertion (another kind of security token, which type is then specified in an additional parameter named [assertionType](#)).

**UserAssertion**  
Sealed Class

Properties

- 🔧 Assertion { get; set; } : string
- 🔧 AssertionType { get; set; } : string

Methods

- ⌚ UserAssertion(string assertion)
- ⌚ UserAssertion(string assertion, string assertionType)

In practice, the OBO flow is often used to acquire a token for a downstream API and store it in the MSAL.NET user token cache so that other parts of the web API can later call on the [overrides](#) of [AcquireTokenOnSilent](#) to call the downstream APIs. This call has the effect of refreshing the tokens, if needed.

```

private void AddAccountToCacheFromJwt(IEnumerable<string> scopes, JwtSecurityToken jwtToken, ClaimsPrincipal principal, HttpContext httpContext)
{
 try
 {
 UserAssertion userAssertion;
 IEnumerable<string> requestedScopes;
 if (jwtToken != null)
 {
 userAssertion = new UserAssertion(jwtToken.RawData, "urn:ietf:params:oauth:grant-type:jwt-bearer");
 requestedScopes = scopes ?? jwtToken.Audiences.Select(a => $"{a}/.default");
 }
 else
 {
 throw new ArgumentOutOfRangeException("tokenValidationContext.SecurityToken should be a JWT Token");
 }

 // Create the application
 var application = BuildConfidentialClientApplication(httpContext, principal);

 // .Result to make sure that the cache is filled-in before the controller tries to get access tokens
 var result = application.AcquireTokenOnBehalfOf(requestedScopes.Except(scopesRequestedByMsalNet),
 userAssertion)
 .ExecuteAsync()
 .GetAwaiter().GetResult();
 }
 catch (MsalException ex)
 {
 Debug.WriteLine(ex.Message);
 throw;
 }
}

```

You can also see an example of on behalf of flow implementation in [NodeJS](#) and [Azure Functions](#).

## Protocol

For more information about the on-behalf-of protocol, see [Microsoft identity platform and OAuth 2.0 On-Behalf-Of flow](#).

## Next steps

[Acquiring a token for the app](#)

# Web API that calls web APIs - acquire a token for the app

5/7/2019 • 2 minutes to read • [Edit Online](#)

Once you've built a client application object, use it to acquire a token that you can use to call a web API.

## Code in the controller

Here's an example of code that will be called in the actions of the API controllers, calling a downstream API (named todolist).

```
private async Task GetTodoList(bool isAppStarting)
{
 ...
 //
 // Get an access token to call the To Do service.
 //
 AuthenticationResult result = null;
 try
 {
 app = BuildConfidentialClient(HttpContext, HttpContext.User);
 result = await app.AcquireTokenSilent(Scopes, account)
 .ExecuteAsync()
 .ConfigureAwait(false);
 }
 ...
}
```

`BuildConfidentialClient()` is similar to what you've seen in the article [Web API that calls web APIs - app configuration](#). `BuildConfidentialClient()` instantiates `IConfidentialClientApplication` with a cache that contains only information for one account. The account is provided by the `GetAccountIdentifier` method.

The `GetAccountIdentifier` method uses the claims associated with the identity of the user for which the web API received the JWT:

```
public static string GetMsalAccountId(this ClaimsPrincipal claimsPrincipal)
{
 string userObjectId = GetObjectId(claimsPrincipal);
 string tenantId = GetTenantId(claimsPrincipal);

 if (!string.IsNullOrWhiteSpace(userObjectId)
 && !string.IsNullOrWhiteSpace(tenantId))
 {
 return $"{userObjectId}.{tenantId}";
 }

 return null;
}
```

## Next steps

[Calling a web API](#)

# Web API that calls web APIs - call an API

5/6/2019 • 2 minutes to read • [Edit Online](#)

Once you have a token, you can call a protected web API. This is done from the controller of your ASP.NET/ASP.NET Core web API.

## Controller code

Here's the continuation of the example code shown in [Protected web API calls web APIs - acquiring a token](#), called in the actions of the API controllers, calling a downstream API (named todolist).

Once you acquired the token, use it as a bearer token to call the downstream API.

```
private async Task GetTodoList(bool isAppStarting)
{
 ...
 //
 // Get an access token to call the To Do service.
 //
 AuthenticationResult result = null;
 try
 {
 app = BuildConfidentialClient(HttpContext, HttpContext.User);
 result = await app.AcquireTokenSilent(Scopes, account)
 .ExecuteAsync()
 .ConfigureAwait(false);
 }
 ...

 // Once the token has been returned by MSAL, add it to the http authorization header, before making the call
 // to access the To Do list service.
 _httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", result.AccessToken);

 // Call the To Do list service.
 HttpResponseMessage response = await _httpClient.GetAsync(TodoListBaseAddress + "/api/todolist");
 ...
}
```

## Next steps

[Move to production](#)

# Web API that calls web APIs - move to production

5/6/2019 • 2 minutes to read • [Edit Online](#)

Once you've acquired a token to call web APIs, you can move your app to production.

Make your application great:

- Enable [logging](#).
- Enable telemetry.
- Enable [proxies and customize HTTP clients](#).

Test your integration:

- Use the [Microsoft identity platform integration checklist](#).

## Learn more

Now that you know the basics of how to call web APIs from your own web API, you might be interested in this tutorial, which describes the code that's used to build a protected web API calling web APIs.

SAMPLE	PLATFORM	DESCRIPTION
<a href="#">active-directory-aspnetcore-webapi-tutorial-v2</a>	ASP.NET Core 2.2 Web API, Desktop (WPF)	ASP.NET Core 2.2 Web API calling Microsoft Graph, itself called from a WPF application using the Microsoft identity platform (v2.0)

# Scenario: Desktop app that calls web APIs

9/25/2019 • 2 minutes to read • [Edit Online](#)

Learn all you need to build a Desktop app that calls web APIs

## Prerequisites

Before reading this article, you should be familiar with the following concepts or read the following articles:

- [Microsoft identity platform overview](#)
- [Authentication basics](#)
- [Audiences](#)
- [Application and service principals](#)
- [Permissions and consent](#)
- [ID tokens and access tokens](#)

## Getting started

If you haven't already, create your first application by following the .NET desktop quickstart, the UWP quickstart or the macOS native app quickstart:

[Quickstart: Acquire a token and call Microsoft Graph API from a Windows desktop app](#)

[Quickstart: Acquire a token and call Microsoft Graph API from a UWP app](#)

[Quickstart: Acquire a token and call Microsoft Graph API from a macOS native app](#)

## Overview

You write a desktop application, and you want to sign in users to your application and call web APIs such as the Microsoft Graph, other Microsoft APIs, or your own web API. You have several possibilities:

- You can use the interactive token acquisition:
  - If your desktop application supports graphical controls, for instance if it's a Windows.Form application, a WPF application or a macOS native application.
  - Or if it's a .NET Core application and you agree to have the authentication interaction with Azure AD happen in the system browser
- For Windows hosted applications, it's also possible for applications running on computers joined to a Windows domain or AAD joined to acquire a token silently by using Integrated Windows Authentication.
- Finally, and although it's not recommended, you can use Username/Password in public client applications. It's still needed in some scenarios (like DevOps), but beware that using it will impose constraints on your application. For instance, it can't sign in user who needs to perform multi-factor authentication (Conditional Access). Also your application won't benefit from single sign-on (SSO).

It's also against the principles of modern authentication and is only provided for legacy reasons.

- Authorization code flow (with PKCE)
- Integrated Windows Authentication
- Username/Password



- If you're writing a portable command-line tool - probably a .NET Core application running on Linux or Mac - and if you accept that the authentication be delegated to the system browser, you will be able to use interactive authentication (.NET Core doesn't provide yet a [Web browser](#) and therefore the authentication happens in the system browser), Otherwise, the best option in that case is to use device code flow. This flow is also used for applications without a browser, such as IoT applications



## Specifics

Desktop applications have a number of specificities, which depends mainly on whether your application uses the interactive authentication or not.

## Next steps

[Desktop app - app registration](#)

# Desktop app that calls web APIs - app registration

9/25/2019 • 2 minutes to read • [Edit Online](#)

This article contains the app registration specificities for a desktop application.

## Supported accounts types

The account types supported in desktop application depend on the experience that you want to light up. Because of this relationship, the supported account types depend on the flows that you want to use.

### Audience for interactive token acquisition

If your desktop application uses interactive authentication, you can sign in users from any [account type](#).

### Audience for desktop app silent flows

- To use Integrated Windows authentication or username/password, your application needs to sign in users in your own tenant (LOB developer), or in Azure Active directory organizations (ISV scenario). These authentication flows aren't supported for Microsoft personal accounts.
- If you want to use the Device code flow, you can't sign in users with their Microsoft personal accounts yet.
- If you sign in users with social identities passing a B2C authority and policy, you can only use the interactive and username-password authentication.

## Redirect URIs

The redirect URIs to use in desktop application will depend on the flow you want to use.

- If you're using the **interactive authentication** or **Device Code Flow**, you'll want to use <https://login.microsoftonline.com/common/oauth2/nativeclient>. You'll achieve this configuration by clicking the corresponding URL in the **Authentication** section for your application.

### IMPORTANT

Today MSAL.NET uses another Redirect URI by default in desktop applications running on Windows (<urn:ietf:wg:oauth:2.0:oob>). In the future we'll want to change this default, and therefore we recommend that you use <https://login.microsoftonline.com/common/oauth2/nativeclient>

- If you're building a native Objective-C or Swift app for macOS, you'll want to register the redirectUri based on your application's bundle identifier in the following format: **msauth.<your.app.bundle.id>://auth** (replace <your.app.bundle.id> with your application's bundle identifier)
- If your app is only using Integrated Windows authentication or username/password, you don't need to register a redirect URI for your application. These flows do a round trip to the Microsoft identity platform v2.0 endpoint, and your application won't be called back on any specific URI.
- To distinguish Device Code Flow, Integrated Windows authentication, and username/password from a confidential client application flow that doesn't have redirect URIs either (the client credential flow used in daemon applications), you need to express that your application is a public client application. To achieve this configuration, go to the **Authentication** section for your application. Then, in the **Advanced settings** subsection, in the **Default client type** paragraph, choose **Yes** to the question **Treat application as a public client**.

The screenshot shows the Azure portal interface for managing app registrations. The left sidebar contains navigation links for Home, Dashboard, All services, Favorites (with App Services selected), Resource groups, App Services, Function Apps, SQL databases, Azure Cosmos DB, Virtual machines, Load balancers, Storage accounts, Virtual networks, Azure Active Directory, Monitor, Advisor, and Security Center. The main content area is titled 'up-console - Authentication' (PREVIEW). It includes sections for Overview, Quickstart, Advanced settings, Implicit grant, and Default client type. A red box highlights the 'Default client type' section, which asks if the application should be treated as a public client. The 'Yes' button is highlighted.

## API permissions

Desktop applications call APIs for the signed-in user. They need to request delegated permissions. However, they can't request application permissions, which are only handled in [daemon applications](#).

## Next steps

[Desktop app - app configuration](#)

# Desktop app that calls web APIs - code configuration

10/30/2019 • 4 minutes to read • [Edit Online](#)

Now that you've created your application, you'll learn how to configure the code with the application's coordinates.

## MSAL libraries

The Microsoft libraries supporting desktop applications are:

MSAL LIBRARY	DESCRIPTION
 MSAL.NET	Supports building a desktop application in multiple platforms- Linux, Windows and MacOS
 MSAL Python	Supports building a desktop application in multiple platforms. Development in progress - in public preview
 MSAL Java	Supports building a desktop application in multiple platforms. Development in progress - in public preview
iOS  MSAL iOS	Supports desktop applications running on macOS only

## Public client application

From a code point of view, desktop applications are public client applications. The configuration will be a bit different based on whether you use interactive authentication or not.

- [.NET](#)
- [Java](#)
- [Python](#)
- [MacOS](#)

You'll need to build and manipulate MSAL.NET `IPublicClientApplication`.

**IPublicClientApplication**

Interface

→ IClientApplicationBase

**Properties**

- IsSystemWebViewAvailable { get; } : bool

**Methods**

- AcquireTokenByIntegratedWindowsAuth(IEnumerable<string> scopes) : AcquireTokenByIntegratedWindowsA...
- AcquireTokenByUsernamePassword(IEnumerable<string> scopes, string username, SecureString password) : A...
- AcquireTokenInteractive(IEnumerable<string> scopes, object parent) : AcquireTokenInteractiveParameterBuil...
- AcquireTokenWithDeviceCode(IEnumerable<string> scopes, Func<DeviceCodeResult, Task> deviceCodeResul...

## Exclusively by code

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

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

If you intend to use interactive authentication or Device Code Flow, as seen above, you want to use the `.WithRedirectUri` modifier:

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

## Using configuration files

The following code instantiates a Public client application from a configuration object, which could be filled-in programmatically or read from a configuration file

```
PublicClientApplicationOptions options = GetOptions(); // your own method
IPublicClientApplication app = PublicClientApplicationBuilder.CreateWithApplicationOptions(options)
 .WithDefaultRedirectUri()
 .Build();
```

## More elaborated configuration

You can elaborate the application building by adding a number of modifiers. For instance, if you want your application to be a multi-tenant application in a national cloud (here US Government), you could write:

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
 .WithDefaultRedirectUri()
 .WithAadAuthority(AzureCloudInstance.AzureUsGovernment,
 AadAuthorityAudience.AzureAdMultipleOrgs)
 .Build();
```

MSAL.NET also contains a modifier for ADFS 2019:

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
 .WithAdfsAuthority("https://consoso.com/adfs")
 .Build();
```

Finally, if you want to acquire tokens for an Azure AD B2C tenant, can specify your tenant as shown in the following code snippet:

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
 .WithB2CAuthority("https://fabrikamb2c.b2clogin.com/tfp/{tenant}/{PolicySignInSignUp}")
 .Build();
```

## Learn more

To learn more on how to configure an MSAL.NET desktop application:

- For the list of all modifiers available on `PublicClientApplicationBuilder`, see the reference documentation [PublicClientApplicationBuilder](#)
- For the description of all the options exposed in `PublicClientApplicationOptions` see [PublicClientApplicationOptions](#), in the reference documentation

## Complete example with configuration Options

Imagine a .NET Core console application that has the following `appsettings.json` configuration file:

```
{
 "Authentication": {
 "AzureCloudInstance": "AzurePublic",
 "AadAuthorityAudience": "AzureAdMultipleOrgs",
 "ClientId": "ebe2ab4d-12b3-4446-8480-5c3828d04c50"
 },
 "WebAPI": {
 "MicrosoftGraphBaseEndpoint": "https://graph.microsoft.com"
 }
}
```

You have little code to read this file using the .NET provided configuration framework;

```

public class SampleConfiguration
{
 /// <summary>
 /// Authentication options
 /// </summary>
 public PublicClientApplicationOptions PublicClientApplicationOptions { get; set; }

 /// <summary>
 /// Base URL for Microsoft Graph (it varies depending on whether the application is ran
 /// in Microsoft Azure public clouds or national / sovereign clouds
 /// </summary>
 public string MicrosoftGraphBaseEndpoint { get; set; }

 /// <summary>
 /// Reads the configuration from a json file
 /// </summary>
 /// <param name="path">Path to the configuration json file</param>
 /// <returns>SampleConfiguration as read from the json file</returns>
 public static SampleConfiguration ReadFromJsonFile(string path)
 {
 // .NET configuration
 IConfigurationRoot Configuration;
 var builder = new ConfigurationBuilder()
 .SetBasePath(Directory.GetCurrentDirectory())
 .AddJsonFile(path);
 Configuration = builder.Build();

 // Read the auth and graph endpoint config
 SampleConfiguration config = new SampleConfiguration()
 {
 PublicClientApplicationOptions = new PublicClientApplicationOptions()
 };
 Configuration.Bind("Authentication", config.PublicClientApplicationOptions);
 config.MicrosoftGraphBaseEndpoint =
 Configuration.GetValue<string>("WebAPI:MicrosoftGraphBaseEndpoint");
 return config;
 }
}

```

Now, to create your application, you'll just need to write the following code:

```

SampleConfiguration config = SampleConfiguration.ReadFromJsonFile("appsettings.json");
var app = PublicClientApplicationBuilder.CreateWithApplicationOptions(config.PublicClientApplicationOptions)
 .WithDefaultRedirectUri()
 .Build();

```

and before the call to the `.Build()` method, you can override your configuration with calls to `.Withxxx` methods as seen previously.

## Next steps

[Acquiring a token for a desktop app](#)

# Desktop app that calls web APIs - acquire a token

10/30/2019 • 35 minutes to read • [Edit Online](#)

Once you have built an instance of the Public Client application, you'll use it to acquire a token that you'll then use to call a web API.

## Recommended pattern

The web API is defined by its `scopes`. Whatever the experience you provide in your application, the pattern that you'll want to use is:

- Systematically attempting to get a token from the token cache by calling `AcquireTokenSilent`
- If this call fails, use the `AcquireToken` flow that you want to use (here represented by `AcquireTokenXX`)
- [.NET](#)
- [Java](#)
- [Python](#)
- [MacOS](#)

### In MSAL.NET

```
AuthenticationResult result;
var accounts = await app.GetAccountsAsync();
IAccount account = ChooseAccount(accounts); // for instance accounts.FirstOrDefault
 // if the app manages is at most one account
try
{
 result = await app.AcquireTokenSilent(scopes, account)
 .ExecuteAsync();
}
catch(MsalUiRequiredException ex)
{
 result = await app.AcquireTokenXX(scopes, account)
 .WithOptionalParameterXXX(parameter)
 .ExecuteAsync();
}
```

Here is now the detail of the various ways to acquire tokens in a desktop application

## Acquiring a token interactively

The following example shows minimal code to get a token interactively for reading the user's profile with Microsoft Graph.

- [.NET](#)
- [Java](#)
- [Python](#)
- [MacOS](#)

### In MSAL.NET

```

string[] scopes = new string[] {"user.read"};
var app = PublicClientApplicationBuilder.Create(clientId).Build();
var accounts = await app.GetAccountsAsync();
AuthenticationResult result;
try
{
 result = await app.AcquireTokenSilent(scopes, accounts.FirstOrDefault())
 .ExecuteAsync();
}
catch(MsalUiRequiredException)
{
 result = await app.AcquireTokenInteractive(scopes)
 .ExecuteAsync();
}

```

## Mandatory parameters

`AcquireTokenInteractive` has only one mandatory parameter `scopes`, which contains an enumeration of strings that define the scopes for which a token is required. If the token is for the Microsoft Graph, the required scopes can be found in api reference of each Microsoft graph API in the section named "Permissions". For instance, to [list the user's contacts](#), the scope "User.Read", "Contacts.Read" will need to be used. See also [Microsoft Graph permissions reference](#).

On Android, you need to also specify the parent activity (using `.WithParentActivityOrWindow`, see below) so that the token gets back to that parent activity after the interaction. If you don't specify it, an exception will be thrown when calling `.ExecuteAsync()`.

## Specific optional parameters in MSAL.NET

### `WithParentActivityOrWindow`

Being interactive, UI is important. `AcquireTokenInteractive` has one specific optional parameter enabling to specify, for platforms supporting it, the parent UI. When used in a desktop application, `.WithParentActivityOrWindow` has a different type depending on the platform:

```

// net45
WithParentActivityOrWindow(IntPtr windowPtr)
WithParentActivityOrWindow(IWin32Window window)

// Mac
WithParentActivityOrWindow(NSWindow window)

// .Net Standard (this will be on all platforms at runtime, but only on NetStandard at build time)
WithParentActivityOrWindow(object parent).

```

Remarks:

- On .NET Standard, the expected `object` is an `Activity` on Android, a `UIViewController` on iOS, an `NSWindow` on MAC, and a `IWin32Window` OR `IntPtr` on Windows.
- On Windows, you must call `AcquireTokenInteractive` from the UI thread so that the embedded browser gets the appropriate UI synchronization context. Not calling from the UI thread may cause messages to not pump properly and/or deadlock scenarios with the UI. One way of calling MSAL from the UI thread if you aren't on the UI thread already is to use the `Dispatcher` on WPF.
- If you're using WPF, to get a window from a WPF control, you can use `WindowInteropHelper.Handle` class. The call is then, from a WPF control (`this`):

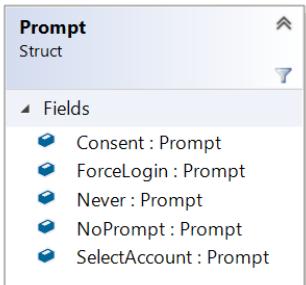
```

result = await app.AcquireTokenInteractive(scopes)
 .WithParentActivityOrWindow(new WindowInteropHelper(this).Handle)
 .ExecuteAsync();

```

### WithPrompt

`WithPrompt()` is used to control the interactivity with the user by specifying a Prompt



The class defines the following constants:

- `SelectAccount` : will force the STS to present the account selection dialog containing accounts for which the user has a session. This option is useful when applications developers want to let user choose among different identities. This option drives MSAL to send `prompt=select_account` to the identity provider. This option is the default, and it does of good job of providing the best possible experience based on the available information (account, presence of a session for the user, and so on....). Don't change it unless you have good reason to do it.
- `Consent` : enables the application developer to force the user be prompted for consent even if consent was granted before. In this case, MSAL sends `prompt=consent` to the identity provider. This option can be used in some security focused applications where the organization governance demands that the user is presented the consent dialog each time the application is used.
- `ForceLogin` : enables the application developer to have the user prompted for credentials by the service even if this user-prompt wouldn't be needed. This option can be useful if Acquiring a token fails, to let the user re-sign-in. In this case, MSAL sends `prompt=login` to the identity provider. Again, we've seen it used in some security focused applications where the organization governance demands that the user relogs-in each time they access specific parts of an application.
- `Never` (for .NET 4.5 and WinRT only) won't prompt the user, but instead will try to use the cookie stored in the hidden embedded web view (See below: Web Views in MSAL.NET). Using this option might fail, and in that case `AcquireTokenInteractive` will throw an exception to notify that a UI interaction is needed, and you'll need to use another `Prompt` parameter.
- `NoPrompt` : Won't send any prompt to the identity provider. This option is only useful for Azure AD B2C edit profile policies (See [B2C specifics](#)).

### WithExtraScopeToConsent

This modifier is used in an advanced scenario where you want the user to pre-consent to several resources upfront (and don't want to use the incremental consent, which is normally used with MSAL.NET / the Microsoft identity platform). For details see [How-to : have the user consent upfront for several resources](#).

```

var result = await app.AcquireTokenInteractive(scopesForCustomerApi)
 .WithExtraScopeToConsent(scopesForVendorApi)
 .ExecuteAsync();

```

### WithCustomWebUi

A Web UI is a mechanism to invoke a browser. This mechanism can be a dedicated UI `WebBrowser` control or a way to delegate opening the browser. MSAL provides Web UI implementations for most platforms, but there are still cases where may want to host the browser yourself:

- platforms not explicitly covered by MSAL, for example, Blazor, Unity, Mono on desktop
- you want to UI test your application and want to use an automated browser that can be used with Selenium
- the browser and the app running MSAL are in separate processes

#### At a glance

To achieve this, you will give to MSAL a `startUrl`, which needs to be displayed in a browser of choice so that the end user can enter their username etc. Once authentication completes, your app will need to pass back to MSAL the `endUrl`, which contains a code provided by Azure AD. The host of the `endUrl` is always the `redirectUri`. To intercept the `endUrl` you can:

- monitor browser redirects until the `redirectUrl` is hit OR
- have the browser redirect to a URL, which you monitor

#### `WithCustomWebUi` is an extensibility point

`WithCustomWebUi` is an extensibility point that allows you provide your own UI in public client applications, and to let the user go through the /Authorize endpoint of the identity provider and let them sign in and consent. MSAL.NET can, then, redeem the authentication code and get a token. It's for instance used in Visual Studio to have electrons applications (for instance VS Feedback) provide the web interaction, but leave it to MSAL.NET to do most of the work. You can also use it if you want to provide UI automation. In public client applications, MSAL.NET uses the PKCE standard ([RFC 7636 - Proof Key for Code Exchange by OAuth Public Clients](#)) to ensure that security is respected: Only MSAL.NET can redeem the code.

```
using Microsoft.Identity.Client.Extensions;
```

#### How to use `WithCustomWebUi`

In order to use `.WithCustomWebUI`, you need to:

1. Implement the `ICustomWebUi` interface (See [here](#)). You'll basically need to implement one method `AcquireAuthorizationCodeAsync` accepting the authorization code URL (computed by MSAL.NET), letting the user go through the interaction with the identity provider, and then returning back the URL by which the identity provider would have called your implementation back (including the authorization code). If you have issues, your implementation should throw a `MsalExtensionException` exception to nicely cooperate with MSAL.
2. In your `AcquireTokenInteractive` call, you can use `.WithCustomUI()` modifier passing the instance of your custom web UI

```
result = await app.AcquireTokenInteractive(scopes)
 .WithCustomWebUi(yourCustomWebUI)
 .ExecuteAsync();
```

#### Examples of implementation of `ICustomWebUi` in test automation - [Selenium WebUI](#)

The MSAL.NET team have rewritten our UI tests to leverage this extensibility mechanism. In case you're interested you can have a look at the [SeleniumWebUI](#) class in the MSAL.NET source code

#### Providing a great experience with `SystemWebViewOptions`

From MSAL.NET 4.1 `SystemWebViewOptions` enables you to specify:

- the URI to navigate to (`BrowserRedirectError`), or the HTML fragment to display (`HtmlMessageError`) in case of sign-in / consent errors in the System web browser
- the URI to navigate to (`BrowserRedirectSuccess`), or the HTML fragment to display (`HtmlMessageSuccess`) in case of successful sign-in / consent.
- the action to run to start the system browser. For this, you can provide your own implementation by setting the `OpenBrowserAsync` delegate. The class also provides a default implementation for two browsers: `OpenWithEdgeBrowserAsync` and `OpenWithChromeEdgeBrowserAsync`, respectively for Microsoft Edge and [Microsoft](#)

## Edge on Chromium.

To use this structure, you can write something like the following:

```
IPublicClientApplication app;
...

options = new SystemWebViewOptions
{
 HtmlMessageError = "Sign-in failed. You can close this tab ...",
 BrowserRedirectSuccess = "https://contoso.com/help-for-my-awesome-commandline-tool.html"
};

var result = app.AcquireTokenInteractive(scopes)
 .WithEmbeddedWebView(false) // The default in .NET Core
 .WithSystemWebViewOptions(options)
 .Build();
```

### Other optional parameters

Learn more about all the other optional parameters for `AcquireTokenInteractive` from the reference documentation for [AcquireTokenInteractiveParameterBuilder](#)

## Integrated Windows authentication

If you want to sign in a domain user on a domain or Azure AD joined machine, you need to use Integrated Windows authentication.

### Constraints

- Integrated Windows authentication (IWA) is only usable for **Federated** users only, that is, users created in an Active Directory and backed by Azure Active Directory. Users created directly in AAD, without AD backing - **managed** users - can't use this authentication flow. This limitation doesn't affect the Username/Password flow.
- IWA is for apps written for .NET Framework, .NET Core, and UWP platforms
- IWA does NOT bypass MFA (multi factor authentication). If MFA is configured, IWA might fail if an MFA challenge is required, because MFA requires user interaction.

#### NOTE

This one is tricky. IWA is non-interactive, but MFA requires user interactivity. You do not control when the identity provider requests MFA to be performed, the tenant admin does. From our observations, MFA is required when you login from a different country, when not connected via VPN to a corporate network, and sometimes even when connected via VPN. Don't expect a deterministic set of rules, Azure Active Directory uses AI to continuously learn if MFA is required. You should fallback to a user prompt (interactive authentication or device code flow) if IWA fails.

- The authority passed in the `PublicClientApplicationBuilder` needs to be:
  - tenant-ed (of the form `https://login.microsoftonline.com/{tenant}/` where `tenant` is either the guid representing the tenant ID or a domain associated with the tenant.)
  - for any work and school accounts (`https://login.microsoftonline.com/organizations/`)
  - Microsoft personal accounts are not supported (you cannot use /common or /consumers tenants)
- Because Integrated Windows Authentication is a silent flow:
  - the user of your application must have previously consented to use the application
  - or the tenant admin must have previously consented to all users in the tenant to use the application.

- In other words:
  - either you as a developer have pressed the **Grant** button on the Azure portal for yourself,
  - or a tenant admin has pressed the **Grant/revoke admin consent for {tenant domain}** button in the **API permissions** tab of the registration for the application (See [Add permissions to access web APIs](#))
  - or you've provided a way for users to consent to the application (See [Requesting individual user consent](#))
  - or you've provided a way for the tenant admin to consent for the application (See [admin consent](#))
- This flow is enabled for .net desktop, .net core, and Windows Universal (UWP) Apps.

For more information on consent, see [Microsoft identity platform permissions and consent](#)

## How to use it

- [.NET](#)
- [Java](#)
- [Python](#)
- [MacOS](#)

In MSAL.NET, you need to use

```
AcquireTokenByIntegratedWindowsAuth(IEnumerable<string> scopes)
```

You normally need only one parameter (`scopes`). However depending on the way your Windows administrator has setup the policies, it can be possible that applications on your windows machine aren't allowed to look up the logged-in user. In that case, use a second method `.WithUsername()` and pass in the username of the logged in user as a UPN format - `joe@contoso.com`. On .NET core only the overload taking the username is available, as the .NET Core platform can't ask the username to the OS.

The following sample presents the most current case, with explanations of the kind of exceptions you can get, and their mitigations

```
static async Task GetATokenForGraph()
{
 string authority = "https://login.microsoftonline.com/contoso.com";
 string[] scopes = new string[] { "user.read" };
 IPublicClientApplication app = PublicClientApplicationBuilder
 .Create(clientId)
 .WithAuthority(authority)
 .Build();

 var accounts = await app.GetAccountsAsync();

 AuthenticationResult result = null;
 if (accounts.Any())
 {
 result = await app.AcquireTokenSilent(scopes, accounts.FirstOrDefault())
 .ExecuteAsync();
 }
 else
 {
 try
 {
 result = await app.AcquireTokenByIntegratedWindowsAuth(scopes)
 .ExecuteAsync(CancellationToken.None);
 }
 catch (MsalUiRequiredException ex)
 {
 // MsalUiRequiredException: AADSTS65001: The user or administrator has not consented to use the application
 // ...
 }
 }
}
```

```

// With ID '{appid}' named '{appname}'.Send an interactive authorization request for this user and
resource.

// you need to get user consent first. This can be done, if you are not using .NET Core (which does not
have any Web UI)
// by doing (once only) an AcquireToken interactive.

// If you are using .NET core or don't want to do an AcquireTokenInteractive, you might want to suggest the
user to navigate
// to a URL to consent: https://login.microsoftonline.com/common/oauth2/v2.0/authorize?client_id=
{clientId}&response_type=code&scope=user.read

// AADSTS50079: The user is required to use multi-factor authentication.
// There is no mitigation - if MFA is configured for your tenant and AAD decides to enforce it,
// you need to fallback to an interactive flows such as AcquireTokenInteractive or AcquireTokenByDeviceCode
}
catch (MsalServiceException ex)
{
 // Kind of errors you could have (in ex.Message)

 // MsalServiceException: AADSTS90010: The grant type is not supported over the /common or /consumers
endpoints. Please use the /organizations or tenant-specific endpoint.
 // you used common.
 // Mitigation: as explained in the message from Azure AD, the authority needs to be tenanted or otherwise
organizations

 // MsalServiceException: AADSTS70002: The request body must contain the following parameter:
'client_secret or client_assertion'.
 // Explanation: this can happen if your application was not registered as a public client application in
Azure AD
 // Mitigation: in the Azure portal, edit the manifest for your application and set the `allowPublicClient`
to `true`
}
catch (MsalClientException ex)
{
 // Error Code: unknown_user Message: Could not identify logged in user
 // Explanation: the library was unable to query the current Windows logged-in user or this user is not
AD or AAD
 // joined (work-place joined users are not supported).

 // Mitigation 1: on UWP, check that the application has the following capabilities: Enterprise
Authentication,
 // Private Networks (Client and Server), User Account Information

 // Mitigation 2: Implement your own logic to fetch the username (e.g. john@contoso.com) and use the
 // AcquireTokenByIntegratedWindowsAuth form that takes in the username

 // Error Code: integrated_windows_auth_not_supported_managed_user
 // Explanation: This method relies on a protocol exposed by Active Directory (AD). If a user was
created in Azure
 // Active Directory without AD backing ("managed" user), this method will fail. Users created in AD and
backed by
 // AAD ("federated" users) can benefit from this non-interactive method of authentication.
 // Mitigation: Use interactive authentication
 }
}

Console.WriteLine(result.Account.Username);
}

```

For the list of possible modifiers on AcquireTokenByIntegratedWindowsAuthentication, see  
[AcquireTokenByIntegratedWindowsAuthParameterBuilder](#)

## Username / Password

You can also acquire a token by providing the username and password. This flow is limited and not recommended,

but there are still use cases where it's necessary.

### This flow isn't recommended

This flow is **not recommended** because your application asking a user for their password isn't secure. For more information about this problem, see [this article](#). The preferred flow for acquiring a token silently on Windows domain joined machines is [Integrated Windows Authentication](#). Otherwise you can also use [Device code flow](#)

#### NOTE

Although this is useful in some cases (DevOps scenarios), if you want to use Username/password in interactive scenarios where you provide your own UI, you should really think about how to move away from it. By using username/password you are giving-up a number of things:

- core tenets of modern identity: password gets fished, replayed. Because we have this concept of a share secret that can be intercepted. This is incompatible with passwordless.
- users who need to do MFA won't be able to sign-in (as there is no interaction)
- Users won't be able to do single sign-on

### Constraints

The following constraints also apply:

- The Username/Password flow isn't compatible with Conditional Access and multi-factor authentication: As a consequence, if your app runs in an Azure AD tenant where the tenant admin requires multi-factor authentication, you can't use this flow. Many organizations do that.
- It works only for Work and school accounts (not MSA)
- The flow is available on .net desktop and .net core, but not on UWP.

### B2C specifics

[More information on using ROPC with B2C.](#)

### How to use it?

- [.NET](#)
- [Java](#)
- [Python](#)
- [MacOS](#)

`IPublicClientApplication` contains the method `AcquireTokenByUsernamePassword`

The following sample presents a simplified case

```

static async Task GetATokenForGraph()
{
 string authority = "https://login.microsoftonline.com/contoso.com";
 string[] scopes = new string[] { "user.read" };
 IPublicClientApplication app;
 app = PublicClientApplicationBuilder.Create(clientId)
 .WithAuthority(authority)
 .Build();
 var accounts = await app.GetAccountsAsync();

 AuthenticationResult result = null;
 if (accounts.Any())
 {
 result = await app.AcquireTokenSilent(scopes, accounts.FirstOrDefault())
 .ExecuteAsync();
 }
 else
 {
 try
 {
 var securePassword = new SecureString();
 foreach (char c in "dummy") // you should fetch the password
 securePassword.AppendChar(c); // keystroke by keystroke

 result = await app.AcquireTokenByUsernamePassword(scopes,
 "joe@contoso.com",
 securePassword)
 .ExecuteAsync();
 }
 catch(MsalException)
 {
 // See details below
 }
 }
 Console.WriteLine(result.Account.Username);
}

```

The following sample presents the most current case, with explanations of the kind of exceptions you can get, and their mitigations

```

static async Task GetATokenForGraph()
{
 string authority = "https://login.microsoftonline.com/contoso.com";
 string[] scopes = new string[] { "user.read" };
 IPublicClientApplication app;
 app = PublicClientApplicationBuilder.Create(clientId)
 .WithAuthority(authority)
 .Build();
 var accounts = await app.GetAccountsAsync();

 AuthenticationResult result = null;
 if (accounts.Any())
 {
 result = await app.AcquireTokenSilent(scopes, accounts.FirstOrDefault())
 .ExecuteAsync();
 }
 else
 {
 try
 {
 var securePassword = new SecureString();
 foreach (char c in "dummy") // you should fetch the password keystroke
 securePassword.AppendChar(c); // by keystroke

 result = await app.AcquireTokenByUsernamePassword(scopes,
 "joe@contoso.com".

```

```

 ,

 securePassword)

 .ExecuteAsync();

 }

 catch (MsalUiRequiredException ex) when (ex.Message.Contains("AADSTS65001"))

 {

 // Here are the kind of error messages you could have, and possible mitigations

 // -----

 // MsalUiRequiredException: AADSTS65001: The user or administrator has not consented to use the application

 // with ID '{appId}' named '{appName}'. Send an interactive authorization request for this user and

 resource.

 // Mitigation: you need to get user consent first. This can be done either statically (through the portal),

 /// or dynamically (but this requires an interaction with Azure AD, which is not possible with

 // the username/password flow)

 // Statically: in the portal by doing the following in the "API permissions" tab of the application

 registration:

 // 1. Click "Add a permission" and add all the delegated permissions corresponding to the scopes you want

 (for instance

 // User.Read and User.ReadBasic.All)

 // 2. Click "Grant/revoke admin consent for <tenant>" and click "yes".

 // Dynamically, if you are not using .NET Core (which does not have any Web UI) by

 // calling (once only) AcquireTokenInteractive.

 // remember that Username/password is for public client applications that is desktop/mobile applications.

 // If you are using .NET core or don't want to call AcquireTokenInteractive, you might want to:

 // - use device code flow (See https://aka.ms/msal-net-device-code-flow)

 // - or suggest the user to navigate to a URL to consent:

https://login.microsoftonline.com/common/oauth2/v2.0/authorize?client_id={clientId}&response_type=code&scope=user.read

 // -----

 // -----

 // ErrorCode: invalid_grant

 // SubError: basic_action

 // MsalUiRequiredException: AADSTS50079: The user is required to use multi-factor authentication.

 // The tenant admin for your organization has chosen to oblige users to perform multi-factor

 authentication.

 // Mitigation: none for this flow

 // Your application cannot use the Username/Password grant.

 // Like in the previous case, you might want to use an interactive flow (AcquireTokenInteractive()),

 // or Device Code Flow instead.

 // Note this is one of the reason why using username/password is not recommended;

 // -----

 // -----

 // ex.ErrorCode: invalid_grant

 // subError: null

 // Message = "AADSTS70002: Error validating credentials."

 // AADSTS50126: Invalid username or password

 // In the case of a managed user (user from an Azure AD tenant opposed to a

 // federated user, which would be owned

 // in another IdP through ADFS), the user has entered the wrong password

 // Mitigation: ask the user to re-enter the password

 // -----

 // -----

 // ex.ErrorCode: invalid_grant

 // subError: null

 // MsalServiceException: ADSTS50034: To sign into this application the account must be added to

 // the {domainName} directory.

 // or The user account does not exist in the {domainName} directory. To sign into this application,

 // the account must be added to the directory.

 // The user was not found in the directory

 // Explanation: wrong username

 // Mitigation: ask the user to re-enter the username.

 // -----

 }

 catch (MsalServiceException ex) when (ex.ErrorCode == "invalid_request")

 {

```

```

}

// -----
// AADSTS90010: The grant type is not supported over the /common or /consumers endpoints.
// Please use the /organizations or tenant-specific endpoint.
// you used common.
// Mitigation: as explained in the message from Azure AD, the authority you use in the application needs
// to be tenanted or otherwise "organizations". change the
// "Tenant": property in the appsettings.json to be a GUID (tenant Id), or domain name (contoso.com)
// if such a domain is registered with your tenant
// or "organizations", if you want this application to sign-in users in any Work and School accounts.
// -----

}

catch (MsalServiceException ex) when (ex.ErrorCode == "unauthorized_client")
{
// -----
// AADSTS700016: Application with identifier '{clientId}' was not found in the directory '{domain}'.
// 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 may have sent your authentication request to the wrong tenant
// Cause: The clientId in the appsettings.json might be wrong
// Mitigation: check the clientId and the app registration
// -----
}

catch (MsalServiceException ex) when (ex.ErrorCode == "invalid_client")
{
// -----
// AADSTS70002: The request body must contain the following parameter: 'client_secret or client_assertion'.
// Explanation: this can happen if your application was not registered as a public client application in
Azure AD
// Mitigation: in the Azure portal, edit the manifest for your application and set the `allowPublicClient`
to `true`
// -----
}

catch (MsalServiceException)
{
throw;
}

catch (MsalClientException ex) when (ex.ErrorCode == "unknown_user_type")
{
// Message = "Unsupported User Type 'Unknown'. Please see https://aka.ms/msal-net-up"
// The user is not recognized as a managed user, or a federated user. Azure AD was not
// able to identify the IdP that needs to process the user
throw new ArgumentException("U/P: Wrong username", ex);
}

catch (MsalClientException ex) when (ex.ErrorCode == "user_realm_discovery_failed")
{
// The user is not recognized as a managed user, or a federated user. Azure AD was not
// able to identify the IdP that needs to process the user. That's for instance the case
// if you use a phone number
throw new ArgumentException("U/P: Wrong username", ex);
}

catch (MsalClientException ex) when (ex.ErrorCode == "unknown_user")
{
// the username was probably empty
// ex.Message = "Could not identify the user logged into the OS. See https://aka.ms/msal-net-iwa for
details."
throw new ArgumentException("U/P: Wrong username", ex);
}

catch (MsalClientException ex) when (ex.ErrorCode == "parsing_wstrust_response_failed")
{
// -----
// In the case of a Federated user (that is owned by a federated IdP, as opposed to a managed user owned in
an Azure AD tenant)
// ID3242: The security token could not be authenticated or authorized.
// The user does not exist or has entered the wrong password
// -----
}

```

```
 }

 Console.WriteLine(result.Account.Username);
 }
}
```

For details on all the modifiers that can be applied to `AcquireTokenByUsernamePassword`, see [AcquireTokenByUsernamePasswordParameterBuilder](#)

## Command-line tool (without web browser)

### Device code flow

If you're writing a command-line tool (that doesn't have Web controls), and can't or don't want to use the previous flows, you'll need to use the Device code flow.

Interactive authentication with Azure AD requires a web browser (for details see [Usage of web browsers](#)).

However, to authenticate users on devices or operating systems that don't provide a Web browser, Device code flow lets the user use another device (for instance another computer or a mobile phone) to sign in interactively. By using the device code flow, the application obtains tokens through a two-step process especially designed for these devices/OSes. Examples of such applications are applications running on IoT, or Command-Line tools (CLI). The idea is that:

1. Whenever user authentication is required, the app provides a code and asks the user to use another device (such as an internet-connected smartphone) to navigate to a URL (for instance, <https://microsoft.com/devicelogin>), where the user will be prompted to enter the code. That done, the web page will lead the user through a normal authentication experience, including consent prompts and multi-factor authentication if necessary.
2. Upon successful authentication, the command-line app will receive the required tokens through a back channel and will use it to perform the web API calls it needs.

### How to use?

- .NET
- Java
- Python
- MacOS

`IPublicClientApplication` contains a method named `AcquireTokenWithDeviceCode`

```
AcquireTokenWithDeviceCode(IEnumerable<string> scopes,
 Func<DeviceCodeResult, Task> deviceCodeResultCallback)
```

This method takes as parameters:

- The `scopes` to request an access token for
- A callback that will receive the `DeviceCodeResult`

## DeviceCodeResult

Class



### Properties

- 🔧 ClientId { get; } : string
- 🔧 DeviceCode { get; } : string
- 🔧 ExpiresOn { get; } : DateTimeOffset
- 🔧 Interval { get; } : long
- 🔧 Message { get; } : string
- 🔧 Scopes { get; } : IReadOnlyCollection<string>
- 🔧 UserCode { get; } : string
- 🔧 VerificationUrl { get; } : string

The following sample code presents the most current case, with explanations of the kind of exceptions you can get, and their mitigation.

```
private const string ClientId = "<client_guid>";
private const string Authority = "https://login.microsoftonline.com/contoso.com";
private readonly string[] Scopes = new string[] { "user.read" };

static async Task<AuthenticationResult> GetATokenForGraph()
{
 IPublicClientApplication pca = PublicClientApplicationBuilder
 .Create(ClientId)
 .WithAuthority(Authority)
 .WithDefaultRedirectUri()
 .Build();

 var accounts = await pca.GetAccountsAsync();

 // All AcquireToken* methods store the tokens in the cache, so check the cache first
 try
 {
 return await pca.AcquireTokenSilent(Scopes, accounts.FirstOrDefault())
 .ExecuteAsync();
 }
 catch (MsalUiRequiredException ex)
 {
 // No token found in the cache or AAD insists that a form interactive auth is required (e.g. the
 // tenant admin turned on MFA)
 // If you want to provide a more complex user experience, check out ex.Classification

 return await AcquireByDeviceCodeAsync(pca);
 }
}

private async Task<AuthenticationResult> AcquireByDeviceCodeAsync(IPublicClientApplication pca)
{
 try
 {
 var result = await pca.AcquireTokenWithDeviceCode(scopes,
 deviceCodeResult =>
 {
 // This will print the message on the console which tells the user where to go sign-in
 using
 // a separate browser and the code to enter once they sign in.
 // The AcquireTokenWithDeviceCode() method will poll the server after firing this
 // device code callback to look for the successful login of the user via that browser.
 // This background polling (whose interval and timeout data is also provided as fields in

```

```

the
 // deviceCodeCallback class) will occur until:
 // * The user has successfully logged in via browser and entered the proper code
 // * The timeout specified by the server for the lifetime of this code (typically ~15
minutes) has been reached
 // * The developing application calls the Cancel() method on a CancellationToken sent into
the method.
 // If this occurs, an OperationCanceledException will be thrown (see catch below for
more details).
 Console.WriteLine(deviceCodeResult.Message);
 return Task.FromResult(0);
 }).ExecuteAsync();
}

Console.WriteLine(result.Account.Username);
return result;
}
// TODO: handle or throw all these exceptions depending on your app
catch (MsalServiceException ex)
{
 // Kind of errors you could have (in ex.Message)

 // AADSTS50059: No tenant-identifying information found in either the request or implied by any
provided credentials.
 // Mitigation: as explained in the message from Azure AD, the authority needs to be tenanted. you have
probably created
 // your public client application with the following authorities:
 // https://login.microsoftonline.com/common or https://login.microsoftonline.com/organizations

 // AADSTS90133: Device Code flow is not supported under /common or /consumers endpoint.
 // Mitigation: as explained in the message from Azure AD, the authority needs to be tenanted

 // AADSTS90002: Tenant <tenantId or domain you used in the authority> not found. This may happen if
there are
 // no active subscriptions for the tenant. Check with your subscription administrator.
 // Mitigation: if you have an active subscription for the tenant this might be that you have a typo in
the
 // tenantId (GUID) or tenant domain name.
 }
 catch (OperationCanceledException ex)
 {
 // If you use a CancellationToken, and call the Cancel() method on it, then this *may* be triggered
 // to indicate that the operation was cancelled.
 // See https://docs.microsoft.com/dotnet/standard/threading/cancellation-in-managed-threads
 // for more detailed information on how C# supports cancellation in managed threads.
 }
 catch (MsalClientException ex)
 {
 // Possible cause - verification code expired before contacting the server
 // This exception will occur if the user does not manage to sign-in before a time out (15 mins) and
the
 // call to `AcquireTokenWithDeviceCode` is not cancelled in between
 }
}
}

```

## File based token cache

In MSAL.NET, an in-memory token cache is provided by default.

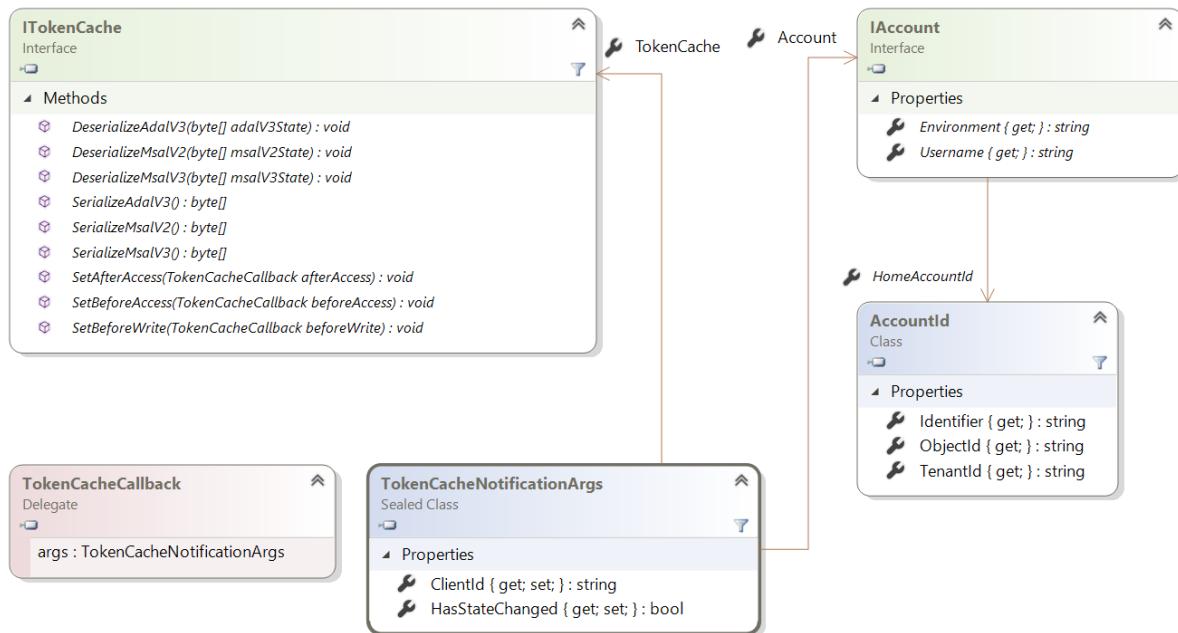
### **Serialization is customizable in Windows desktop apps and web apps/web APIs**

In the case of .NET Framework and .NET core, if you don't do anything extra, the in-memory token cache lasts for the duration of the application. To understand why serialization isn't provided out of the box, remember MSAL .NET desktop/core applications can be console or Windows applications (which would have access to the file system), **but also** Web applications or web API. These Web apps and web APIs might use some specific cache mechanisms like databases, distributed caches, redis caches and so on. To have a persistent token cache application

in .NET Desktop or Core, you'll need to customize the serialization.

Classes and interfaces involved in token cache serialization are the following types:

- `ITokenCache`, which defines events to subscribe to token cache serialization requests, as well as methods to serialize or de-serialize the cache at various formats (ADAL v3.0, MSAL 2.x, and MSAL 3.x = ADAL v5.0)
- `TokenCacheCallback` is a callback passed to the events so that you can handle the serialization. they'll be called with arguments of type `TokenCacheNotificationArgs`.
- `TokenCacheNotificationArgs` only provides the `ClientId` of the application and a reference to the user for which the token is available



### IMPORTANT

MSAL.NET creates token caches for you and provides you with the `IToken` cache when you call an application's `UserTokenCache` and `AppTokenCache` properties. You aren't supposed to implement the interface yourself. Your responsibility, when you implement a custom token cache serialization, is to:

- React to `BeforeAccess` and `AfterAccess` "events" (or their `Async` counterparts). The `BeforeAccess` delegate is responsible to deserialize the cache, whereas the `AfterAccess` one is responsible for serializing the cache.
- Part of these events store or load blobs, which are passed through the event argument to whatever storage you want.

The strategies are different depending on if you're writing a token cache serialization for a public client application (Desktop), or a confidential client application (web app/web API, daemon app).

Since MSAL V2.x you have several options, depending on if you want to serialize the cache only to the MSAL.NET format (unified format cache that is common with MSAL, but also across the platforms), or if you also want to support the [legacy](#) Token cache serialization of ADAL V3.

The customization of Token cache serialization to share the SSO state between ADAL.NET 3.x, ADAL.NET 5.x, and MSAL.NET is explained in part of the following sample: [active-directory-dotnet-v1-to-v2](#)

### Simple token cache serialization (MSAL only)

Below is an example of a naive implementation of custom serialization of a token cache for desktop applications. Here the user token cache in a file in the same folder as the application.

After you build the application, you enable the serialization by calling `TokenCacheHelper.EnableSerialization()`

passing the application `UserTokenCache`

```
app = PublicClientApplicationBuilder.Create(ClientId)
 .Build();
TokenCacheHelper.EnableSerialization(app.UserTokenCache);
```

This helper class looks like the following code snippet:

```
static class TokenCacheHelper
{
 public static void EnableSerialization(ITokenCache tokenCache)
 {
 tokenCache.SetBeforeAccess(BeforeAccessNotification);
 tokenCache.SetAfterAccess(AfterAccessNotification);
 }

 ///<summary>
 /// Path to the token cache
 ///</summary>
 public static readonly string CacheFilePath = System.Reflection.Assembly.GetExecutingAssembly().Location +
".msalcache.bin";

 private static readonly object FileLock = new object();

 private static void BeforeAccessNotification(TokenCacheNotificationArgs args)
 {
 lock (FileLock)
 {
 args.TokenCache.DeserializeMsalV3(File.Exists(CacheFilePath)
 ? ProtectedData.Unprotect(File.ReadAllBytes(CacheFilePath),
 null,
 DataProtectionScope.CurrentUser)
 : null);
 }
 }

 private static void AfterAccessNotification(TokenCacheNotificationArgs args)
 {
 // if the access operation resulted in a cache update
 if (args.HasStateChanged)
 {
 lock (FileLock)
 {
 // reflect changes in the persistent store
 File.WriteAllBytes(CacheFilePath,
 ProtectedData.Protect(args.TokenCache.SerializeMsalV3(),
 null,
 DataProtectionScope.CurrentUser));
 }
 }
 }
}
```

A preview of a product quality token cache file-based serializer for public client applications (for desktop applications running on Windows, Mac and linux) is available from the [Microsoft.Identity.Client.Extensions.Msal](#) open-source library. You can include it in your applications from the following nuget package:

[Microsoft.Identity.Client.Extensions.Msal](#).

## NOTE

Disclaimer. The Microsoft.Identity.Client.Extensions.Msal library is an extension over MSAL.NET. Classes in these libraries might make their way into MSAL.NET in the future, as is or with breaking changes.

## Dual token cache serialization (MSAL unified cache + ADAL V3)

If you want to implement token cache serialization both with the Unified cache format (common to ADAL.NET 4.x and MSAL.NET 2.x, and with other MSALs of the same generation or older, on the same platform), you can get inspired by the following code:

```
string appLocation = Path.GetDirectoryName(Assembly.GetEntryAssembly().Location);
string cacheFolder = Path.GetFullPath(appLocation) + @"..\..\..\..";
string adalV3cacheFileName = Path.Combine(cacheFolder, "cacheAdalV3.bin");
string unifiedCacheFileName = Path.Combine(cacheFolder, "unifiedCache.bin");

IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
 .Build();
FilesBasedTokenCacheHelper.EnableSerialization(app.UserTokenCache,
 unifiedCacheFileName,
 adalV3cacheFileName);
```

This time the helper class looks like the following code:

```
using System;
using System.IO;
using System.Security.Cryptography;
using Microsoft.Identity.Client;

namespace CommonCacheMsalV3
{
 /// <summary>
 /// Simple persistent cache implementation of the dual cache serialization (ADAL V3 legacy
 /// and unified cache format) for a desktop applications (from MSAL 2.x)
 /// </summary>
 static class FilesBasedTokenCacheHelper
 {
 /// <summary>
 /// Get the user token cache
 /// </summary>
 /// <param name="adalV3CacheFileName">File name where the cache is serialized with the
 /// ADAL V3 token cache format. Can
 /// be <c>null</c> if you don't want to implement the legacy ADAL V3 token cache
 /// serialization in your MSAL 2.x+ application</param>
 /// <param name="unifiedCacheFileName">File name where the cache is serialized
 /// with the Unified cache format, common to
 /// ADAL V4 and MSAL V2 and above, and also across ADAL/MSAL on the same platform.
 /// Should not be <c>null</c></param>
 /// <returns></returns>
 public static void EnableSerialization(ITokenCache cache, string unifiedCacheFileName, string
adalV3CacheFileName)
 {
 UnifiedCacheFileName = unifiedCacheFileName;
 AdalV3CacheFileName = adalV3CacheFileName;

 cache.SetBeforeAccess(BeforeAccessNotification);
 cache.SetAfterAccess(AfterAccessNotification);
 }

 /// <summary>
 /// File path where the token cache is serialized with the unified cache format
 /// (ADAL.NET V4 / MSAL.NET V2)
 }
}
```

```

/// <summary>
/// </summary>
public static string UnifiedCacheFileName { get; private set; }

/// <summary>
/// File path where the token cache is serialized with the legacy ADAL V3 format
/// </summary>
public static string AdalV3CacheFileName { get; private set; }

private static readonly object FileLock = new object();

public static void BeforeAccessNotification(TokenCacheNotificationArgs args)
{
 lock (FileLock)
 {
 args.TokenCache.DeserializeAdalV3(ReadFromFileIfExists(AdalV3CacheFileName));
 try
 {
 args.TokenCache.DeserializeMsalV3(ReadFromFileIfExists(UnifiedCacheFileName));
 }
 catch(Exception ex)
 {
 // Compatibility with the MSAL v2 cache if you used one
 args.TokenCache.DeserializeMsalV2(ReadFromFileIfExists(UnifiedCacheFileName));
 }
 }
}

public static void AfterAccessNotification(TokenCacheNotificationArgs args)
{
 // if the access operation resulted in a cache update
 if (args.HasStateChanged)
 {
 lock (FileLock)
 {
 WriteToFileIfNotNull(UnifiedCacheFileName, args.TokenCache.SerializeMsalV3());
 if (!string.IsNullOrWhiteSpace(AdalV3CacheFileName))
 {
 WriteToFileIfNotNull(AdalV3CacheFileName, args.TokenCache.SerializeAdalV3());
 }
 }
 }
}

/// <summary>
/// Read the content of a file if it exists
/// </summary>
/// <param name="path">File path</param>
/// <returns>Content of the file (in bytes)</returns>
private static byte[] ReadFromFileIfExists(string path)
{
 byte[] protectedBytes = (!string.IsNullOrEmpty(path) && File.Exists(path))
 ? File.ReadAllBytes(path) : null;
 byte[] unprotectedBytes = encrypt ?
 ((protectedBytes != null) ? ProtectedData.Unprotect(protectedBytes, null,
 DataProtectionScope.CurrentUser) : null)
 : protectedBytes;
 return unprotectedBytes;
}

/// <summary>
/// Writes a blob of bytes to a file. If the blob is <c>null</c>, deletes the file
/// </summary>
/// <param name="path">path to the file to write</param>
/// <param name="blob">Blob of bytes to write</param>
private static void WriteToFileIfNotNull(string path, byte[] blob)
{
 if (blob != null)
 {

```

```
byte[] protectedBytes = encrypt
 ? ProtectedData.Protect(blob, null, DataProtectionScope.CurrentUser)
 : blob;
File.WriteAllBytes(path, protectedBytes);
}
else
{
 File.Delete(path);
}
}

// Change if you want to test with an un-encrypted blob (this is a json format)
private static bool encrypt = true;
}
}
```

## Next steps

[Calling a web API from the desktop app](#)

# Desktop app that calls web APIs - call a web API

10/30/2019 • 4 minutes to read • [Edit Online](#)

Now that you have a token, you can call a protected web API.

## Calling a web API

- [.NET](#)
- [Python](#)
- [Java](#)
- [MacOS](#)

### AuthenticationResult properties in MSAL.NET

The methods to acquire tokens return an `AuthenticationResult` (or, for the async methods, a `Task<AuthenticationResult>`).

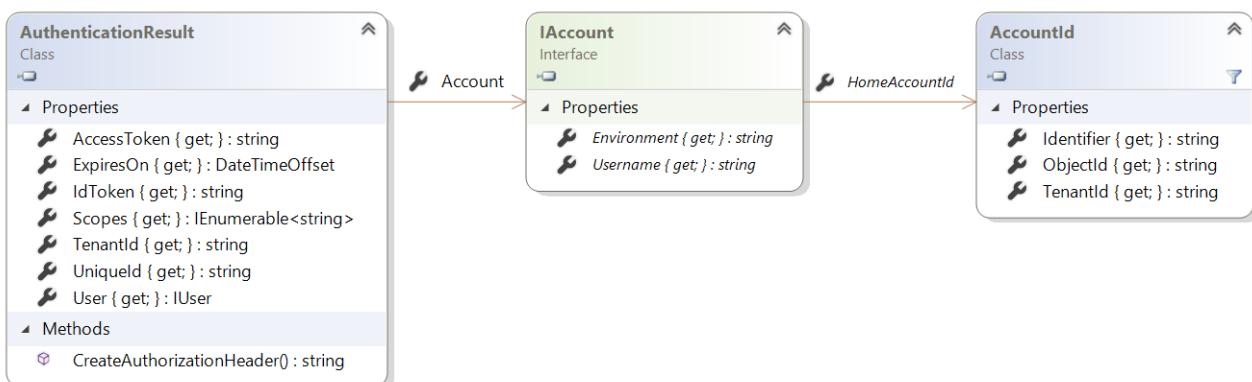
In MSAL.NET, `AuthenticationResult` exposes:

- `AccessToken` for the Web API to access resources. This parameter is a string, usually a base64 encoded JWT but the client should never look inside the access token. The format isn't guaranteed to remain stable and it can be encrypted for the resource. People writing code depending on access token content on the client is one of the biggest sources of errors and client logic breaks. See also [Access tokens](#)
- `IdToken` for the user (this parameter is an encoded JWT). See [ID Tokens](#)
- `ExpiresOn` tells the date/time when the token expires
- `TenantId` contains the tenant in which the user was found. For guest users (Azure AD B2B scenarios), the Tenant ID is the guest tenant, not the unique tenant. When the token is delivered for a user, `AuthenticationResult` also contains information about this user. For confidential client flows where tokens are requested with no user (for the application), this User information is null.
- The `Scopes` for which the token was issued.
- The unique Id for the user.

## IAccount

MSAL.NET defines the notion of Account (through the `IAccount` interface). This breaking change provides the right semantics: the fact that the same user can have several accounts, in different Azure AD directories. Also MSAL.NET provides better information in the case of guest scenarios, as home account information is provided.

The following diagram shows the structure of the `IAccount` interface:



The `AccountId` class identifies an account in a specific tenant. It has the following properties:

PROPERTY	DESCRIPTION
TenantId	A string representation for a GUID, which is the ID of the tenant where the account resides.
ObjectId	A string representation for a GUID, which is the ID of the user who owns the account in the tenant.
Identifier	Unique identifier for the account. Identifier is the concatenation of ObjectId and TenantId separated by a comma and are not base64 encoded.

The IAccount interface represents information about a single account. The same user can be present in different tenants, that is, a user can have multiple accounts. Its members are:

PROPERTY	DESCRIPTION
Username	A string containing the displayable value in UserPrincipalName (UPN) format, for example, john.doe@contoso.com. This string can be null, whereas the HomeAccountId and HomeAccountId.Identifier won't be null. This property replaces the DisplayableId property of IUser in previous versions of MSAL.NET.
Environment	A string containing the identity provider for this account, for example, login.microsoftonline.com. This property replaces the IdentityProvider property of IUser, except that IdentityProvider also had information about the tenant (in addition to the cloud environment), whereas here the value is only the host.
HomeAccountId	AccountId of the home account for the user. This property uniquely identifies the user across Azure AD tenants.

## Using the token to call a protected API

Once the AuthenticationResult has been returned by MSAL (in result), you need to add it to the HTTP authorization header, before making the call to access the protected Web API.

```
httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", result.AccessToken);

// Call Web API.
HttpResponseMessage response = await _httpClient.GetAsync(apiUri);
...
}
```

## Next steps

[Move to production](#)

# Desktop app that calls web APIs - move to production

10/30/2019 • 2 minutes to read • [Edit Online](#)

This article provides you details to improve your application further and move it to production.

## Handling errors in desktop applications

In the different flows, you've learned how to handle the errors for the silent flows (as shown in code snippets). You've also seen that there are cases where interaction is needed (incremental consent and Conditional Access).

## How to have the user consent upfront for several resources

### NOTE

Getting consent for several resources works for Microsoft identity platform, but not for Azure Active Directory (Azure AD) B2C. Azure AD B2C supports only admin consent, not user consent.

The Microsoft identity platform (v2.0) endpoint doesn't allow you to get a token for several resources at once. Therefore, the `scopes` parameter can only contain scopes for a single resource. You can ensure that the user pre-consents to several resources by using the `extraScopesToConsent` parameter.

For instance, if you have two resources, which have two scopes each:

- `https://mytenant.onmicrosoft.com/customerapi` - with 2 scopes `customer.read` and `customer.write`
- `https://mytenant.onmicrosoft.com/vendorapi` - with 2 scopes `vendor.read` and `vendor.write`

You should use the `.WithAdditionalPromptToConsent` modifier that has the `extraScopesToConsent` parameter.

For instance:

### In MSAL.NET

```
string[] scopesForCustomerApi = new string[]
{
 "https://mytenant.onmicrosoft.com/customerapi/customer.read",
 "https://mytenant.onmicrosoft.com/customerapi/customer.write"
};

string[] scopesForVendorApi = new string[]
{
 "https://mytenant.onmicrosoft.com/vendorapi/vendor.read",
 "https://mytenant.onmicrosoft.com/vendorapi/vendor.write"
};

var accounts = await app.GetAccountsAsync();
var result = await app.AcquireTokenInteractive(scopesForCustomerApi)
 .WithAccount(accounts.FirstOrDefault())
 .WithExtraScopeToConsent(scopesForVendorApi)
 .ExecuteAsync();
```

### In MSAL for iOS and macOS

Objective-C:

```

NSArray *scopesForCustomerApi = @[@"https://mytenant.onmicrosoft.com/customerapi/customer.read",
 @"https://mytenant.onmicrosoft.com/customerapi/customer.write"];

NSArray *scopesForVendorApi = @[@"https://mytenant.onmicrosoft.com/vendorapi/vendor.read",
 @"https://mytenant.onmicrosoft.com/vendorapi/vendor.write"]

MSALInteractiveTokenParameters *interactiveParams = [[MSALInteractiveTokenParameters alloc]
initWithScopes:scopesForCustomerApi webViewParameters:[MSALWebviewParameters new]];
interactiveParams.extraScopesToConsent = scopesForVendorApi;
[application acquireTokenWithParameters:interactiveParams completionBlock:^(MSALResult *result, NSError *error) { /* handle result */ }];

```

Swift:

```

let scopesForCustomerApi = ["https://mytenant.onmicrosoft.com/customerapi/customer.read",
 "https://mytenant.onmicrosoft.com/customerapi/customer.write"]

let scopesForVendorApi = ["https://mytenant.onmicrosoft.com/vendorapi/vendor.read",
 "https://mytenant.onmicrosoft.com/vendorapi/vendor.write"]

let interactiveParameters = MSALInteractiveTokenParameters(scopes: scopesForCustomerApi, webViewParameters:
MSALWebviewParameters())
interactiveParameters.extraScopesToConsent = scopesForVendorApi
application.acquireToken(with: interactiveParameters, completionBlock: { (result, error) in /* handle result */
*/ })

```

This call will get you an access token for the first web API.

When you need to call the second web API, you can call `AcquireTokenSilent` API:

```
AcquireTokenSilent(scopesForVendorApi, accounts.FirstOrDefault()).ExecuteAsync();
```

### **Microsoft personal account requires reconsenting each time the app is run**

For Microsoft personal accounts users, reprompting for consent on each native client (desktop/mobile app) call to authorize is the intended behavior. Native client identity is inherently insecure (contrary to confidential client application which exchange a secret with the Microsoft Identity platform to prove their identity). The Microsoft identity platform chose to mitigate this insecurity for consumer services by prompting the user for consent, each time the application is authorized.

## Next steps

Make your application great:

- Enable [logging](#).
- Enable telemetry.
- Enable [proxies and customize HTTP clients](#).

Test your integration:

- Use the [Microsoft identity platform integration checklist](#).

# Scenario: Daemon application that calls web APIs

9/17/2019 • 2 minutes to read • [Edit Online](#)

Learn all you need to build a daemon application that calls web APIs.

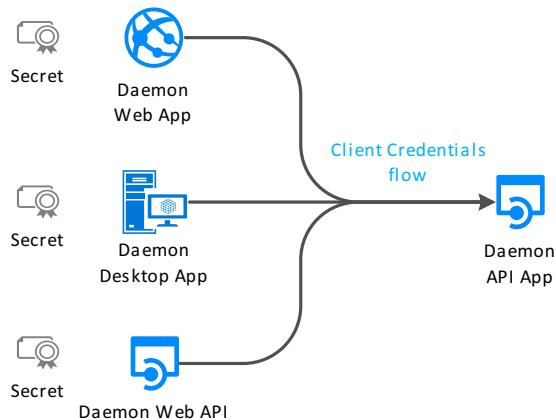
## Prerequisites

Before reading this article, you should be familiar with the following concepts or read the following articles:

- [Microsoft identity platform overview](#)
- [Authentication basics](#)
- [Audiences](#)
- [Application and service principals](#)
- [Permissions and consent](#)
- [ID tokens and access tokens](#)

## Overview

Your application can acquire a token to call a web API on behalf of itself (not on behalf of a user). This scenario is useful for daemon applications. It's using the standard OAuth 2.0 [client credentials](#) grant.



Here are some examples of use cases for daemon apps:

- Web applications that are used to provision or administer users, or do batch processes in a directory
- Desktop applications (such as windows services on Windows, or daemons processes on Linux) that perform batch jobs, or an operating system service running in the background
- Web APIs that need to manipulate directories, not specific users

There's another common case where non-daemon applications use client credentials: even when they act on behalf of users, they need to access a web API or a resource with their identity for technical reasons. An example is access to secrets in KeyVault or an Azure SQL database for a cache.

Applications that acquire a token for their own identities:

- Are confidential client applications. These apps, given that they access resources independently of a user, need to prove their identity. They're also rather sensitive apps, which they need to be approved by the Azure Active Directory (Azure AD) tenant admins.
- Have registered a secret (application password or certificate) with Azure AD. This secret is passed-in during the call to Azure AD to get a token.

# Specifics

## IMPORTANT

- User interaction isn't possible with a daemon application. A daemon application requires its own identity. This type of application requests an access token by using its application identity and presenting its application ID, credential (password or certificate), and application ID URI to Azure AD. After successful authentication, the daemon receives an access token (and a refresh token) from the Microsoft identity platform endpoint, which is then used to call the web API (and is refreshed as needed).
- Because user interaction is not possible, incremental consent won't be possible. All the required API permissions need to be configured at application registration, and the code of the application just requests statically defined permissions. This also means that daemon applications won't support incremental consent.

For developers, the end-to-end experience for this scenario has the following aspects:

- Daemon applications can only work in Azure AD tenants. It wouldn't make sense to build a daemon application that attempts to manipulate Microsoft personal accounts. If you're a line-of-business (LOB) app developer, you'll create your daemon app in your tenant. If you're an ISV, you might want to create a multi-tenant daemon application. It will need to be consented by each tenant admin.
- During the [Application registration](#), the **Reply URI** isn't needed. You need to share secrets or certificates or signed assertions with Azure AD, and you need to request applications permissions and grant admin consent to use those app permissions.
- The [Application configuration](#) needs to provide client credentials as shared with Azure AD during the application registration.
- The [scope](#) used to acquire a token with the client credentials flow needs to be a static scope.

## Next steps

[Daemon app - app registration](#)

# Daemon app that calls web APIs - app registration

10/30/2019 • 2 minutes to read • [Edit Online](#)

For a daemon application, here's what you need to know when registering the app.

## Supported account types

Given that daemon applications only make sense in Azure AD tenants, when you create the application you'll need to choose:

- either **Accounts in this organizational directory only**. This choice is the most common case, as daemon applications are usually written by line-of-business (LOB) developers.
- or **Accounts in any organizational directory**. You'll make this choice if you're an ISV providing a utility tool to your customers. You'll need customer's tenants admins to approve it.

## Authentication - no Reply URI needed

In the case where your confidential client application uses **only** the client credentials flow, the Reply URI doesn't need to be registered. It's not needed either for the application configuration/construction. The client credentials flow doesn't use it.

## API Permissions - app permissions and admin consent

A daemon application can only request application permissions to APIs (not delegated permissions). In the **API Permission** page for the application registration, after you've selected **Add a permission** and chosen the API family, choose **Application permissions**, and then select your permissions

The screenshot shows the Microsoft Azure portal interface. The left sidebar has a dark theme with various service icons like Home, Dashboard, All services, etc. The main content area is titled 'dotnetcore-daemon-v2 - API permissions'. On the left, under 'Manage', 'API permissions' is selected. The right pane starts with 'Request API permissions' and 'PREVIEW'. It shows two sections: 'Delegated permissions' (selected) and 'Application permissions' (dashed border). Under 'Delegated permissions', there's a note about needing to access the API as the signed-in user. Under 'Application permissions', there's a note about running as a background service or daemon without a signed-in user. Below these are sections for 'Select permissions' and 'Grant consent'. The 'Select permissions' table lists various Microsoft Graph permissions like AccessReview, Application, AuditLog, Calendars, Calls, ChannelMessage, Chat, Contacts, Device, Directory, and Domain. At the bottom are 'Add permissions' and 'Discard' buttons.

## NOTE

The Web API that you want to call needs to define **application permissions (app roles)**, not delegated permissions. For details on how to expose such an API, see [Protected web API: App registration - when your web API is called by a daemon app](#)

Daemon applications require have a tenant admin pre-consent to the application calling the web API. This consent is provided in the same **API Permission** page, by a tenant admin selecting **Grant admin consent to our organization**

If you're an ISV building a multi-tenant application, you'd want to check the [Deployment - case of multi-tenant daemon apps](#) paragraph.

## Registration of secrets or certificates

Like for any confidential client application, you need to register a secret or certificate. You can register your application secrets either through the interactive experience in the [Azure portal](#), or using command-line tools (like PowerShell)

### Registering client secrets using the application registration portal

The management of client credentials happens in the **certificates & secrets** page for an application:

Credentials enable applications to identify themselves to the authentication service when receiving tokens at a web addressable location (using an HTTPS scheme). For a higher level of assurance, we recommend using a certificate (instead of a client secret) as a credential.

THUMBPRINT	START DATE	EXPIRES

DESCRIPTION	EX...	VALUE

- the application secret (also named client secret) is generated by Azure AD, during the registration of the confidential client application. This generation happens when you select **New client secret**. At that point, you must copy the secret string in the clipboard for use in your app, before selecting **Save**. This string won't be presented any longer.
- the certificate is uploaded in the application registration using the **Upload certificate** button. Azure AD only supports certificates that are directly registered on the application, and do not follow certificate chains.

For details, see [Quickstart: Configure a client application to access web APIs | Add credentials to your application](#)

### Registering client secrets using PowerShell

Alternatively, you can register your application with Azure AD using command-line tools. The [active-directory-dotnetcore-daemon-v2](#) sample shows how to register an application secret or a certificate with an Azure AD application:

- For details on how to register an application secret, see [AppCreationScripts/Configure.ps1](#)
- For details on how to register a certificate with the application, see [AppCreationScripts-withCert/Configure.ps1](#)

## Next steps

[Daemon app - app code configuration](#)

# Daemon app that calls web APIs - code configuration

10/30/2019 • 5 minutes to read • [Edit Online](#)

Learn how to configure the code for your daemon application that calls web APIs.

## MSAL Libraries supporting daemon apps

The Microsoft libraries supporting daemon apps are:

MSAL LIBRARY	DESCRIPTION
 MSAL.NET	Supported platforms to build a daemon application are .NET Framework and .NET Core platforms (not UWP, Xamarin.iOS, and Xamarin.Android as those platforms are used to build public client applications)
 MSAL Python	Development in progress - in public preview
 MSAL Java	Development in progress - in public preview

## Configuration of the Authority

Given that the daemon applications don't use delegated permissions, but application permissions, their *supported account type* can't be *Accounts in any organizational directory and personal Microsoft accounts (for example, Skype, Xbox, Outlook.com)*. Indeed, there's no tenant admin to grant consent to the daemon application for Microsoft personal accounts. You'll need to choose *accounts in my organization* or *accounts in any organization*.

Therefore the authority specified in the application configuration should be tenant-ed (specifying a Tenant ID or a domain name associated with your organization).

If you're an ISV and want to provide a multi-tenant tool, you can use `organizations`. But keep in mind that you'll also need to explain to your customers how to grant admin consent. See [Requesting consent for an entire tenant](#) for details. Also there's currently a limitation in MSAL: `organizations` is only allowed when the client credentials are an application secret (not a certificate).

## Application configuration and instantiation

In MSAL libraries, the client credentials (secret or certificate) are passed as a parameter of the confidential client application construction.

### IMPORTANT

Even if your application is a console application running as a service, if it's a daemon application it needs to be a confidential client application.

## Configuration file

The configuration file defines:

- the authority or the cloud instance and tenantId
  - the ClientID that you got from the application registration
  - either a client secret, or a certificate
- 
- .NET
  - Python
  - Java

[appsettings.json](#) from the [.NET Core console daemon](#) sample.

```
{
 "Instance": "https://login.microsoftonline.com/{0}",
 "Tenant": "[Enter here the tenantID or domain name for your Azure AD tenant]",
 "ClientId": "[Enter here the ClientId for your application]",
 "ClientSecret": "[Enter here a client secret for your application]",
 "CertificateName": "[Or instead of client secret: Enter here the name of a certificate (from the user cert
store) as registered with your application]"
}
```

Either you provide a clientSecret or a certificateName. Both settings are exclusive.

## Instantiation of the MSAL application

To instantiate the MSAL application, you need to:

- add, reference, or import the MSAL package (depending on the language)
- Then the construction is different depending on if you're using client secrets or certificates (or, as an advanced scenario, signed assertions)

### Reference the package

Reference the MSAL package in your application code.

- .NET
- Python
- Java

Add the [Microsoft.IdentityClient](#) NuGet package to your application. In MSAL.NET, the confidential client application is represented by the `IConfidentialClientApplication` interface. Use MSAL.NET namespace in the source code

```
using Microsoft.Identity.Client;
IConfidentialClientApplication app;
```

### Instantiate the confidential client application with client secrets

Here is the code to instantiate the confidential client application with a client secret:

- .NET
- Python
- Java

```
app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
 .WithClientSecret(config.ClientSecret)
 .WithAuthority(new Uri(config.Authority))
 .Build();
```

#### Instantiate the confidential client application With client certificate

Here is the code to build an application with a certificate:

- [.NET](#)
- [Python](#)
- [Java](#)

```
X509Certificate2 certificate = ReadCertificate(config.CertificateName);
app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
 .WithCertificate(certificate)
 .WithAuthority(new Uri(config.Authority))
 .Build();
```

#### Advanced scenario - instantiate the confidential client application with client assertions

- [.NET](#)
- [Python](#)
- [Java](#)

Instead of a client secret or a certificate, the confidential client application can also prove its identity using client assertions.

MSAL.NET has two methods to provide signed assertions to the confidential client app:

- `.WithClientAssertion()`
- `.WithClientClaims()`

When you use `WithClientAssertion`, you need to provide a signed JWT. This advanced scenario is detailed in [Client assertions](#)

```
string signedClientAssertion = ComputeAssertion();
app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
 .WithClientAssertion(signedClientAssertion)
 .Build();
```

When you use `WithClientClaims`, MSAL.NET will compute itself a signed assertion containing the claims expected by Azure AD plus additional client claims that you want to send. Here is a code snippet on how to do that:

```
string ipAddress = "192.168.1.2";
var claims = new Dictionary<string, string> { { "client_ip", ipAddress } };
X509Certificate2 certificate = ReadCertificate(config.CertificateName);
app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
 .WithAuthority(new Uri(config.Authority))
 .WithClientClaims(certificate, claims)
 .Build();``
```

Again, for details, see [Client assertions](#).

## Next steps

- [.NET](#)

- Python
- Java

Daemon app - acquiring tokens for the app

# Daemon app that calls web APIs - acquire a token

10/30/2019 • 4 minutes to read • [Edit Online](#)

Once the confidential client application is constructed, you can acquire a token for the app by calling

`AcquireTokenForClient`, passing the scope, and forcing or not a refresh of the token.

## Scopes to request

The scope to request for a client credential flow is the name of the resource followed by `/.default`. This notation tells Azure AD to use the **application level permissions** declared statically during the application registration. Also, as seen previously, these API permissions must be granted by a tenant administrator

- .NET
- Python
- Java

```
ResourceId = "someAppIDURI";
var scopes = new [] { ResourceId + ".default" };
```

### Case of Azure AD (v1.0) resources

The scope used for client credentials should always be `resourceId + ".default"`

#### IMPORTANT

For MSAL asking an access token for a resource accepting a v1.0 access token, Azure AD parses the desired audience from the requested scope by taking everything before the last slash and using it as the resource identifier. Therefore if, like Azure SQL (<https://database.windows.net>) the resource expects an audience ending with a slash (for Azure SQL:

`https://database.windows.net/`), you'll need to request a scope of `https://database.windows.net//.default` (note the double slash). See also MSAL.NET issue #747: Resource url's trailing slash is omitted, which caused sql auth failure.

## AcquireTokenForClient API

To acquire a token for the app, you'll use `AcquireTokenForClient` or the equivalent depending on the platforms.

- .NET
- Python
- Java

```

using Microsoft.Identity.Client;

// With client credentials flows the scopes is ALWAYS of the shape "resource/.default", as the
// application permissions need to be set statically (in the portal or by PowerShell), and then granted by
// a tenant administrator
string[] scopes = new string[] { "https://graph.microsoft.com/.default" };

AuthenticationResult result = null;
try
{
 result = await app.AcquireTokenForClient(scopes)
 .ExecuteAsync();
}
catch (MsalUiRequiredException ex)
{
 // The application does not have sufficient permissions
 // - did you declare enough app permissions in during the app creation?
 // - did the tenant admin needs to grant permissions to the application.
}
catch (MsalServiceException ex) when (ex.Message.Contains("AADSTS70011"))
{
 // Invalid scope. The scope has to be of the form "https://resourceurl/.default"
 // Mitigation: change the scope to be as expected !
}

```

## Protocol

If you don't have yet a library for your language of choice, you might want to use the protocol directly:

### First case: Access token request with a shared secret

```

POST /{tenant}/oauth2/v2.0/token HTTP/1.1 //Line breaks for clarity
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

client_id=535fb089-9ff3-47b6-9bfb-4f1264799865
&scope=https%3A%2F%2Fgraph.microsoft.com%2F.default
&client_secret=qWgdYAmab0YSkuL1qKv5bPX
&grant_type=client_credentials

```

### Second case: Access token request with a certificate

```

POST /{tenant}/oauth2/v2.0/token HTTP/1.1 // Line breaks for clarity
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

scope=https%3A%2F%2Fgraph.microsoft.com%2F.default
&client_id=97e0a5b7-d745-40b6-94fe-5f77d35c6e05
&client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJSUzI1NiIsIng1dCI6Imd40HRHeXN5amNScUtqRlBuZDdSRnd2d1pJMCJ9.eyJ{a lot of characters
here}M8U3bSUKKJDEg
&grant_type=client_credentials

```

For more information, see the protocol documentation: [Microsoft identity platform and the OAuth 2.0 client credentials flow](#).

## Application token cache

In MSAL.NET, `AcquireTokenForClient` uses the **application token cache** (All the other `AcquireTokenXX` methods use the user token cache) Don't call `AcquireTokenSilent` before calling `AcquireTokenForClient` as `AcquireTokenSilent` uses the **user** token cache. `AcquireTokenForClient` checks the **application** token cache itself and updates it.

# Troubleshooting

## Did you use the resource/.default scope?

If you get an error message telling you that you used an invalid scope, you probably didn't use the `resource/.default` scope.

## Did you forget to provide admin consent? Daemon apps need it!

If you get an error when calling the API **Insufficient privileges to complete the operation**, the tenant administrator needs to grant permissions to the application. See step 6 of Register the client app above. You'll typically see an error like the following error description:

```
Failed to call the web API: Forbidden
Content: {
 "error": {
 "code": "Authorization_RequestDenied",
 "message": "Insufficient privileges to complete the operation.",
 "innerError": {
 "request-id": "<guid>",
 "date": "<date>"
 }
 }
}
```

## Next steps

- [.NET](#)
- [Python](#)
- [Java](#)

[Daemon app - calling a web API](#)

# Daemon app that calls web APIs - call a web API from the app

10/30/2019 • 3 minutes to read • [Edit Online](#)

A daemon app can call a web API from a .NET daemon application or call several pre-approved web APIs.

## Calling a web API daemon application

Here is how to use the token to call an API

- [.NET](#)
- [Python](#)
- [Java](#)

### AuthenticationResult properties in MSAL.NET

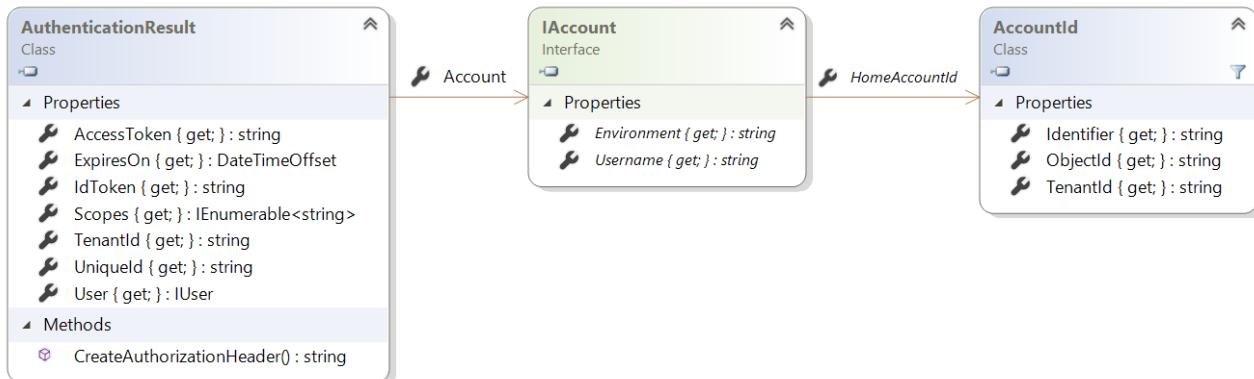
The methods to acquire tokens return an `AuthenticationResult` (or, for the async methods, a `Task<AuthenticationResult>`).

In MSAL.NET, `AuthenticationResult` exposes:

- `AccessToken` for the Web API to access resources. This parameter is a string, usually a base64 encoded JWT but the client should never look inside the access token. The format isn't guaranteed to remain stable and it can be encrypted for the resource. People writing code depending on access token content on the client is one of the biggest sources of errors and client logic breaks. See also [Access tokens](#)
- `IdToken` for the user (this parameter is an encoded JWT). See [ID Tokens](#)
- `ExpiresOn` tells the date/time when the token expires
- `TenantId` contains the tenant in which the user was found. For guest users (Azure AD B2B scenarios), the Tenant ID is the guest tenant, not the unique tenant. When the token is delivered for a user, `AuthenticationResult` also contains information about this user. For confidential client flows where tokens are requested with no user (for the application), this User information is null.
- The `Scopes` for which the token was issued.
- The unique Id for the user.

### IAccount

MSAL.NET defines the notion of Account (through the `IAccount` interface). This breaking change provides the right semantics: the fact that the same user can have several accounts, in different Azure AD directories. Also MSAL.NET provides better information in the case of guest scenarios, as home account information is provided. The following diagram shows the structure of the `IAccount` interface:



The `AccountId` class identifies an account in a specific tenant. It has the following properties:

PROPERTY	DESCRIPTION
<code>TenantId</code>	A string representation for a GUID, which is the ID of the tenant where the account resides.
<code>ObjectId</code>	A string representation for a GUID, which is the ID of the user who owns the account in the tenant.
<code>Identifier</code>	Unique identifier for the account. <code>Identifier</code> is the concatenation of <code>ObjectId</code> and <code>TenantId</code> separated by a comma and are not base64 encoded.

The `IAccount` interface represents information about a single account. The same user can be present in different tenants, that is, a user can have multiple accounts. Its members are:

PROPERTY	DESCRIPTION
<code>Username</code>	A string containing the displayable value in UserPrincipalName (UPN) format, for example, john.doe@contoso.com. This string can be null, whereas the <code>HomeAccountId</code> and <code>HomeAccountId.Identifier</code> won't be null. This property replaces the <code>DisplayableId</code> property of <code>IUser</code> in previous versions of MSAL.NET.
<code>Environment</code>	A string containing the identity provider for this account, for example, <code>login.microsoftonline.com</code> . This property replaces the <code>IdentityProvider</code> property of <code>IUser</code> , except that <code>IdentityProvider</code> also had information about the tenant (in addition to the cloud environment), whereas here the value is only the host.
<code>HomeAccountId</code>	AccountId of the home account for the user. This property uniquely identifies the user across Azure AD tenants.

## Using the token to call a protected API

Once the `AuthenticationResult` has been returned by MSAL (in `result`), you need to add it to the HTTP authorization header, before making the call to access the protected Web API.

```
httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", result.AccessToken);

// Call Web API.
HttpResponseMessage response = await _httpClient.GetAsync(apiUri);
...
}
```

## Calling several APIs

For daemon apps, the web APIs that you call need to be pre-approved. There won't be any incremental consent with daemon apps (there's no user interaction). The tenant admin needs to pre-consent the application and all the API permissions. If you want to call several APIs, you'll need to acquire a token for each resource, each time calling `AcquireTokenForClient`. MSAL will use the application token cache to avoid unnecessary service calls.

## Next steps

- [.NET](#)
- [Python](#)
- [Java](#)

[Daemon app - move to production](#)

# Daemon app that calls web APIs - move to production

10/30/2019 • 2 minutes to read • [Edit Online](#)

Now that you know how to acquire and use a token for a service-to-service call, learn how to move your app to production.

## Deployment - case of multi-tenant daemon apps

If you're an ISV creating a daemon application that can run in several tenants, you'll need to make sure that the tenant admins:

- Provisions a service principal for the application
- Grants consent to the application

You'll need to explain to your customers how to perform these operations. For more info, see [Requesting consent for an entire tenant](#).

Make your application great:

- Enable [logging](#).
- Enable telemetry.
- Enable [proxies and customize HTTP clients](#).

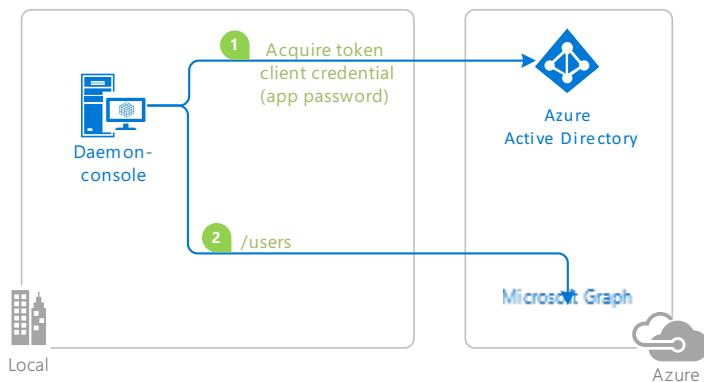
Test your integration:

- Use the [Microsoft identity platform integration checklist](#).

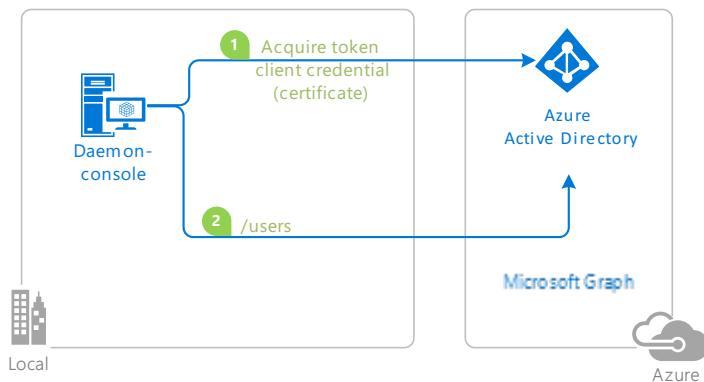
## Next steps

Here are a few links to learn more:

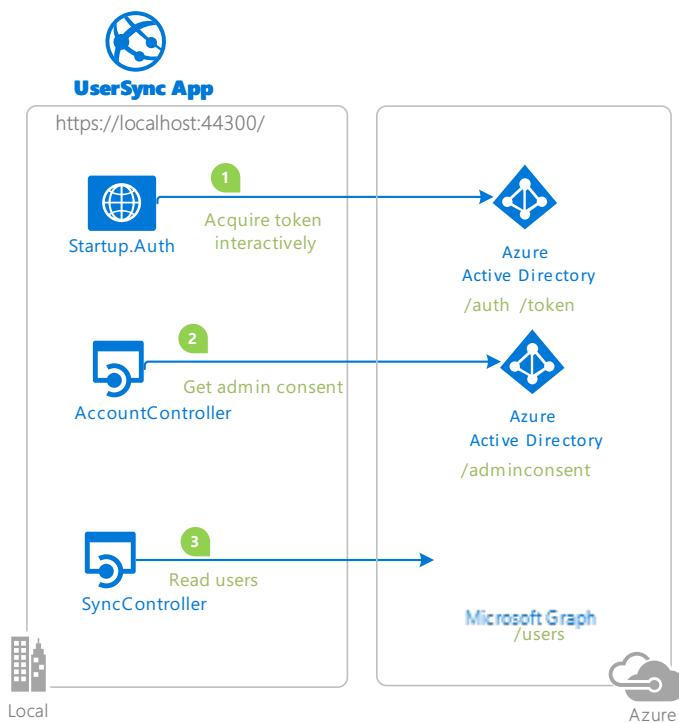
- [.NET](#)
- [Python](#)
- [Java](#)
- If you have not already, try the quickstart [Acquire a token and call Microsoft Graph API from a console app using app's identity](#).
- Reference documentation for:
  - Instantiating [ConfidentialClientApplication](#)
  - Calling [AcquireTokenForClient](#)
- Other samples/tutorials:
  - [microsoft-identity-platform-console-daemon](#) features a simple .NET Core daemon console application that displays the users of a tenant querying the Microsoft Graph.



The same sample also illustrates the variation with certificates.



- [microsoft-identity-platform-aspnet-webapp-daemon](#) features an ASP.NET MVC web application that syncs data from Microsoft Graph using the identity of the application instead of on behalf of a user. The sample also illustrates the admin consent process.



# Scenario: Mobile application that calls web APIs

10/23/2019 • 2 minutes to read • [Edit Online](#)

Learn all you need to know to build a mobile app that calls web APIs.

## Prerequisites

Before reading this article, you should be familiar with the following concepts or read the following articles:

- [Microsoft identity platform overview](#)
- [Authentication basics](#)
- [Audiences](#)
- [Application and service principals](#)
- [Permissions and consent](#)
- [ID tokens and access tokens](#)

## Getting started

Create your first mobile application and try out a quickstart!

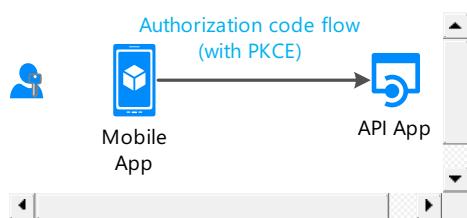
[Quickstart: Acquire a token and call Microsoft Graph API from an Android app](#)

[Quickstart: Acquire a token and call Microsoft Graph API from an iOS app](#)

[Quickstart: Acquire a token and call Microsoft Graph API from a Xamarin iOS & Android app](#)

## Overview

A personalized, seamless user experience is essential for mobile apps. Microsoft identity platform enables mobile developers to create that experience for iOS and Android users. Your application can sign in Azure Active Directory (Azure AD) users, personal Microsoft account users, and Azure AD B2C users and acquire tokens to call a web API on their behalf. To implement these flows, we'll use Microsoft Authentication Library (MSAL), which implements the industry standard [OAuth2.0 authorization code flow](#).



Considerations for mobile apps:

- **User experience is key:** Allow users to see the value of your app before asking for sign-in, and request only the required permissions.
- **Support all user configurations:** Many mobile business users are under Conditional Access and device compliance policies. Be sure to support these key scenarios.
- **Implement single sign-on (SSO):** MSAL and Microsoft identity platform make enabling single sign-on simple through the device's browser or the Microsoft Authenticator (and Intune Company Portal on Android).

## Specifics

Keep these considerations in mind when you build a mobile app on Microsoft identity platform:

- Depending on the platform, some user interaction might be required the first time users sign in. For example, iOS requires apps to show user interaction when using SSO the first time through Microsoft Authenticator (and Intune Company Portal on Android).
- On iOS and Android, MSAL might use an external browser (which might appear on top of your app) to sign in users. You can customize the configuration to use in-app WebViews instead.
- Never use a secret in a mobile application. It will be accessible to all users.

## Next steps

[App registration](#)

# Mobile app that calls web APIs - app registration

10/23/2019 • 2 minutes to read • [Edit Online](#)

This article contains the app registration instructions for creating a mobile application.

## Supported accounts types

The account types supported in mobile applications depend on the experience you want to enable, and the flows you want to use.

### Audience for interactive token acquisition

Most mobile applications use interactive authentication. If that's your case, you can sign in users from any [account type](#)

### Audience for Integrated authentication, username/password, and B2C

- If you intend to use Integrated Windows authentication (possible in UWP apps) or username/password, your application needs to sign in users in your own tenant (LOB developer), or in Azure Active directory organizations (ISV scenario). These authentication flows aren't supported for Microsoft personal accounts
- If you sign in users with social identities passing a B2C authority and policy, you can only use interactive and username-password authentication. Username-password is currently only supported on Xamarin.iOS, Xamarin.Android and UWP.

For the big picture, see [Scenarios and supported authentication flows](#) and [Scenarios and supported platforms and languages](#)

## Platform configuration and redirect URIs

### Interactive authentication

When building a mobile app using interactive authentication, the most critical registration step is the redirect URI. This can be set through the [platform configuration in the Authentication blade](#).

This experience will enable your app to get single sign-on (SSO) through the Microsoft Authenticator (and Intune Company Portal on Android) as well as support device management policies.

Note that there is a preview experience in the app registration portal to help you compute the brokered reply URI for iOS and Android applications:

1. In the app registration choose **Authentication** and selection **Try-out the new experience**

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various service icons like Home, Dashboard, App Services, and Azure Active Directory. The main content area is titled 'my ios app - Authentication'. It has tabs for Overview, Quickstart, Manage, and Support + Troubleshooting. Under Manage, 'Authentication' is selected. A 'TRY OUT THE NEW EXPERIENCE' button is highlighted with a red box. Below it, there's a section for 'Redirect URLs' with a 'TYPE' dropdown set to 'Web' and a 'REDIRECT URI' input field containing 'e.g. https://myapp.com/auth'. There's also a list of suggested redirect URLs for public clients.

## 2. Select Add platform

This screenshot shows the 'Platform configurations' page for the same application. The sidebar and navigation bar are identical to the previous screenshot. The main content area is titled 'my ios app - Platform configurations'. It includes sections for 'Platform configurations' (with a note about required additional configuration for the target platform), 'Supported account types' (radio buttons for 'Accounts in this organizational directory only' or 'Accounts in any organizational directory'), and 'Advanced settings' (with a 'Default client type' dropdown set to 'Yes'). A prominent orange box highlights the '+ Add a platform' button.

## 3. When the list of platforms is supported, select iOS

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various service icons like App Services, SQL databases, and Storage accounts. The main area is titled 'my ios app - Platform configurations'. It has sections for 'Web applications' (with 'Web' selected), 'Mobile applications' (with 'iOS' selected and highlighted by a red box), and 'Desktop + devices'. Under 'Mobile applications', there's a sub-section for 'iOS' which includes Objective-C, Swift, and Xamarin. The 'Advanced settings' section at the bottom has a 'Default client type' dropdown set to 'Public client'. A note says 'Treat application as a public client. Required for the use of the following flows where a redirect URI is required.' followed by a list of three items: 'Resource owner password credential (ROPC) Learn more', 'Device code flow Learn more', and 'Integrated Windows Authentication (IWA) Learn more'.

#### 4. Enter your bundle ID as requested, and then press **Register**

This screenshot shows the 'Configure your iOS app' step in the Azure portal. The 'iOS' platform is selected. In the 'Advanced settings' section, there's a 'Default client type' dropdown set to 'Public client'. Below it, a note says 'Treat application as a public client. Required for the use of the following flows where a redirect URI is required.' followed by a list of three items: 'Resource owner password credential (ROPC) Learn more', 'Device code flow Learn more', and 'Integrated Windows Authentication (IWA) Learn more'. At the bottom right, there are 'Configure' and 'Cancel' buttons. The 'Configure' button is highlighted with a red box.

#### 5. The redirect URI is computed for you.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various service icons like App Services, SQL databases, and Azure Active Directory. The main content area is titled "my ios app - Platform configurations". It has a navigation bar with "Save", "Discard", "Switch to the old experience", and "Get feedback?". Below this is a section titled "Platform configurations" with a note about redirect URIs and authentication settings. A "Quickstart" button is available. The "iOS" platform configuration is selected, showing a "BUNDLE ID" of "com.yourcompany.xforms" and a "REDIRECT URI" of "msauth.com.yourcompany.xforms://auth". There are "Add URI" and "View" buttons. At the bottom, there's a section for "Supported account types" with two radio button options: "Accounts in this organizational directory only (msidentity-samples-testing)" (selected) and "Accounts in any organizational directory". A note states: "Due to temporary differences in supported functionality, we don't recommend enabling personal Microsoft accounts for an existing registration. If you need to enable personal accounts, you can do so using the manifest editor." A "Help me decide..." link is also present.

If you prefer to manually configure the redirect URI, you can do so through the Application Manifest. The recommended format is the following:

- **iOS:** `msauth.<BUNDLE_ID>://auth` (for instance "msauth.com.yourcompany.appName://auth")
- **Android:** `msauth://<PACKAGE_NAME>/<SIGNATURE_HASH>`
  - The Android signature hash can be generated using the release or debug keys through the KeyTool command.

## Username password

If your app is only using Username/Password, you don't need to register a redirect URI for your application. Indeed, this flow does a round trip to the Microsoft identity platform v2.0 endpoint and your application won't be called back on any specific URI. However, you need to express that your application is a public client application. This configuration is achieved by going to the **Authentication** section for your application, and in the **Advanced settings** subsection, choose **Yes**, to the question **Treat application as a public client** (in the **Default client type** paragraph)

## API permissions

Mobile applications call APIs on behalf of the signed-in user. Your app needs to request delegated permissions, also referred to as scopes. Depending on the desired experience, this can be done statically through the Azure portal or dynamically at run-time. Statically registering permissions allows admins to easily approve your app and is recommended.

## Next steps

[Code configuration](#)

# Mobile app that calls web APIs - code configuration

10/30/2019 • 7 minutes to read • [Edit Online](#)

Once you've created your application, you'll learn how to configure the code using the app registration parameters. Mobile applications also have some complex specifics, which have to do with fitting into the framework used to build these apps.

## MSAL libraries supporting mobile apps

The Microsoft libraries supporting mobile apps are:

MSAL LIBRARY	DESCRIPTION
 MSAL.NET	To develop portable applications. MSAL.NET supported platforms to build a mobile application are UWP, Xamarin.iOS, and Xamarin.Android.
 MSAL.iOS	To develop native iOS applications with Objective-C or Swift
 MSAL.Android	To develop native Android applications in Java for Android

## Instantiating the application

### Android

Mobile applications use the `PublicClientApplication` class. Here is how to instantiate it:

```
PublicClientApplication sampleApp = new PublicClientApplication(
 this.getApplicationContext(),
 R.raw.auth_config);
```

### iOS

Mobile applications on iOS need to instantiate the `MSALPublicClientApplication` class.

Objective-C:

```
NSError *msalError = nil;

MSALPublicClientApplicationConfig *config = [[MSALPublicClientApplicationConfig alloc] initWithClientId:@"<your-client-id-here>"];
MSALPublicClientApplication *application = [[MSALPublicClientApplication alloc] initWithConfiguration:config
error:&msalError];
```

Swift:

```
let config = MSALPublicClientApplicationConfig(clientId: "<your-client-id-here>")
if let application = try? MSALPublicClientApplication(configuration: config){ /* Use application */}
```

There are [additional MSALPublicClientApplicationConfig properties](#) which can override the default authority, specify a redirect URI or change MSAL token caching behavior.

## Xamarin or UWP

The following paragraph explains how to instantiate the application for Xamarin.iOS, Xamarin.Android, and UWP apps.

### Instantiating the application

In Xamarin, or UWP, the simplest way to instantiate the application is as follows, where the `clientId` is the Guid of your registered app.

```
var app = PublicClientApplicationBuilder.Create(clientId)
 .Build();
```

There are additional `Withparameter` methods which set the UI parent, override the default authority, specify a client name and version (for telemetry), specify a redirect URI, Specify the Http factory to use (for instance to handle proxies, specify telemetry and logging) . This is the topic of the following paragraphs.

#### Specifying the parent UI/Window/Activity

On Android, you need to pass the parent activity before you do the interactive authentication. On iOS, when using a broker, you need to pass-in the ViewController. In the same way on UWP, you might want to pass-in the parent window. This is possible when you acquire the token, but it's also possible to specify a callback at app creation time a delegate returning the UIParent.

```
IPublicClientApplication application = PublicClientApplicationBuilder.Create(clientId)
 .ParentActivityOrWindowFunc(() => parentUi)
 .Build();
```

On Android, we recommend you use the `CurrentActivityPlugin` [here](#). Then your `PublicClientApplication` builder code would look like this:

```
// Requires MSAL.NET 4.2 or above
var pca = PublicClientApplicationBuilder
 .Create("<your-client-id-here>")
 .WithParentActivityOrWindow(() => CrossCurrentActivity.Current)
 .Build();
```

#### More app building parameters

- For the list of all modifiers available on `PublicClientApplicationBuilder`, see the reference documentation [PublicClientApplicationBuilder](#)
- For the description of all the options exposed in `PublicClientApplicationOptions` see [PublicClientApplicationOptions](#), in the reference documentation

## Xamarin iOS specific considerations

On Xamarin iOS, there are several considerations that you must take into account when using MSAL.NET:

1. [Override and implement the `OpenUrl` function in the `AppDelegate`](#)
2. [Enable Keychain groups](#)
3. [Enable token cache sharing](#)
4. [Enable Keychain access](#)

Details are provided in [Xamarin iOS considerations](#)

## MSAL for iOS and macOS specific considerations

Similar considerations apply when using MSAL for iOS and macOS:

1. [Implement the `openURL` callback](#)
2. [Enable keychain access groups](#)
3. [Customize browsers and WebViews](#)

## Xamarin Android specific considerations

Here are Xamarin Android specifics:

- [Ensuring control goes back to MSAL once the interactive portion of the authentication flow ends](#)
- [Update the Android manifest](#)
- [Use the embedded web view \(optional\)](#)
- [Troubleshooting](#)

Details are provided in [Xamarin Android considerations](#)

Finally, there are some specificities to know about the browsers on Android. They are explained in [Xamarin Android-specific considerations with MSAL.NET](#)

### **UWP specific considerations**

On UWP, you can use corporate networks. For additional information about using the MSAL library with UWP, see [Universal Windows Platform-specific considerations with MSAL.NET](#).

## Configuring the application to use the broker

### **Why use brokers in iOS and Android applications?**

On Android and iOS, brokers enable:

- Single Sign On (SSO) when device is registered with AAD. Your users won't need to sign-in to each application.
- Device identification. Enables Azure AD device related Conditional Access policies, by accessing the device certificate that was created on the device when it was workplace joined.
- Application identification verification. When an application calls the broker, it passes its redirect url, and the broker verifies it.

### **Enable the broker on Xamarin**

To enable one of these features, use the `WithBroker()` parameter when calling the `PublicClientApplicationBuilder.CreateApplication` method. `.WithBroker()` is set to true by default. Follow the steps below for [Xamarin.iOS](#).

### **Enable the broker for MSAL for Android**

See [Brokered auth in Android](#) for information about enabling a broker on Android.

### **Enable the broker for MSAL for iOS and macOS**

Brokered authentication is enabled by default for AAD scenarios in MSAL for iOS and macOS. Follow the steps below to configure your application for brokered authentication support for [MSAL for iOS and macOS](#). Note that some steps are different between [MSAL for Xamarin.iOS](#) and [MSAL for iOS and macOS](#).

### **Brokered Authentication for Xamarin.iOS**

Follow the steps below to enable your Xamarin.iOS app to talk with the [Microsoft Authenticator](#) app.

#### **Step 1: Enable broker support**

Broker support is enabled on a per- `PublicClientApplicationBuilder` basis. It's disabled by default. You must use the `WithBroker()` parameter (set to true by default) when creating the `PublicClientApplication` through the `PublicClientApplicationBuilder`.

```
var app = PublicClientApplicationBuilder
 .Create(ClientId)
 .WithBroker()
 .WithReplyUri(redirectUriOnIos) // $"msauth.{Bundle.Id}://auth" (see step 6 below)
 .Build();
```

### Step 2: Update AppDelegate to handle the callback

When MSAL.NET calls the broker, the broker will, in turn, call back to your application through the `AppDelegate.OpenUrl` method. Since MSAL will wait for the response from the broker, your application needs to cooperate to call MSAL.NET back. You do this by updating the `AppDelegate.cs` file to override the below method.

```
public override bool OpenUrl(UIApplication app, NSURL url,
 string sourceApplication,
 NSObject annotation)
{
 if (AuthenticationContinuationHelper.IsBrokerResponse(sourceApplication))
 {
 AuthenticationContinuationHelper.SetBrokerContinuationEventArgs(url);
 return true;
 }
 else if (!AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(url))
 {
 return false;
 }
 return true;
}
```

This method is invoked every time the application is launched, and is used as an opportunity to process the response from the broker and complete the authentication process initiated by MSAL.NET.

### Step 3: Set a UIViewController()

With Xamarin iOS, you don't normally need to set an object window, but in this case you do in order to send and receive responses from a broker. Still in `AppDelegate.cs`, set a ViewController.

Do the following to set the object window:

1. In `AppDelegate.cs`, set the `App.RootViewController` to a new `UIViewController()`. This will make sure there's a `UIViewController` with the call to the broker. If it isn't set correctly, you may get this error:  
"uiviewcontroller\_required\_for\_ios\_broker":"UIViewController is null, so MSAL.NET cannot invoke the ios broker. See <https://aka.ms/msal-net-ios-broker>"
2. On the `AcquireTokenInteractive` call, use the `.WithParentActivityOrWindow(App.RootViewController)` and pass in the reference to the object window you'll use.

### For example:

In `App.cs`:

```
public static object RootViewController { get; set; }
```

In `AppDelegate.cs`:

```
LoadApplication(new App());
App.RootViewController = new UIViewController();
```

In the Acquire Token call:

```
result = await app.AcquireTokenInteractive(scopes)
 .WithParentActivityOrWindow(App.RootViewController)
 .ExecuteAsync();
```

#### Step 4: Register a URL scheme

MSAL.NET uses URLs to invoke the broker and then return the broker response back to your app. To finish the round trip, you need to register a URL scheme for your app in the `Info.plist` file.

Prefix the `CFBundleURLSchemes` with `msauth`. Then add `CFBundleURLName` to the end.

```
$"msauth.(BundleId)"
```

**For example:** `msauth.com.yourcompany.xforms`

#### NOTE

This URL scheme will become part of the `RedirectUri` used for uniquely identifying your app when receiving the response from the broker.

```
<key>CFBundleURLTypes</key>
<array>
<dict>
<key>CFBundleTypeRole</key>
<string>Editor</string>
<key>CFBundleURLName</key>
<string>com.yourcompany.xforms</string>
<key>CFBundleURLSchemes</key>
<array>
<string>msauth.com.yourcompany.xforms</string>
</array>
</dict>
</array>
```

#### Step 5: LSApplicationQueriesSchemes

MSAL uses `-canOpenURL:` to check if the broker is installed on the device. In iOS 9, Apple locked down what schemes an application can query for.

Add `msauthv2` to the `LSApplicationQueriesSchemes` section of the `Info.plist` file.

```
<key>LSApplicationQueriesSchemes</key>
<array>
<string>msauthv2</string>
</array>
```

## Brokered Authentication for MSAL for iOS and macOS

Brokered authentication is enabled by default for AAD scenarios.

#### Step 1: Update AppDelegate to handle the callback

When MSAL for iOS and macOS calls the broker, the broker will, in turn, call back to your application through the `openURL` method. Since MSAL will wait for the response from the broker, your application needs to cooperate to call MSAL back. You do this by updating the `AppDelegate.m` file to override the below method.

Objective-C:

```

- (BOOL)application:(UIApplication *)app
 openURL:(NSURL *)url
 options:(NSDictionary<UIApplicationOpenURLOptionsKey,id> *)options
{
 return [MSALPublicClientApplication handleMSALResponse:url
sourceApplication:options[UIApplicationOpenURLOptionsSourceApplicationKey]];
}

```

Swift:

```

func application(_ app: UIApplication, open url: URL, options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {

 guard let sourceApplication = options[UIApplication.OpenURLOptionsKey.sourceApplication] as? String
else {
 return false
}

 return MSALPublicClientApplication.handleMSALResponse(url, sourceApplication: sourceApplication)
}

```

Note, that if you adopted UISceneDelegate on iOS 13+, MSAL callback needs to be placed into the `scene:openURLContexts:` of UISceneDelegate instead (see [Apple documentation](#)). MSAL `handleMSALResponse:sourceApplication:` must be called only once for each URL.

#### Step 2: Register a URL scheme

MSAL for iOS and macOS uses URLs to invoke the broker and then return the broker response back to your app. To finish the round trip, you need to register a URL scheme for your app in the `Info.plist` file.

Prefix your custom URL scheme with `msauth.`. Then add **your Bundle Identifier** to the end.

`msauth.(BundleId)`

**For example:** `msauth.com.yourcompany.xforms`

#### NOTE

This URL scheme will become part of the RedirectUri used for uniquely identifying your app when receiving the response from the broker. Make sure that the RedirectUri in the format of `msauth.(BundleId)://auth` is registered for your application in the [Azure Portal](#).

```

<key>CFBundleURLTypes</key>
<array>
<dict>
<key>CFBundleURLSchemes</key>
<array>
<string>msauth.[BUNDLE_ID]</string>
</array>
</dict>
</array>

```

#### Step 3: LSApplicationQueriesSchemes

**Add** `LSApplicationQueriesSchemes` to allow making call to Microsoft Authenticator if installed. Note that "msauthv3" scheme is needed when compiling your app with Xcode 11 and later.

```
<key>LSApplicationQueriesSchemes</key>
<array>
 <string>msauthv2</string>
 <string>msauthv3</string>
</array>
```

## Brokered authentication for Xamarin.Android

MSAL.NET does not yet support brokers for Android.

## Next steps

[Acquiring a token](#)

# Xamarin Android-specific considerations with MSAL.NET

10/23/2019 • 2 minutes to read • [Edit Online](#)

This article discusses specific considerations when using Xamarin Android with the Microsoft Authentication Library for .NET (MSAL.NET).

## Set the parent activity

On Xamarin.Android, you need to set the parent activity so that the token gets back once the interaction has happened.

```
var authResult = AcquireTokenInteractive(scopes)
 .WithParentActivityOrWindow(parentActivity)
 .ExecuteAsync();
```

You can also set this at the PublicClientApplication level (in MSAL4.2+) via a callback.

```
// Requires MSAL.NET 4.2 or above
var pca = PublicClientApplicationBuilder
 .Create("<your-client-id-here>")
 .WithParentActivityOrWindow(() => parentActivity)
 .Build();
```

A recommendation is to use the CurrentActivityPlugin [here](#). Then your PublicClientApplication builder code would look like this:

```
// Requires MSAL.NET 4.2 or above
var pca = PublicClientApplicationBuilder
 .Create("<your-client-id-here>")
 .WithParentActivityOrWindow(() => CrossCurrentActivity.Current)
 .Build();
```

## Ensuring control goes back to MSAL once the interactive portion of the authentication flow ends

On Android, you need to override the `OnActivityResult` method of the `Activity` and call the `SetAuthenticationContinuationEventArgs` method of the `AuthenticationContinuationHelper` MSAL class.

```
protected override void OnActivityResult(int requestCode,
 Result resultCode, Intent data)
{
 base.OnActivityResult(requestCode, resultCode, data);
 AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(requestCode,
 resultCode,
 data);
}
```

That line ensures that the control goes back to MSAL once the interactive portion of the authentication flow ended.

## Update the Android manifest

The `AndroidManifest.xml` should contain the following values:

```
<activity android:name="microsoft.identity.client.BrowserTabActivity">
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<category android:name="android.intent.category.BROWSABLE" />
<data android:scheme="msal{client_id}" android:host="auth" />
</intent-filter>
</activity>
```

## Use the embedded web view (optional)

By default MSAL.NET uses the system web browser, which enables you to get SSO with Web applications and other apps. In some rare cases, you might want to specify that you want to use the embedded web view. For more information, see [MSAL.NET uses a Web browser](#) and [Android system browser](#).

```
bool useEmbeddedWebView = !app.IsSystemWebViewAvailable;

var authResult = AcquireTokenInteractive(scopes)
 .WithParentActivityOrWindow(parentActivity)
 .WithEmbeddedWebView(useEmbeddedWebView)
 .ExecuteAsync();
```

## Troubleshooting

If you create a new Xamarin.Forms application and add a reference to the MSAL.Net NuGet package, this will just work. However, if you want to upgrade an existing Xamarin.Forms application to MSAL.NET preview 1.1.2 or later you might experience build issues.

To troubleshoot these issues, you should:

- Update the existing MSAL.NET NuGet package to MSAL.NET preview 1.1.2 or later
- Check that Xamarin.Forms automatically updated to version 2.5.0.122203 (if not, update to this version)
- Check that Xamarin.Android.Support.v4 automatically updated to version 25.4.0.2 (if not, update to this version)
- All the Xamarin.Android.Support packages should be targeting version 25.4.0.2
- Clean/Rebuild
- Try setting the max parallel project builds to 1 in Visual Studio (Options->Projects and Solutions->Build and Run-> Maximum number of parallel projects builds)
- Alternatively, if you're building from the command line, try removing /m from your command if you're using it.

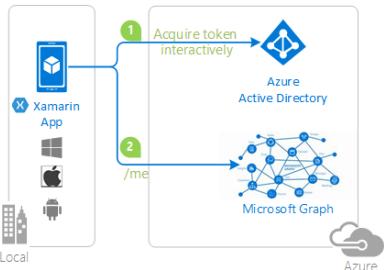
### Error: The name 'AuthenticationContinuationHelper' does not exist in the current context

This is probably because Visual Studio didn't correctly update the `Android.csproj*` file. Sometimes the `<HintPath>` filepath incorrectly contains `netstandard13` instead of **monoandroid90**.

```
<Reference Include="Microsoft.Identity.Client, Version=3.0.4.0, Culture=neutral,
PublicKeyToken=0a613f4dd989e8ae,
processorArchitecture=MSIL">
<HintPath>..\\..\\packages\\Microsoft.Identity.Client.3.0.4-
preview\\lib\\monoandroid90\\Microsoft.Identity.Client.dll</HintPath>
</Reference>
```

## Next steps

More details and samples are provided in the [Android Specific Considerations](#) paragraph of the following sample's `readme.md` file:

SAMPLE	PLATFORM	DESCRIPTION
<a href="https://github.com/Azure-Samples/active-directory-xamarin-native-v2">https://github.com/Azure-Samples/active-directory-xamarin-native-v2</a>	Xamarin iOS, Android, UWP	A simple Xamarin Forms app showcasing how to use MSAL to authenticate MSA and Azure AD via the AADD v2.0 endpoint, and access the Microsoft Graph with the resulting token. 

# Xamarin Android system browser considerations with MSAL.NET

10/30/2019 • 2 minutes to read • [Edit Online](#)

This article discusses specific considerations when using the system browser on Xamarin Android with the Microsoft Authentication Library for .NET (MSAL.NET).

Starting with MSAL.NET 2.4.0-preview, MSAL.NET supports browsers other than Chrome and no longer requires Chrome be installed on the Android device for authentication.

We recommend you use browsers that support custom tabs, such as these:

BROWSERS WITH CUSTOM TABS SUPPORT	PACKAGE NAME
Chrome	com.android.chrome
Microsoft Edge	com.microsoft.emmx
Firefox	org.mozilla.firefox
Ecosia	com.ecosia.android
Kiwi	com.kiwibrowser.browser
Brave	com.brave.browser

In addition to browsers with custom tabs support, based on our testing, a few browsers that don't support custom tabs will also work for authentication: Opera, Opera Mini, InBrowser, and Maxthon. For more information, read [table for test results](#).

## Known issues

- If the user has no browser enabled on the device, MSAL.NET will throw an `AndroidActivityNotFoundException`.
  - **Mitigation:** Inform the user that they should enable a browser (preferably one with custom tabs support) on their device.
- If authentication fails (ex. authentication launches with DuckDuckGo), MSAL.NET will return an `AuthenticationCanceled MsalClientException`.
  - **Root Problem:** A browser with custom tabs support was not enabled on the device. Authentication launched with an alternate browser, which wasn't able to complete authentication.
  - **Mitigation:** Inform the user that they should install a browser (preferably one with custom tab support) on their device.

## Devices and browsers tested

The following table lists the devices and browsers that have been tested.

	BROWSER*	RESULT
Huawei/One+	Chrome*	Pass
Huawei/One+	Edge*	Pass
Huawei/One+	Firefox*	Pass
Huawei/One+	Brave*	Pass
One+	Ecosia*	Pass
One+	Kiwi*	Pass
Huawei/One+	Opera	Pass
Huawei	OperaMini	Pass
Huawei/One+	InBrowser	Pass
One+	Maxthon	Pass
Huawei/One+	DuckDuckGo	User canceled auth
Huawei/One+	UC Browser	User canceled auth
One+	Dolphin	User canceled auth
One+	CM browser	User canceled auth
Huawei/One+	none installed	AndroidActivityNotFound ex

\* Supports custom tabs

## Next steps

For code snippets and additional information on using system browser with Xamarin Android, read this [guide](#).

# Xamarin iOS-specific considerations with MSAL.NET

10/30/2019 • 4 minutes to read • [Edit Online](#)

On Xamarin iOS, there are several considerations that you must take into account when using MSAL.NET

- Known issues with iOS 12 and authentication
- Override and implement the `OpenUrl` function in the `AppDelegate`
- Enable Keychain groups
- Enable token cache sharing
- Enable Keychain access

## Known issues with iOS 12 and authentication

Microsoft has released a [security advisory](#) to provide information about an incompatibility between iOS 12 and some types of authentication. The incompatibility breaks social, WS Fed, and OIDC logins. This advisory also provides guidance on what developers can do to remove current security restrictions added by ASP.NET to their applications to become compatible with iOS 12.

When developing MSAL.NET applications on Xamarin iOS, you may see an infinite loop when trying to sign in to websites from iOS 12 (similar to this [ADAL issue](#)).

You might also see a break in ASP.NET Core OIDC authentication with iOS 12 Safari as described in this [WebKit issue](#).

## Implement OpenUrl

First you need to override the `OpenUrl` method of the `FormsApplicationDelegate` derived class and call `AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs`.

```
public override bool OpenUrl(UIApplication app, NSURL url, NSDictionary options)
{
 AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(url);
 return true;
}
```

You'll also need to define a URL scheme, require permissions for your app to call another app, have a specific form for the redirect URL, and register this redirect URL in the [Azure portal](#)

### Enable keychain access

To enable keychain access, your application must have a keychain access group. You can set your keychain access group by using the `WithIosKeychainSecurityGroup()` api when creating your application as shown below:

To enable single sign-on, you need to set the `PublicClientApplicationBuilder.iOSKeychainSecurityGroup` property to the same value in all of the applications.

An example of this using MSAL v3.x would be:

```
var builder = PublicClientApplicationBuilder
 .Create(ClientId)
 .WithIosKeychainSecurityGroup("com.microsoft.msalrocks")
 .Build();
```

The entitlements.plist should be updated to look like the following XML fragment:

This change is *in addition* to enabling keychain access in the `Entitlements.plist` file, using either the below access group or your own:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>keychain-access-groups</key>
 <array>
 <string>$(AppIdentifierPrefix)com.microsoft.msalrocks</string>
 </array>
</dict>
</plist>
```

An example of this using MSAL v4.x would be:

```
PublicClientApplication.iOSKeychainSecurityGroup = "com.microsoft.msalrocks";
```

When using the `WithIosKeychainSecurityGroup()` api, MSAL will automatically append your security group to the end of the application's "team ID" (AppIdentifierPrefix) because when you build your application using xcode, it will do the same. See [iOS entitlements documentation for more details](#). That's why you need to update the entitlements to include \$(AppIdentifierPrefix) before the keychain access group in the entitlements.plist.

### Enable token cache sharing across iOS applications

From MSAL 2.x, you can specify a Keychain Access Group to use for persisting the token cache across multiple applications. This setting enables you to share the token cache between several applications having the same keychain access group including those developed with [ADAL.NET](#), MSAL.NET Xamarin.iOS applications, and native iOS applications developed with [ADAL.objc](#) or [MSAL.objc](#).

Sharing the token cache allows single sign-on between all of the applications that use the same Keychain access Group.

To enable this cache sharing, you need to set the use the 'WithIosKeychainSecurityGroup()' method to set the keychain access group to the same value in all applications sharing the same cache as shown in the example above.

Earlier, it was mentioned that MSAL added the \$(AppIdentifierPrefix) whenever you use the `WithIosKeychainSecurityGroup()` api. This is because the AppIdentifierPrefix or the "team ID" is used to ensure only applications made by the same publisher can share keychain access.

#### NOTE

The `KeychainSecurityGroup` property is deprecated.

Previously, from MSAL 2.x, developers were forced to include the TeamId prefix when using the `KeychainSecurityGroup` property.

From MSAL 2.7.x, when using the new `iOSKeychainSecurityGroup` property, MSAL will resolve the TeamId prefix during runtime. When using this property, the value should not contain the TeamId prefix. Use the new `iOSKeychainSecurityGroup` property, which does not require you to provide the TeamId, as the previous `KeychainSecurityGroup` property is now obsolete.

## Use Microsoft Authenticator

Your application can use Microsoft Authenticator (a broker) to enable:

- Single Sign On (SSO). Your users won't need to sign-in to each application.
- Device identification. By accessing the device certificate, which was created on the device when it was workplace joined. Your application will be ready if the tenant admins enable Conditional Access related to the devices.
- Application identification verification. When an application calls the broker, it passes its redirect url, and the broker verifies it.

For details on how to enable the broker, see [Use Microsoft Authenticator or Microsoft Intune company portal on Xamarin iOS and Android applications](#).

### **Sample illustrating Xamarin iOS specific properties**

More details are provided in the [iOS Specific Considerations](#) paragraph of the following sample's readme.md file:

SAMPLE	PLATFORM	DESCRIPTION
<a href="https://github.com/Azure-Samples/active-directory-xamarin-native-v2">https://github.com/Azure-Samples/active-directory-xamarin-native-v2</a>	Xamarin iOS, Android, UWP	A simple Xamarin Forms app showcasing how to use MSAL to authenticate MSA and Azure AD via the Azure AD V2.0 endpoint, and access the Microsoft Graph with the resulting token.

# Use Microsoft Authenticator or Microsoft Intune Company Portal on Xamarin applications

10/23/2019 • 3 minutes to read • [Edit Online](#)

On Android and iOS, brokers like Microsoft Authenticator or Microsoft Intune Company Portal enable (Android only):

- Single sign-on (SSO). Your users won't need to sign in to each application.
- Device identification. The broker accesses the device certificate, which was created on the device when it was workplace joined.
- Application identification verification. When an application calls the broker, it passes its redirect URL, and the broker verifies it.

To enable one of these features, application developers need to use the `WithBroker()` parameter when they call the `PublicClientApplicationBuilder.CreateApplication` method. `.WithBroker()` is set to true by default. Developers also need to follow the steps here for [iOS](#) or [Android](#) applications.

## Brokered authentication for iOS

Follow these steps to enable your Xamarin.iOS app to talk with the [Microsoft Authenticator](#) app.

### Step 1: Enable broker support

Broker support is enabled on a per-PublicClientApplication basis. It's disabled by default. Use the `WithBroker()` parameter (set to true by default) when you create the PublicClientApplication through the `PublicClientApplicationBuilder`.

```
var app = PublicClientApplicationBuilder
 .Create(ClientId)
 .WithBroker()
 .WithReplyUri(redirectUriOnIos) // $"msauth.{Bundle.Id}://auth" (see step 6 below)
 .Build();
```

### Step 2: Update AppDelegate to handle the callback

When the Microsoft Authentication Library for .NET (MSAL.NET) calls the broker, the broker in turn calls back to your application through the `OpenUrl` method of the `AppDelegate` class. Because MSAL waits for the response from the broker, your application needs to cooperate to call MSAL.NET back. To enable this cooperation, update the `AppDelegate.cs` file to override the following method.

```

public override bool OpenUrl(UIApplication app, NSURL url,
 string sourceApplication,
 NSObject annotation)
{
 if (AuthenticationContinuationHelper.IsBrokerResponse(sourceApplication))
 {
 AuthenticationContinuationHelper.SetBrokerContinuationEventArgs(url);
 return true;
 }

 else if (!AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(url))
 {
 return false;
 }

 return true;
}

```

This method is invoked every time the application is launched. It's used as an opportunity to process the response from the broker and complete the authentication process initiated by MSAL.NET.

### Step 3: Set a UIViewController()

Still in `AppDelegate.cs`, you need to set an object window. Normally, with Xamarin iOS, you don't need to set the object window. To send and receive responses from the broker, you need an object window.

To do this, you do two things.

1. In `AppDelegate.cs`, set the `App.RootViewController` to a new `UIViewController()`. This assignment makes sure there's a `UIViewController` with the call to the broker. If it isn't set correctly, you might get this error:  
`"uiviewController_required_for_ios_broker": "UIViewController is null, so MSAL.NET cannot invoke the iOS broker. See https://aka.ms/msal-net-ios-broker"`
2. On the `AcquireTokenInteractive` call, use the `.WithParentActivityOrWindow(App.RootViewController)` and pass in the reference to the object window you'll use.

#### For example:

In `App.cs`:

```
public static object RootViewController { get; set; }
```

In `AppDelegate.cs`:

```
LoadApplication(new App());
App.RootViewController = new UIViewController();
```

In the acquire token call:

```
result = await app.AcquireTokenInteractive(scopes)
 .WithParentActivityOrWindow(App.RootViewController)
 .ExecuteAsync();
```

### Step 4: Register a URL scheme

MSAL.NET uses URLs to invoke the broker and then return the broker response back to your app. To finish the round trip, register a URL scheme for your app in the `Info.plist` file.

The `CFBundleURLSchemes` name must include `msauth.` as a prefix, followed by your `CFBundleURLName`.

```
$"msauth.(BundleId)"
```

#### For example:

```
msauth.com.yourcompany.xforms
```

##### NOTE

This URL scheme becomes part of the redirect URI that's used to uniquely identify your app when it receives the response from the broker.

```
<key>CFBundleURLTypes</key>
<array>
<dict>
<key>CFBundleTypeRole</key>
<string>Editor</string>
<key>CFBundleURLName</key>
<string>com.yourcompany.xforms</string>
<key>CFBundleURLSchemes</key>
<array>
<string>msauth.com.yourcompany.xforms</string>
</array>
</dict>
</array>
```

#### Step 5: Add the broker identifier to the LSApplicationQueriesSchemes section

MSAL uses `-canOpenURL:` to check if the broker is installed on the device. In iOS 9, Apple locked down what schemes an application can query for.

Add `msauthv2` to the `LSApplicationQueriesSchemes` section of the `Info.plist` file.

```
<key>LSApplicationQueriesSchemes</key>
<array>
<string>msauthv2</string>
</array>
```

#### Step 6: Register your redirect URI in the application portal

Using the broker adds an extra requirement on your redirect URI. The redirect URI *must* have the following format:

```
$"msauth.{BundleId}://auth"
```

#### For example:

```
public static string redirectUriOnIos = "msauth.com.yourcompany.XForms://auth";
```

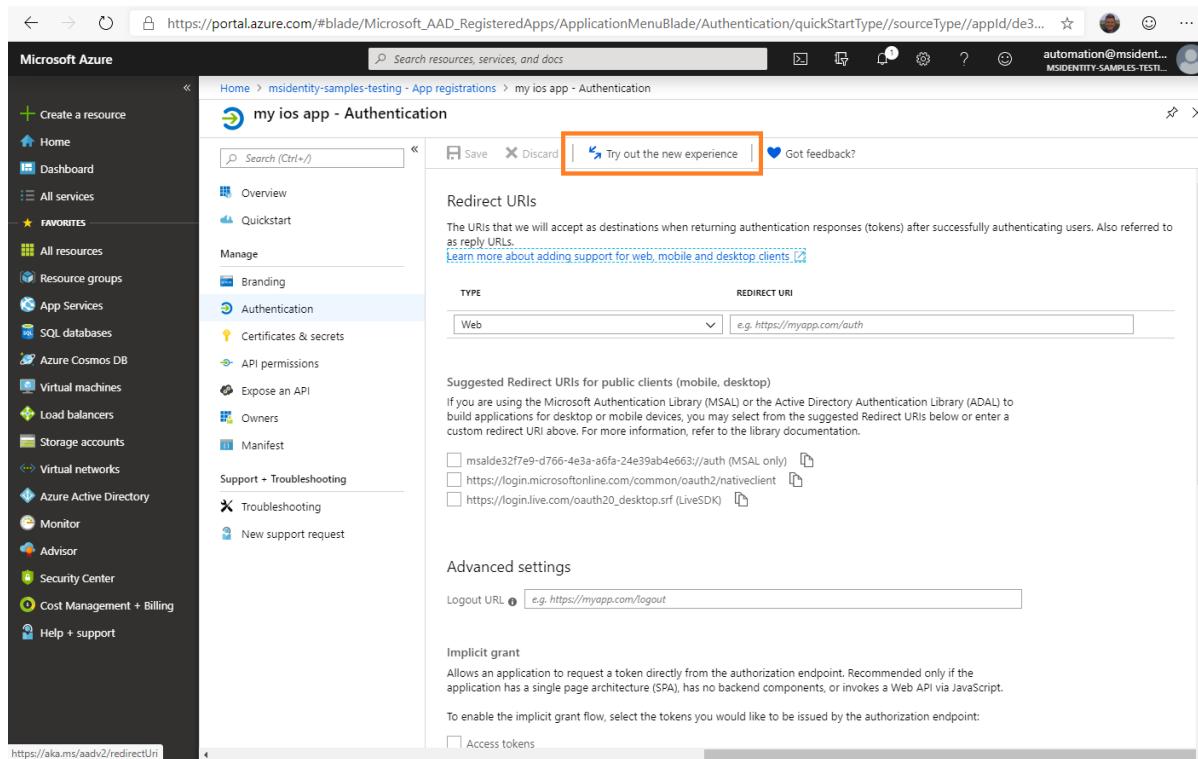
Notice that the redirect URI matches the `CFBundleURLSchemes` name you included in the `Info.plist` file.

#### Step 7: Make sure the redirect URI is registered with your app

This redirect URI needs to be registered on the app registration portal (<https://portal.azure.com>) as a valid redirect URI for your application.

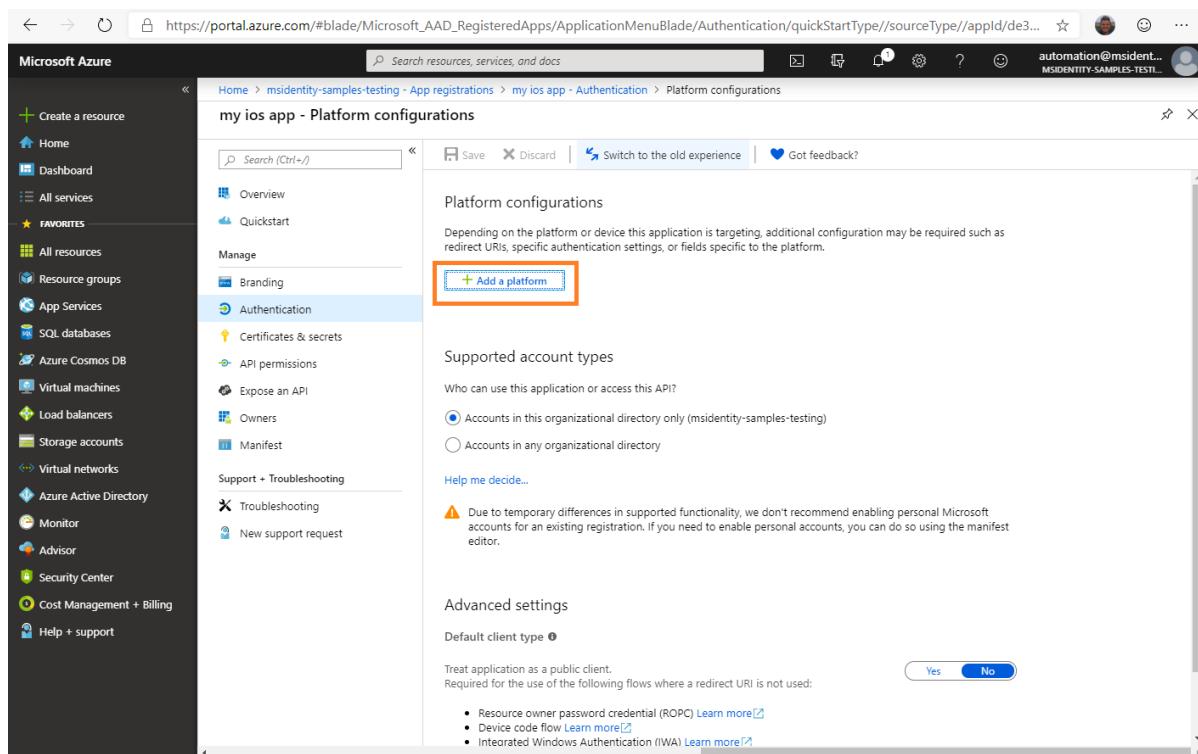
The portal has a new experience app registration portal to help you compute the brokered reply URI from the bundle ID.

## 1. In the app registration, choose **Authentication** and select **Try out the new experience**.



The screenshot shows the Microsoft Azure portal with the URL [https://portal.azure.com/#blade/Microsoft\\_AAD\\_RegisteredApps/ApplicationMenuBlade/Authentication/quickStartType//sourceType//appId/de3...](https://portal.azure.com/#blade/Microsoft_AAD_RegisteredApps/ApplicationMenuBlade/Authentication/quickStartType//sourceType//appId/de3...). The left sidebar includes options like Home, Dashboard, All services, and App Services. The main content area is titled 'my ios app - Authentication'. At the top right, there are 'Save', 'Discard', 'Try out the new experience' (which is highlighted with a red box), and 'Got feedback?' buttons. Below this, the 'Redirect URLs' section is visible, showing a 'Web' type entry with the URL 'e.g. https://myapp.com/auth'. There's also a section for 'Suggested Redirect URIs for public clients (mobile, desktop)' with three entries listed.

## 2. Select **Add a platform**.



The screenshot shows the Microsoft Azure portal with the URL [https://portal.azure.com/#blade/Microsoft\\_AAD\\_RegisteredApps/ApplicationMenuBlade/Authentication/quickStartType//sourceType//appId/de3...](https://portal.azure.com/#blade/Microsoft_AAD_RegisteredApps/ApplicationMenuBlade/Authentication/quickStartType//sourceType//appId/de3...). The left sidebar includes options like Home, Dashboard, All services, and App Services. The main content area is titled 'my ios app - Platform configurations'. At the top right, there are 'Save', 'Discard', 'Switch to the old experience' (which is highlighted with a red box), and 'Got feedback?' buttons. Below this, the 'Platform configurations' section is visible, with a note about targeting specific platforms. A red box highlights the '+ Add a platform' button. The 'Supported account types' section follows, with a note about who can use the application. The 'Accounts in this organizational directory only (msidentity-samples-testing)' radio button is selected. The 'Advanced settings' section is also shown, with a note about treating the application as a public client and a 'Yes' or 'No' button.

## 3. When the list of platforms is supported, select **iOS**.

The screenshot shows the Azure portal interface for managing app registrations. On the left, there's a sidebar with various service icons like App Services, Storage accounts, and Virtual machines. The main area is titled 'my ios app - Platform configurations'. It has tabs for Overview, Quickstart, Manage, and Authentication (which is currently selected). Under 'Platform configurations', it says 'Depending on the platform or device this application i redirect URIs, specific authentication settings, or fields'. There's a 'Supported account types' section with two radio button options: 'Accounts in this organizational directory only (msi)' (selected) and 'Accounts in any organizational directory'. Below this is a 'Help me decide...' link and a warning about temporary differences in supported functions. The 'Advanced settings' section includes a 'Default client type' dropdown set to 'Public client'. At the bottom, there's a list of authentication flows: Resource owner password credential (ROPC), Device code flow (Learn more), and Integrated Windows Authentication (IWA) (Learn more). On the right, there are sections for 'Web applications' (Web, Single-page apps, Web apps), 'Mobile applications' (iOS, Objective-C, Swift, Xamarin; Android, Java, Kotlin, Xamarin), and 'Desktop + devices' (Windows, UWP, Console, IoT & Limited-entry Devices, Classic iOS + Android).

#### 4. Enter your bundle ID as requested, and then select **Configure**.

This screenshot shows the continuation of the configuration process. The URL in the address bar is https://portal.azure.com/#blade/Microsoft\_AAD\_RegisteredApps/ApplicationMenuBlade/Authentication/quickStartType//sourceType//appId/de3... The main content area is titled 'Configure your iOS app' with a 'Quickstart' and 'Docs' link. It says 'Configuring your iOS app enables your users to get device-wide SSO through the Microsoft Authenticator and seamlessly access your application. You will be able to change this later.' A 'Bundle ID' field is highlighted with the value 'com.yourcompany.xform'. At the bottom, there are 'Configure' and 'Cancel' buttons.

#### 5. The redirect URI is computed for you.

The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various service icons like Home, Dashboard, All services, Favorites, All resources, Resource groups, App Services, SQL databases, Azure Cosmos DB, Virtual machines, Load balancers, Storage accounts, Virtual networks, Azure Active Directory, Monitor, Security Center, Cost Management + Billing, and Help + support. The main content area has a header "my ios app - Platform configurations". Below the header are sections for Overview, Quickstart, Manage, and Authentication. Under Authentication, there are sub-options for Certificates & secrets, API permissions, Expose an API, Owners, and Manifest. A "Support + Troubleshooting" section includes links for Troubleshooting and New support request. The main content area also includes a "Platform configurations" section with a note about redirect URIs and specific authentication settings for the platform. It shows an "iOS" configuration card with a "BUNDLE ID" of "com.yourcompany.xforms" and a "REDIRECT URI" of "msauth.com.yourcompany.xforms://auth". There are "Quickstart", "Docs", and "View" buttons for this card. Below this is a "Supported account types" section with a note about who can use the application or access the API, showing a radio button selected for "Accounts in this organizational directory only (msidentity-samples-testing)". There's also a "Help me decide..." link and a warning message about temporary differences in supported functionality regarding personal Microsoft accounts.

## Brokered authentication for Android

The broker support isn't available for Android.

## Next steps

Learn about [Universal Windows Platform-specific considerations with MSAL.NET](#).

# Universal Windows Platform-specific considerations with MSAL.NET

10/23/2019 • 2 minutes to read • [Edit Online](#)

On UWP, there are several considerations that you must take into account when using MSAL.NET.

## The UseCorporateNetwork property

In the WinRT platform, `PublicClientApplication` has the following boolean property `UseCorporateNetwork`. This property enables Win8.1 and UWP applications to benefit from Integrated Windows Authentication (and therefore SSO with the user signed-in with the operating system) if the user is signed-in with an account in a federated Azure AD tenant. When you set this property, MSAL.NET leverages WAB (Web Authentication Broker).

### IMPORTANT

Setting this property to true assumes that the application developer has enabled Integrated Windows Authentication (IWA) in the application. For this:

- In the `Package.appxmanifest` for your UWP application, in the **Capabilities** tab, enable the following capabilities:
  - Enterprise Authentication
  - Private Networks (Client & Server)
  - Shared User Certificate

IWA isn't enabled by default because applications requesting the Enterprise Authentication or Shared User Certificates capabilities require a higher level of verification to be accepted into the Windows Store, and not all developers may wish to perform the higher level of verification.

The underlying implementation on the UWP platform (WAB) doesn't work correctly in Enterprise scenarios where Conditional Access was enabled. The symptom is that the user tries to sign-in with Windows hello, and is proposed to choose a certificate, but:

- the certificate for the pin isn't found,
- or the user chooses it, but never get prompted for the Pin.

A workaround is to use an alternative method (username/password + phone authentication), but the experience isn't good.

## Troubleshooting

Some customers have reported that in some specific enterprise environments, there was the following sign-in error:

We can't connect to the service you need right now. Check your network connection or try this again later

whereas they know they have an internet connection, and that works with a public network.

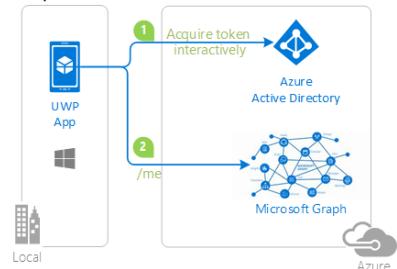
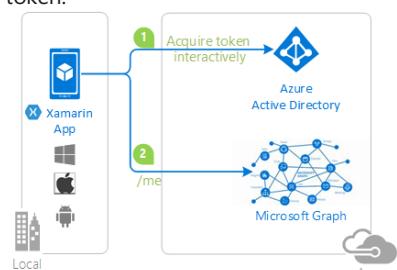
A workaround is to make sure that WAB (the underlying Windows component) allows private network. You can do that by setting a registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\authhost.exe\EnablePrivateNetwork = 00000001
```

For details, see [Web authentication broker - Fiddler](#).

## Next steps

More details are provided in the following samples:

SAMPLE	PLATFORM	DESCRIPTION
<a href="#">active-directory-dotnet-native-uwp-v2</a>	UWP	A Universal Windows Platform client application using msal.net, accessing the Microsoft Graph for a user authenticating with Azure AD v2.0 endpoint.  The diagram illustrates the flow of a token exchange. A UWP App icon is connected to a local machine icon. From the local machine, a line leads to an 'Acquire token interactively' icon, which then connects to an 'Azure Active Directory' box. This box is linked to a 'Microsoft Graph' box, which contains a network of nodes. Finally, an arrow labeled '/me' points from the Microsoft Graph back to the UWP App icon.
<a href="https://github.com/Azure-Samples/active-directory-xamarin-native-v2">https://github.com/Azure-Samples/active-directory-xamarin-native-v2</a>	Xamarin iOS, Android, UWP	A simple Xamarin Forms app showcasing how to use MSAL to authenticate MSA and Azure AD via the AAD v2.0 endpoint, and access the Microsoft Graph with the resulting token.  This diagram is similar to the one above, showing a Xamarin App icon connected to a local machine. The local machine connects to an 'Acquire token interactively' icon, which then connects to an 'Azure Active Directory' box. An arrow labeled '/me' points from the Microsoft Graph back to the Xamarin App icon. The Microsoft Graph box shows a network of nodes.

# Mobile app that calls web APIs - get a token

10/23/2019 • 6 minutes to read • [Edit Online](#)

Before you can begin calling protected web APIs, your app will need an access token. This article walks you through the process for getting a token by using the Microsoft Authentication Library (MSAL).

## Scopes to request

When you request a token, you need to define a scope. The scope determines what data your app can access.

The easiest approach is to combine the desired web API's `App ID URI` with the scope `.default`. Doing so tells Microsoft identity platform that your app requires all scopes set in the portal.

### Android

```
String[] SCOPES = {"https://graph.microsoft.com/.default"};
```

### iOS

```
let scopes = ["https://graph.microsoft.com/.default"]
```

### Xamarin

```
var scopes = new [] {"https://graph.microsoft.com/.default"};
```

## Get tokens

### Acquire tokens via MSAL

MSAL allows apps to acquire tokens silently and interactively. Just call these methods and MSAL returns an access token for the requested scopes. The correct pattern is to perform a silent request and fall back to an interactive request.

### Android

```

String[] SCOPES = {"https://graph.microsoft.com/.default"};
PublicClientApplication sampleApp = new PublicClientApplication(
 this.getApplicationContext(),
 R.raw.auth_config);

// Check if there are any accounts we can sign in silently.
// Result is in the silent callback (success or error).
sampleApp.getAccounts(new PublicClientApplication.AccountsLoadedCallback() {
 @Override
 public void onAccountsLoaded(final List<IAccount> accounts) {

 if (accounts.isEmpty() && accounts.size() == 1) {
 // TODO: Create a silent callback to catch successful or failed request.
 sampleApp.acquireTokenSilentAsync(SCOPES, accounts.get(0), getAuthSilentCallback());
 } else {
 /* No accounts or > 1 account. */
 }
 }
});

[...]

// No accounts found. Interactively request a token.
// TODO: Create an interactive callback to catch successful or failed request.
sampleApp.acquireToken(getActivity(), SCOPES, getAuthInteractiveCallback());

```

## iOS

### First try acquiring a token silently:

Objective-C:

```

NSArray *scopes = @[@"https://graph.microsoft.com/.default"];
NSString *accountIdentifier = @"my.account.id";

MSALAccount *account = [application accountForIdentifier:accountIdentifier error:nil];

MSALSilentTokenParameters *silentParams = [[MSALSilentTokenParameters alloc] initWithScopes:scopes
account:account];
[application acquireTokenSilentWithParameters:silentParams completionBlock:^(MSALResult *result, NSError
*error) {

 if (!error)
 {
 // You'll want to get the account identifier to retrieve and reuse the account
 // for later acquireToken calls
 NSString *accountIdentifier = result.account.identifier;

 // Access token to call the Web API
 NSString *accessToken = result.accessToken;
 }

 // Check the error
 if (error && [error.domain isEqual:MSALErrorDomain] && error.code == MSALErrorInteractionRequired)
 {
 // Interactive auth will be required, call acquireTokenWithParameters:error:
 return;
 }
}];

```

Swift:

```

let scopes = ["https://graph.microsoft.com/.default"]
let accountIdentifier = "my.account.id"

guard let account = try? application.account(forIdentifier: accountIdentifier) else { return }
let silentParameters = MSALSilentTokenParameters(scopes: scopes, account: account)
application.acquireTokenSilent(with: silentParameters) { (result, error) in

 guard let authResult = result, error == nil else {

 let nsError = error! as NSError

 if (nsError.domain == MSALErrorDomain &&
 nsError.code == MSALError.interactionRequired.rawValue) {

 // Interactive auth will be required, call acquireToken()
 return
 }
 return
 }

 // You'll want to get the account identifier to retrieve and reuse the account
 // for later acquireToken calls
 let accountIdentifier = authResult.account.identifier

 // Access token to call the Web API
 let accessToken = authResult.accessToken
}

```

**Then if MSAL returns `MSALErrorInteractionRequired`, try acquiring tokens interactively:**

Objective-C:

```

UIViewController *viewController = ...; // Pass a reference to the view controller that should be used when
getting a token interactively
MSALWebviewParameters *webParameters = [[MSALWebviewParameters alloc]
initWithParentViewController:viewController];
MSALInteractiveTokenParameters *interactiveParams = [[MSALInteractiveTokenParameters alloc]
initWithScopes:scopes webviewParameters:webParameters];
[application acquireTokenWithParameters:interactiveParams completionBlock:^(MSALResult *result, NSError
*error) {
if (!error)
{
 // You'll want to get the account identifier to retrieve and reuse the account
 // for later acquireToken calls
 NSString *accountIdentifier = result.account.identifier;

 NSString *accessToken = result.accessToken;
}
}];

```

Swift:

```

let viewController = ... // Pass a reference to the view controller that should be used when getting a token
interactively
let webviewParameters = MSALWebviewParameters(parentViewController: viewController)
let interactiveParameters = MSALInteractiveTokenParameters(scopes: scopes, webviewParameters:
webviewParameters)
application.acquireToken(with: interactiveParameters, completionBlock: { (result, error) in

guard let authResult = result, error == nil else {
print(error!.localizedDescription)
return
}

// Get access token from result
let accessToken = authResult.accessToken
})

```

MSAL for iOS and macOS supports various modifiers when getting a token interactively or silently.

- [Common parameters when getting a token](#)
- [Parameters for interactive token acquisition](#)
- [Parameters for silent token acquisition](#)

#### Xamarin

The following example shows minimal code to get a token interactively for reading the user's profile with Microsoft Graph.

```

string[] scopes = new string[] {"user.read"};
var app = PublicClientApplicationBuilder.Create(clientId).Build();
var accounts = await app.GetAccountsAsync();
AuthenticationResult result;
try
{
 result = await app.AcquireTokenSilent(scopes, accounts.FirstOrDefault())
 .ExecuteAsync();
}
catch(MsalUiRequiredException)
{
 result = await app.AcquireTokenInteractive(scopes)
 .ExecuteAsync();
}

```

#### Mandatory parameters in MSAL.NET

`AcquireTokenInteractive` has only one mandatory parameter `scopes`, which contains an enumeration of strings that define the scopes for which a token is required. If the token is for the Microsoft Graph, the required scopes can be found in api reference of each Microsoft graph API in the section named "Permissions". For instance, to [list the user's contacts](#), the scope "User.Read", "Contacts.Read" will need to be used. See also [Microsoft Graph permissions reference](#).

If you have not specified it when building the app, on Android, you need to also specify the parent activity (using `.WithParentActivityOrWindow`, see below) so that the token gets back to that parent activity after the interaction. If you don't specify it, an exception will be thrown when calling `.ExecuteAsync()`.

#### Specific optional parameters in MSAL.NET

##### `WithPrompt`

`WithPrompt()` is used to control the interactivity with the user by specifying a Prompt

The screenshot shows the 'Fields' section of the 'Prompt' struct. It lists five fields: Consent, ForceLogin, Never, NoPrompt, and SelectAccount, each represented by a blue circular icon.

The class defines the following constants:

- `SelectAccount` : will force the STS to present the account selection dialog containing accounts for which the user has a session. This option is useful when applications developers want to let user choose among different identities. This option drives MSAL to send `prompt=select_account` to the identity provider. This option is the default, and it does of good job of providing the best possible experience based on the available information (account, presence of a session for the user, and so on....). Don't change it unless you have good reason to do it.
- `Consent` : enables the application developer to force the user be prompted for consent even if consent was granted before. In this case, MSAL sends `prompt=consent` to the identity provider. This option can be used in some security focused applications where the organization governance demands that the user is presented the consent dialog each time the application is used.
- `ForceLogin` : enables the application developer to have the user prompted for credentials by the service even if this user-prompt wouldn't be needed. This option can be useful if Acquiring a token fails, to let the user re-sign-in. In this case, MSAL sends `prompt=login` to the identity provider. Again, we've seen it used in some security focused applications where the organization governance demands that the user relogs-in each time they access specific parts of an application.
- `Never` (for .NET 4.5 and WinRT only) won't prompt the user, but instead will try to use the cookie stored in the hidden embedded web view (See below: Web Views in MSAL.NET). Using this option might fail, and in that case `AcquireTokenInteractive` will throw an exception to notify that a UI interaction is needed, and you'll need to use another `Prompt` parameter.
- `NoPrompt` : Won't send any prompt to the identity provider. This option is only useful for Azure AD B2C edit profile policies (See [B2C specifics](#)).

#### `WithExtraScopeToConsent`

This modifier is used in an advanced scenario where you want the user to pre-consent to several resources upfront (and don't want to use the incremental consent, which is normally used with MSAL.NET / the Microsoft identity platform v2.0). For details see [How-to : have the user consent upfront for several resources](#).

```
var result = await app.AcquireTokenInteractive(scopesForCustomerApi)
 .WithExtraScopeToConsent(scopesForVendorApi)
 .ExecuteAsync();
```

#### Other optional parameters

Learn more about all the other optional parameters for `AcquireTokenInteractive` from the reference documentation for [AcquireTokenInteractiveParameterBuilder](#)

## Acquire tokens via the protocol

We don't recommend using the protocol directly. If you do, the app won't support some single sign-on (SSO), device management, and Conditional Access scenarios.

When you use the protocol to get tokens for mobile apps, you need to make two requests: get an authorization code and exchange it for a token.

### Get authorization code

```
https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize?
client_id=<CLIENT_ID>
&response_type=code
&redirect_uri=<ENCODED_REDIRECT_URI>
&response_mode=query
&scope=openid%20offline_access%20https%3A%2F%2Fgraph.microsoft.com%2F.default
&state=12345
```

#### Get access and refresh token

```
POST /{tenant}/oauth2/v2.0/token HTTP/1.1
Host: https://login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

client_id=<CLIENT_ID>
&scope=https%3A%2F%2Fgraph.microsoft.com%2F.default
&code=0AAABAAAAiL9Kn2Z27UubvWFPbm0gLWQJVzCTE9UkP3pSx1aXxUjq3n8b2JRLk40xVXr...
&redirect_uri=<ENCODED_REDIRECT_URI>
&grant_type=authorization_code
```

## Next steps

[Calling a web API](#)

# Mobile app that calls web APIs - call a web API

10/23/2019 • 6 minutes to read • [Edit Online](#)

After your app has signed in a user and received tokens, MSAL exposes several pieces of information about the user, the user's environment, and the tokens issued. Your app can use these values to call a web API or display a welcome message to the user.

First, we'll look at the MSAL result. Then we'll look at how to use an access token from the `AuthenticationResult` or `result` to call a protected web API.

## MSAL result

MSAL provides the following values:

- `AccessToken` : Used to call protected web APIs in an HTTP Bearer request.
- `IdToken` : Contains useful information about the signed-in user, like the user's name, the home tenant, and a unique identifier for storage.
- `ExpiresOn` : The expiration time of the token. MSAL handles auto refresh for apps.
- `TenantId` : The identifier of the tenant that the user signed in with. For guest users (Azure Active Directory B2B), this value will identify the tenant that the user signed in with, not the user's home tenant.
- `Scopes` : The scopes that were granted with your token. The granted scopes might be a subset of the scopes that you requested.

MSAL also provides an abstraction for an `Account`. An `Account` represents the current user's signed-in account.

- `HomeAccountId` : The identifier of the user's home tenant.
- `UserName` : The user's preferred user name. This might be empty for Azure Active Directory B2C users.
- `AccountIdentifier` : The identifier of the signed-in user. This value will be the same as the `HomeAccountId` value in most cases, unless the user is a guest in another tenant.

## Call an API

After you have the access token, it's easy to call a web API. Your app will use the token to construct an HTTP request and then run the request.

### Android

```

RequestQueue queue = Volley.newRequestQueue(this);
JSONObject parameters = new JSONObject();

try {
 parameters.put("key", "value");
} catch (Exception e) {
 // Error when constructing.
}

JsonObjectRequest request = new JsonObjectRequest(Request.Method.GET, MSGRAPH_URL,
 parameters,new Response.Listener<JSONObject>() {
 @Override
 public void onResponse(JSONObject response) {
 // Successfully called Graph. Process data and send to UI.
 }
 }, new Response.ErrorListener() {
 @Override
 public void onErrorResponse(VolleyError error) {
 // Error.
 }
}) {
 @Override
 public Map<String, String> getHeaders() throws AuthFailureError {
 Map<String, String> headers = new HashMap<>();

 // Put access token in HTTP request.
 headers.put("Authorization", "Bearer " + authResult.getAccessToken());
 return headers;
 }
};

request.setRetryPolicy(new DefaultRetryPolicy(
 3000,
 DefaultRetryPolicy.DEFAULT_MAX_RETRIES,
 DefaultRetryPolicy.DEFAULT_BACKOFF_MULT));
queue.add(request);

```

## MSAL for iOS and macOS

The methods to acquire tokens return an `MSALResult` object. `MSALResult` exposes an `accessToken` property which can be used to call a web API. Access token should be added to the HTTP authorization header, before making the call to access the protected Web API.

Objective-C:

```

NSMutableURLRequest *urlRequest = [NSMutableURLRequest new];
urlRequest.URL = [NSURL URLWithString:@"https://contoso.api.com"];
urlRequest.HTTPMethod = @"GET";
urlRequest.allHTTPHeaderFields = @{@"Authorization" : [NSString stringWithFormat:@"Bearer %@", accessToken]};
};

NSURLSessionDataTask *task =
[[NSURLSession sharedSession] dataTaskWithRequest:urlRequest
 completionHandler:^(NSData * _Nullable data, NSURLResponse * _Nullable response, NSError * _Nullable error) {}];
[task resume];

```

Swift:

```

let urlRequest = NSMutableURLRequest()
urlRequest.url = URL(string: "https://contoso.api.com")!
urlRequest.httpMethod = "GET"
urlRequest.allHTTPHeaderFields = ["Authorization" : "Bearer \\"(accessToken)" "]

let task = URLSession.shared.dataTask(with: urlRequest as URLRequest) { (data: Data?, response: URLResponse?, error: Error?) in }
task.resume()

```

## Xamarin

### AuthenticationResult properties in MSAL.NET

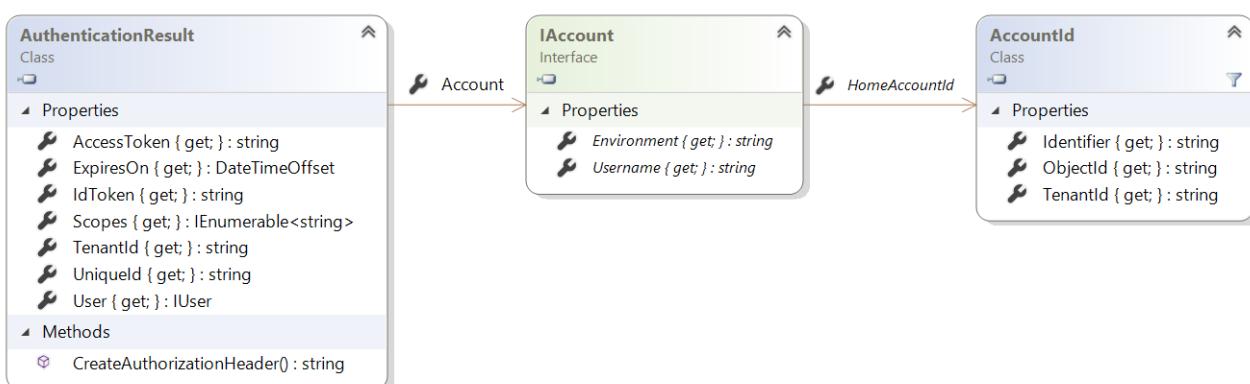
The methods to acquire tokens return an `AuthenticationResult` (or, for the async methods, a `Task<AuthenticationResult>`).

In MSAL.NET, `AuthenticationResult` exposes:

- `AccessToken` for the Web API to access resources. This parameter is a string, usually a base64 encoded JWT but the client should never look inside the access token. The format isn't guaranteed to remain stable and it can be encrypted for the resource. People writing code depending on access token content on the client is one of the biggest sources of errors and client logic breaks. See also [Access tokens](#)
- `IdToken` for the user (this parameter is an encoded JWT). See [ID Tokens](#)
- `ExpiresOn` tells the date/time when the token expires
- `TenantId` contains the tenant in which the user was found. For guest users (Azure AD B2B scenarios), the Tenant ID is the guest tenant, not the unique tenant. When the token is delivered for a user, `AuthenticationResult` also contains information about this user. For confidential client flows where tokens are requested with no user (for the application), this User information is null.
- The `Scopes` for which the token was issued.
- The unique Id for the user.

## IAccount

MSAL.NET defines the notion of Account (through the `IAccount` interface). This breaking change provides the right semantics: the fact that the same user can have several accounts, in different Azure AD directories. Also MSAL.NET provides better information in the case of guest scenarios, as home account information is provided. The following diagram shows the structure of the `IAccount` interface:



The `AccountId` class identifies an account in a specific tenant. It has the following properties:

PROPERTY	DESCRIPTION
<code>TenantId</code>	A string representation for a GUID, which is the ID of the tenant where the account resides.

PROPERTY	DESCRIPTION
<code>ObjectId</code>	A string representation for a GUID, which is the ID of the user who owns the account in the tenant.
<code>Identifier</code>	Unique identifier for the account. <code>Identifier</code> is the concatenation of <code>ObjectId</code> and <code>TenantId</code> separated by a comma and are not base64 encoded.

The `IAccount` interface represents information about a single account. The same user can be present in different tenants, that is, a user can have multiple accounts. Its members are:

PROPERTY	DESCRIPTION
<code>Username</code>	A string containing the displayable value in UserPrincipalName (UPN) format, for example, john.doe@contoso.com. This string can be null, whereas the <code>HomeAccountId</code> and <code>HomeAccountId.Identifier</code> won't be null. This property replaces the <code>DisplayableId</code> property of <code>IUser</code> in previous versions of MSAL.NET.
<code>Environment</code>	A string containing the identity provider for this account, for example, <code>login.microsoftonline.com</code> . This property replaces the <code>IdentityProvider</code> property of <code>IUser</code> , except that <code>IdentityProvider</code> also had information about the tenant (in addition to the cloud environment), whereas here the value is only the host.
<code>HomeAccountId</code>	Accountid of the home account for the user. This property uniquely identifies the user across Azure AD tenants.

## Using the token to call a protected API

Once the `AuthenticationResult` has been returned by MSAL (in `result`), you need to add it to the HTTP authorization header, before making the call to access the protected Web API.

```
httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", result.AccessToken);

// Call Web API.
HttpResponseMessage response = await _httpClient.GetAsync(apiUri);
...
}
```

## Making several API requests

If you need to call the same API several times, or if you need to call multiple APIs, take the following into consideration when you build your app:

- **Incremental consent:** Microsoft identity platform allows apps to get user consent as permissions are required, rather than all at the start. Each time your app is ready to call an API, it should request only the scopes it needs to use.
- **Conditional Access:** In certain scenarios, you might get additional Conditional Access requirements when you make several API requests. This can happen if the first request has no Conditional Access policies applied and your app attempts to silently access a new API that requires Conditional Access. To handle this scenario, be sure to catch errors from silent requests and be prepared to make an interactive request. To learn more, see

## Calling several APIs in Xamarin or UWP - Incremental consent and Conditional Access

If you need to call several APIs for the same user, once you've acquired a token for a user, you can avoid repeatedly asking the user for credentials by subsequently calling `AcquireTokenSilent` to get a token.

```
var result = await app.AcquireTokenXX("scopeApi1")
 .ExecuteAsync();

result = await app.AcquireTokenSilent("scopeApi2")
 .ExecuteAsync();
```

The cases where interaction is required is when:

- The user consented for the first API, but now needs to consent for more scopes (incremental consent)
- The first API didn't require multiple-factor authentication, but the next one does.

```
var result = await app.AcquireTokenXX("scopeApi1")
 .ExecuteAsync();

try
{
 result = await app.AcquireTokenSilent("scopeApi2")
 .ExecuteAsync();
}
catch(MsalUiRequiredException ex)
{
 result = await app.AcquireTokenInteractive("scopeApi2")
 .WithClaims(ex.Cclaims)
 .ExecuteAsync();
}
```

## Next steps

[Move to production](#)

# Mobile app that calls web APIs - move to production

7/23/2019 • 2 minutes to read • [Edit Online](#)

This article provides details about how to improve the quality and reliability of your app before you move it to production.

## Handling errors in mobile applications

A number of error conditions can occur in your app at this point. The main scenarios to handle are silent failures and fallbacks to interaction. Other conditions that you should consider for production include no-network situations, service outages, requirements for admin consent, and other scenario-specific cases.

Each MSAL library has sample code and wiki content that describes how to handle these conditions:

- [MSAL Android Wiki](#)
- [MSAL iOS Wiki](#)
- [MSAL.NET Wiki](#)

## Mitigating and investigating issues

To diagnose issues in your app, it helps to collect data. For information about the kinds of data you can collect, see the MSAL platform wikis.

- Users might ask for help when they encounter problems. A best practice is to capture and temporarily store logs and provide a location where users can upload them. MSAL provides logging extensions to capture detailed information about authentication.
- If it's available, enable telemetry through MSAL to gather data about how users are signing in to your app.

## Next steps

Make your application great:

- Enable [logging](#).
- Enable [telemetry](#).
- Enable [proxies and customize HTTP clients](#).

Test your integration:

- Use the [Microsoft identity platform integration checklist](#).

Try out additional samples available from [Samples | Desktop and mobile public client apps](#)

# Overview of Microsoft Authentication Library (MSAL)

11/4/2019 • 2 minutes to read • [Edit Online](#)

Microsoft Authentication Library (MSAL) enables developers to acquire [tokens](#) from the Microsoft identity platform endpoint in order to access secured Web APIs. These Web APIs can be the Microsoft Graph, other Microsoft APIS, third-party Web APIs, or your own Web API. MSAL is available for .NET, JavaScript, Android, and iOS, which support many different application architectures and platforms.

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 (your org, several orgs, work, and school and Microsoft personal accounts, social identities with Azure AD B2C, users in sovereign, and national clouds).
- Helps you set up your application from configuration files.
- Helps you troubleshoot your app by exposing actionable exceptions, logging, and telemetry.

## Application types and scenarios

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, and daemons and server-side applications.

MSAL can be used in many application scenarios, including the following:

- [Single page applications \(JavaScript\)](#)
- [Web app signing in users](#)
- [Web application signing in a user and calling a web API on behalf of the user](#)
- [Protecting a web API so only authenticated users can access it](#)
- [Web API calling another downstream web API on behalf of the signed-in user](#)
- [Desktop application calling a web API on behalf of the signed-in user](#)
- [Mobile application calling a Web API on behalf of the user who's signed-in interactively.](#)
- [Desktop/service daemon application calling web API on behalf of itself](#)

## Languages and frameworks

LIBRARY	SUPPORTED PLATFORMS AND FRAMEWORKS
<a href="#">MSAL.NET</a>	.NET Framework, .NET Core, Xamarin Android, Xamarin iOS, Universal Windows Platform
<a href="#">MSALjs</a>	JavaScript/TypeScript frameworks such as AngularJS, Ember.js, or Durandal.js

LIBRARY	SUPPORTED PLATFORMS AND FRAMEWORKS
MSAL for Android	Android
MSAL for iOS and macOS	iOS and macOS
MSAL Java (preview)	Java
MSAL Python (preview)	Python

## Differences between ADAL and MSAL

Active Directory Authentication Library (ADAL) integrates with the Azure AD for developers (v1.0) endpoint, where MSAL integrates with the Microsoft identity platform (v2.0) endpoint. The v1.0 endpoint supports work accounts, but not personal accounts. The v2.0 endpoint is the unification of Microsoft personal accounts and work accounts into a single authentication system. Additionally, with MSAL you can also get authentications for Azure AD B2C.

For more specific information, read about [migrating to MSAL.NET from ADAL.NET](#) and [migrating to MSAL.js from ADAL.js](#).

# Accounts & tenant profiles (Android)

9/30/2019 • 6 minutes to read • [Edit Online](#)

This article provides an overview of what an `account` is in the Microsoft identity platform.

The Microsoft Authentication Library (MSAL) API replaces the term *user* with the term *account*. One reason is that a user (human or software agent) may have, or can use, multiple accounts. These accounts may be in the user's own organization, and/or in other organizations that the user is a member of.

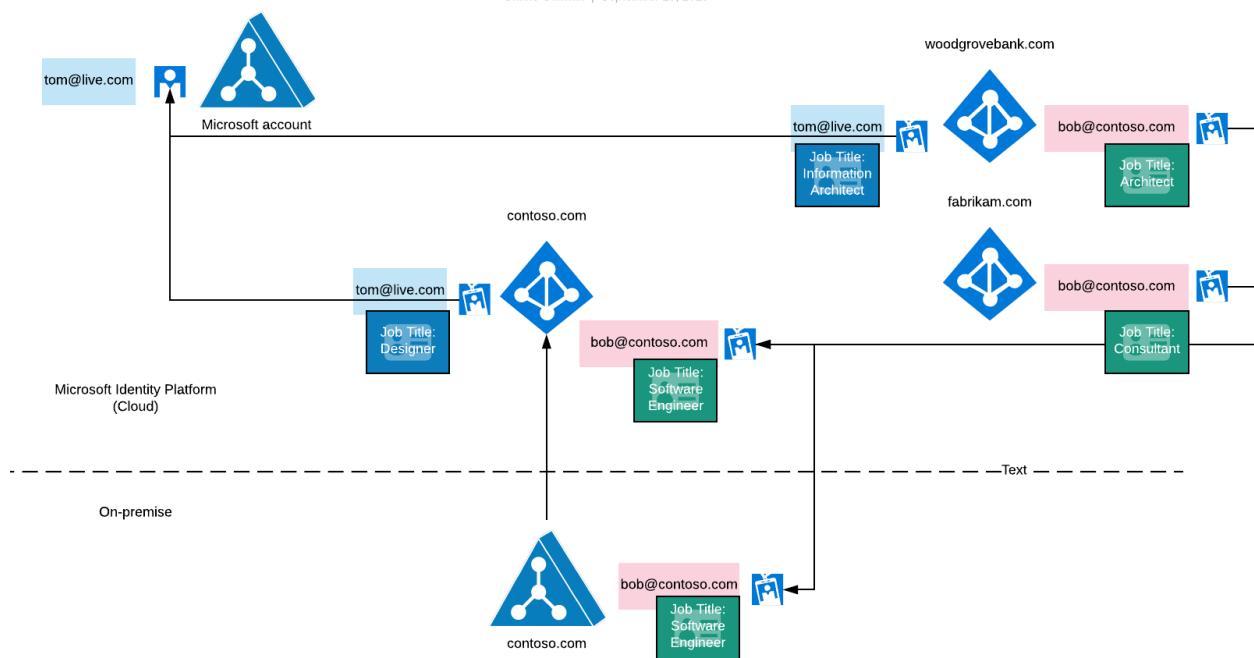
An account in the Microsoft identity platform consists of:

- A unique identifier.
- One or more credentials used to demonstrate ownership/control of the account.
- One or more profiles consisting of attributes such as:
  - Picture, Given Name, Family Name, Title, Office Location
- An account has a source of authority or system of record. This is the system where the account is created and where the credentials associated with that account are stored. In multi-tenant systems like the Microsoft identity platform, the system of record is the `tenant` where the account was created. This tenant is also referred as the `home tenant`.
- Accounts in the Microsoft identity platform have the following systems of record:
  - Azure Active Directory, including Azure Active Directory B2C.
  - Microsoft account (Live).
- Accounts from systems of record outside of the Microsoft identity platform are represented within the Microsoft identity platform including:
  - identities from connected on-premises directories (Windows Server Active Directory)
  - external identities from LinkedIn, GitHub, and so on. In these cases, an account has both an origin system of record and a system of record within the Microsoft identity platform.
- The Microsoft identity platform allows one account to be used to access resources belonging to multiple organizations (Azure Active Directory tenants).
  - To record that an account from one system of record (AAD Tenant A) has access to a resource in another system of record (AAD Tenant B), the account must be represented in the tenant where the resource is defined. This is done by creating a local record of the account from system A in system B.
  - This local record, that is the representation of the account, is bound to the original account.
  - MSAL exposes this local record as a `Tenant Profile`.
  - Tenant Profile can have different attributes that are appropriate to the local context, such as Job Title, Office Location, Contact Information, etc.
- Because an account may be present in one or more tenants, an account may have more than one profile.

## NOTE

MSAL treats the Microsoft account system (Live, MSA) as another tenant within the Microsoft identity platform. The tenant id of the Microsoft account tenant is: `9188040d-6c67-4c5b-b112-36a304b66dad`

# Account overview diagram



In the above diagram:

- The account `bob@contoso.com` is created in the on-premises Windows Server Active Directory (origin on-premises system of record).
- The account `tom@live.com` is created in the Microsoft account tenant.
- `bob@contoso.com` has access to at least one resource in the following Azure Active Directory tenants:
  - `contoso.com` (cloud system of record - linked to on-premises system of record)
  - `fabrikam.com`
  - `woodgrovebank.com`
  - A tenant profile for `bob@contoso.com` exists in each of these tenants.
- `tom@live.com` has access to resources in the following Microsoft tenants:
  - `contoso.com`
  - `fabrikam.com`
  - A tenant profile for `tom@live.com` exists in each of these tenants.
- Information about Tom and Bob in other tenants may differ from that in the system of record. They may differ by attributes such as Job title, Office Location, and so on. They may be members of groups and/or roles within each organization (Azure Active Directory Tenant). We refer to this information as `bob@contoso.com` tenant profile.

In the diagram, `bob@contoso.com` and `tom@live.com` have access to resources in different Azure Active Directory tenants. For more information, see [Add Azure Active Directory B2B collaboration users in the Azure portal](#).

## Accounts and single sign-on (SSO)

The MSAL token cache stores a *single refresh token* per account. That refresh token can be used to silently request access tokens from multiple Microsoft identity platform tenants. When a broker is installed on a device, the account is managed by the broker, and device-wide single sign-on becomes possible.

## IMPORTANT

Business to Consumer (B2C) account and refresh token behavior differs from the rest of the Microsoft identity platform. For more information, see [B2C Policies & Accounts](#).

## Account identifiers

The MSAL account ID isn't an account object ID. It isn't meant to be parsed and/or relied upon to convey anything other than uniqueness within the Microsoft identity platform.

For compatibility with the Azure AD Authentication Library (ADAL), and to ease Migration from ADAL to MSAL, MSAL can look up accounts using any valid identifier for the account available in the MSAL cache. For example, the following will always retrieve the same account object for tom@live.com because each of the identifiers is valid:

```
// The following would always retrieve the same account object for tom@live.com because each identifier is valid

IAccount account = app.getAccount("<tome@live.com msal account id>");
IAccount account = app.getAccount("<tom@live.com contoso user object id>");
IAccount account = app.getAccount("<tom@live.com woodgrovebank user object id>");
```

## Accessing claims about an account

Besides requesting an access token, MSAL also always requests an ID token from each tenant. It does this by always requesting the following scopes:

- openid
- profile

The ID token contains a list of claims. `Claims` are name/value pairs about the account, and are used to make the request.

As mentioned previously, each tenant where an account exists may store different information about the account, including but not limited to attributes such as: job title, office location, and so on.

While an account may be a member or guest in multiple organizations, MSAL doesn't query a service to get a list of the tenants the account is a member of. Instead, MSAL builds up a list of tenants that the account is present in, as a result of token requests that have been made.

The claims exposed on the account object are always the claims from the 'home tenant'/{authority} for an account. If that account hasn't been used to request a token for their home tenant, MSAL can't provide claims via the account object. For example:

```
// Psuedo Code
IAccount account = getAccount("accountid");

String username = account.getClaims().get("preferred_username");
String tenantId = account.getClaims().get("tid"); // tenant id
String objectId = account.getClaims().get("oid"); // object id
String issuer = account.getClaims().get("iss"); // The tenant specific authority that issued the id_token
```

## TIP

To see a list of claims available from the account object, refer to [claims in an id\\_token](#)

**TIP**

To include additional claims in your id\_token, refer to the optional claims documentation in [How to: Provide optional claims to your Azure AD app](#)

## Access tenant profile claims

To access claims about an account as they appear in other tenants, you first need to cast your account object to `IMultiTenantAccount`. All accounts may be multi-tenant, but the number of tenant profiles available via MSAL is based on which tenants you have requested tokens from using the current account. For example:

```
// Psuedo Code
IAccount account = getAccount("accountid");
IMultiTenantAccount multiTenantAccount = (IMultiTenantAccount)account;

multiTenantAccount.getTenantProfiles().get("tenantid for fabrikam").getClaims().get("family_name");
multiTenantAccount.getTenantProfiles().get("tenantid for contoso").getClaims().get("family_name");
```

## B2C policies & accounts

Refresh tokens for an account aren't shared across B2C policies. As a result, single sign-on using tokens isn't possible. This doesn't mean that single sign-on isn't possible. It means single sign-on has to use an interactive experience in which a cookie is available to enable single sign-on.

This also means that in the case of MSAL, if you acquire tokens using different B2C policies, then these are treated as separate accounts - each with their own identifier. If you want to use an account to request a token using `acquireTokenSilent`, then you'll need to select the account from the list of accounts that matches the policy that you're using with the token request. For example:

```
// Get Account For Policy

String policyId = "SignIn";
IAccount signInPolicyAccount = getAccountForPolicyId(app, policyId);

private IAccount getAccountForPolicy(IPublicClientApplication app, String policyId)
{
 List<IAccount> accounts = app.getAccounts();

 foreach(IAccount account : accounts)
 {
 if (account.getClaims().get("tfp").equals(policyId))
 {
 return account;
 }
 }

 return null;
}
```

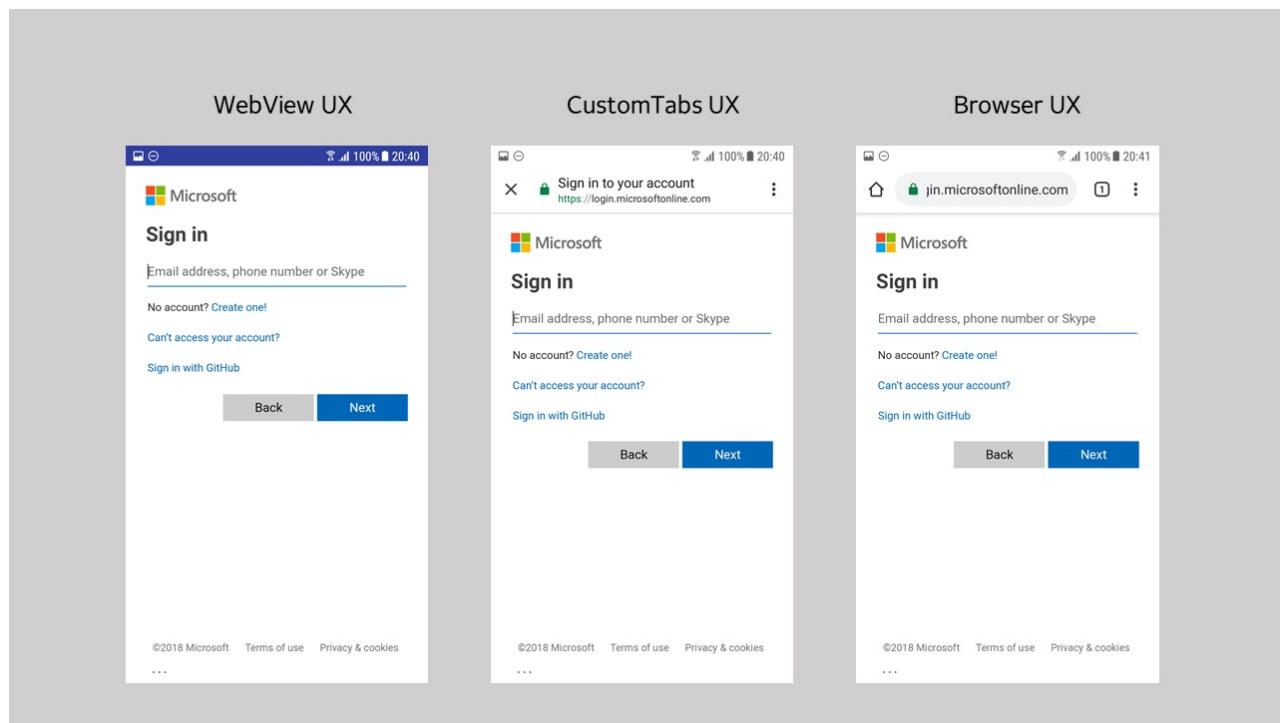
# Authorization agents (Android)

9/30/2019 • 3 minutes to read • [Edit Online](#)

This article describes the different authorization agents that the Microsoft Authentication Library (MSAL) allows your app to use and how to enable them.

Choosing a specific strategy for authorization agents is optional and represents additional functionality apps can customize. Most apps will use the MSAL defaults (see [Understand the Android MSAL configuration file](#) to see the various defaults).

MSAL supports authorization using a `WebView`, or the system browser. The image below shows how it looks using the `WebView`, or the system browser with CustomTabs or without CustomTabs:



## Single sign-in implications

By default, applications integrated with MSAL use the system browser's Custom Tabs to authorize. Unlike WebViews, Custom Tabs share a cookie jar with the default system browser enabling fewer sign-ins with web or other native apps that have integrated with Custom Tabs.

If the application uses a `WebView` strategy without integrating Microsoft Authenticator or Company Portal support into their app, users won't have a Single Sign On (SSO) experience across the device or between native apps and web apps.

If the application uses MSAL with Microsoft Authenticator or Company Portal support, then users can have a SSO experience across applications if the user has an active sign-in with one of the apps.

## WebView

To use the in-app WebView, put the following line in the app configuration JSON that is passed to MSAL:

```
"authorization_user_agent" : "WEBVIEW"
```

When using the in-app `WebView`, the user signs in directly to the app. The tokens are kept inside the sandbox of the app and aren't available outside the app's cookie jar. As a result, the user can't have a SSO experience across applications unless the apps integrate with the Authenticator or Company Portal.

However, `WebView` does provide the capability to customize the look and feel for sign-in UI. See [Android WebViews](#) for more about how to do this customization.

## Default browser plus custom tabs

By default, MSAL uses the browser and a `custom tabs` strategy. You can explicitly indicate this strategy to prevent changes in future releases to `DEFAULT` by using the following JSON configuration in the custom configuration file:

```
"authorization_user_agent" : "BROWSER"
```

Use this approach to provide a SSO experience through the device's browser. MSAL uses a shared cookie jar, which allows other native apps or web apps to achieve SSO on the device by using the persist session cookie set by MSAL.

## Browser selection heuristic

Because it's impossible for MSAL to specify the exact browser package to use on each of the broad array of Android phones, MSAL implements a browser selection heuristic that tries to provide the best cross-device SSO.

MSAL retrieves the full list of browsers installed on the device to select which browser to use. The list is in the order returned by the package manager, which indirectly reflects the user's preferences. For example, the default browser, if set, is the first entry in the list. The *first* browser in the list will be chosen regardless of whether it supports custom tabs. If the browser supports Custom Tabs, MSAL will launch the Custom Tab. Custom Tabs have a look and feel closer to an in-app `WebView` and allow basic UI customization. See [Custom Tabs in Android](#) to learn more.

If there are no browser packages on the device, MSAL uses the in-app `WebView`.

The order of browsers in the browser list is determined by the operating system. It is in order from most preferred to least. If the device default setting isn't changed, the same browser should be launched for each sign in to ensure a SSO experience.

### NOTE

MSAL no longer always prefers Chrome if another browser is set as default. For example, on a device which has both Chrome and another browser pre-installed, MSAL will use the browser the user has set as the default.

## Tested Browsers

The following browsers have been tested to see if they correctly redirect to the `"redirect_uri"` specified in the configuration file:

	BUILT-IN BROWSER	CHROME	OPERA	MICROSOFT EDGE	UC BROWSER	FIREFOX
Nexus 4 (API 17)	pass	pass	not applicable	not applicable	not applicable	not applicable
Samsung S7 (API 25)	pass*	pass	pass	pass	fail	pass

	<b>BUILT-IN BROWSER</b>	<b>CHROME</b>	<b>OPERA</b>	<b>MICROSOFT EDGE</b>	<b>UC BROWSER</b>	<b>FIREFOX</b>
Huawei (API 26)	pass**	pass	fail	pass	pass	pass
Vivo (API 26)	pass	pass	pass	pass	pass	fail
Pixel 2 (API 26)	pass	pass	pass	pass	fail	pass
Oppo	pass	not applicable***	not applicable	not applicable	not applicable	not applicable
OnePlus (API 25)	pass	pass	pass	pass	fail	pass
Nexus (API 28)	pass	pass	pass	pass	fail	pass
MI	pass	pass	pass	pass	fail	pass

\*Samsung's built-in browser is Samsung Internet.

\*\*Huawei's built-in browser is Huawei Browser.

\*\*\*The default browser can't be changed inside the Oppo device setting.

# Brokered auth in Android

10/30/2019 • 4 minutes to read • [Edit Online](#)

## Introduction

You must use one of Microsoft's authentication brokers to participate in device-wide Single Sign-On (SSO) and to meet organizational Conditional Access policies. Integrating with a broker provides the following benefits:

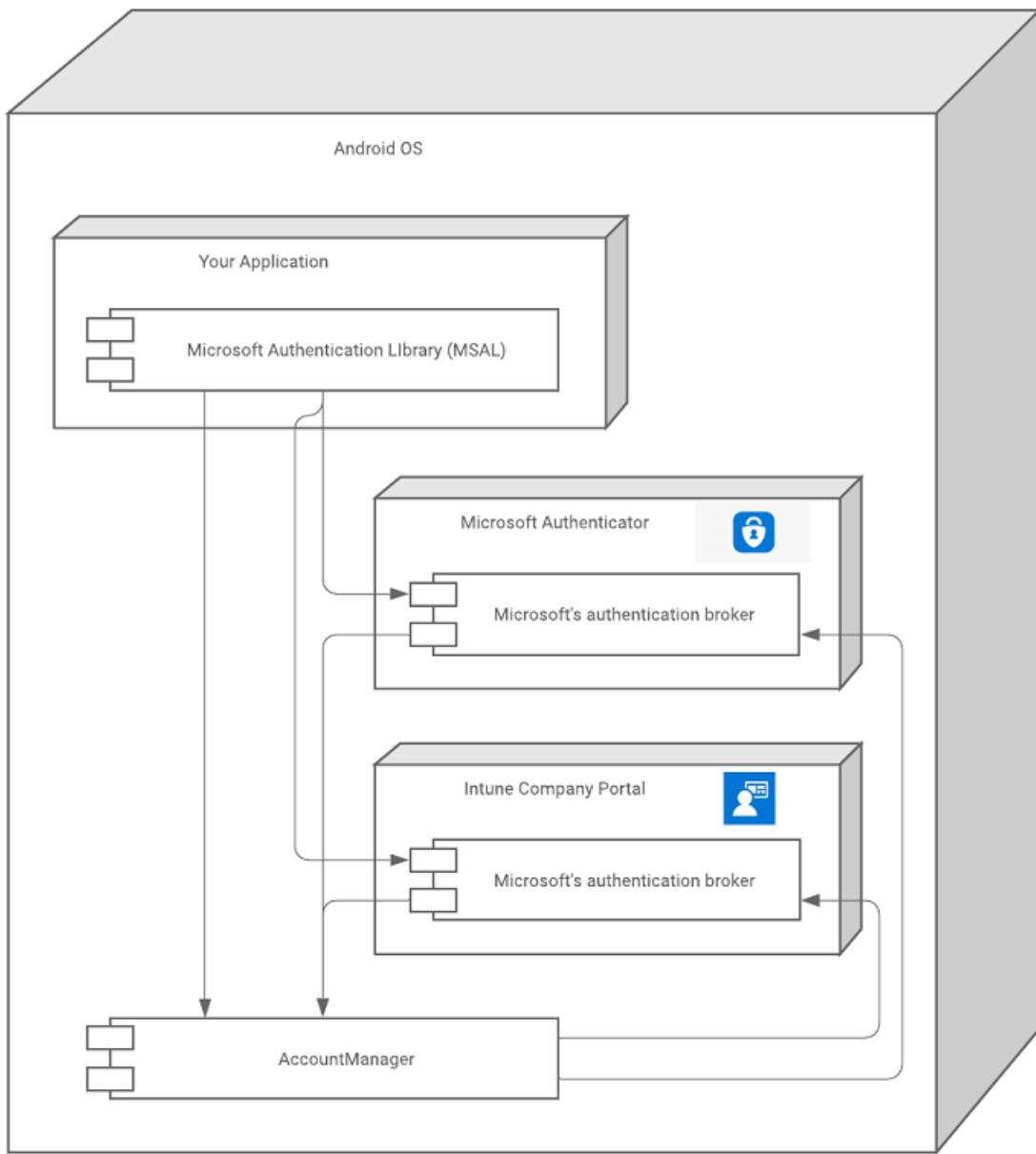
- Device single sign-on
- Conditional access for:
  - Intune App Protection
  - Device Registration (Workplace Join)
  - Mobile Device Management
- Device-wide Account Management
  - via Android AccountManager & Account Settings
  - "Work Account" - custom account type

On Android, the Microsoft Authentication Broker is a component that's included with [Microsoft Authenticator App](#) and [Intune Company Portal](#)

### TIP

Only one application that hosts the broker will be active as the broker at a time. Which application is active as a broker is determined by installation order on the device. The first to be installed, or the last present on the device, becomes the active broker.

The following diagram illustrates the relationship between your app, the Microsoft Authentication Library (MSAL), and Microsoft's authentication brokers.



## Installing apps that host a broker

Broker-hosting apps can be installed by the device owner from their app store (typically Google Play Store) at any time. However, some APIs (resources) are protected by Conditional Access Policies that require devices to be:

- registered (workplace joined) and/or
- enrolled in Device Management or
- enrolled in Intune App Protection

If a device does not already have a broker app installed, MSAL instructs the user to install one as soon as the app attempts to get a token interactively. The app will then need to lead the user through the steps to make the device compliant with the required policy.

## Effects of installing and uninstalling a broker

### When a broker is installed

When a broker is installed on a device, all subsequent interactive token requests (calls to `acquireToken()`) are handled by the broker rather than locally by MSAL. Any SSO state previously available to MSAL is not available to the broker. As a result, the user will need to authenticate again, or select an account from the existing list of

accounts known to the device.

Installing a broker does not require the user to sign in again. Only when the user needs to resolve an `MsalUiRequiredException` will the next request go to the broker. `MsalUiRequiredException` is thrown for a number of reasons, and needs to be resolved interactively. These are some common reasons:

- The user changed the password associated with their account.
- The user's account no longer meets a Conditional Access policy.
- The user revoked their consent for the app to be associated with their account.

## When a broker is uninstalled

If there is only one broker hosting app installed, and it is removed, then the user will need to sign in again. Uninstalling the active broker removes the account and associated tokens from the device.

If Intune Company Portal is installed and is operating as the active broker, and Microsoft Authenticator is also installed, then if the Intune Company Portal (active broker) is uninstalled the user will need to sign in again. Once they sign in again, the Microsoft Authenticator app becomes the active broker.

## Integrating with a broker

### Generating a redirect URI for a broker

You must register a redirect URI that is compatible with the broker. The redirect URI for the broker needs to include your app's package name, as well as the base64 encoded representation of your app's signature.

The format of the redirect URI is: `msauth://<yourpackagename>/<base64urlencodedsignature>`

Generate your Base64 url encoded signature using your app's signing keys. Here are some example commands that use your debug signing keys:

#### macOS

```
keytool -exportcert -alias androiddebugkey -keystore ~/.android/debug.keystore | openssl sha1 -binary |
openssl base64
```

#### Windows

```
keytool -exportcert -alias androiddebugkey -keystore %HOMEPATH%\.android\debug.keystore | openssl sha1 -binary
| openssl base64
```

See [Sign your app](#) for information about signing your app.

#### IMPORTANT

Use your production signing key for the production version of your app.

### Configure MSAL to use a broker

To use a broker in your app, you must attest that you've configured your broker redirect. For example, include both your broker enabled redirect URI--and indicate that you registered it--by including the following in your MSAL configuration file:

```
"redirect_uri" : "<yourbrokerredirecturi>,
"broker_redirect_uri_registered": true
```

**TIP**

The new Azure portal app registration UI helps you generate the broker redirect URI. If you registered your app using the older experience, or did so using the Microsoft app registration portal, you may need to generate the redirect URI and update the list of redirect URLs in the portal manually.

### Broker-related exceptions

MSAL communicates with the broker in two ways:

- Broker bound service
- Android AccountManager

MSAL first uses the broker bound service because calling this service doesn't require any Android permissions. If binding to the bound service fails, MSAL will use the Android AccountManager API. MSAL only does this if your app has already been granted the `"READ_CONTACTS"` permission.

If you get an `MsalClientException` with error code `"BROKER_BIND_FAILURE"`, then there are two options:

- Ask the user to disable power optimization for the Microsoft Authenticator app and the Intune Company Portal.
- Ask the user to grant the `"READ_CONTACTS"` permission

# Migrating applications to MSAL.NET

10/30/2019 • 11 minutes to read • [Edit Online](#)

Both Microsoft Authentication Library for .NET (MSAL.NET) and Azure AD Authentication Library for .NET (ADAL.NET) are used to authenticate Azure AD entities and request tokens from Azure AD. Up until now, most developers have worked with Azure AD for developers platform (v1.0) to authenticate Azure AD identities (work and school accounts) by requesting tokens using Azure AD Authentication Library (ADAL). Using MSAL:

- you can authenticate a broader set of Microsoft identities (Azure AD identities and Microsoft accounts, and social and local accounts through Azure AD B2C) as it uses the Microsoft identity platform endpoint,
- your users will get the best single-sign-on experience.
- your application can enable incremental consent, and supporting Conditional Access is easier
- you benefit from the innovation.

**MSAL.NET is now the recommended auth library to use with the Microsoft identity platform.** No new features will be implemented on ADAL.NET. The efforts are focused on improving MSAL.

This article describes the differences between the Microsoft Authentication Library for .NET (MSAL.NET) and Azure AD Authentication Library for .NET (ADAL.NET) and helps you migrate to MSAL.

## Differences between ADAL and MSAL apps

In most cases you want to use MSAL.NET and the Microsoft identity platform endpoint, which is the latest generation of Microsoft authentication libraries. Using MSAL.NET, you acquire tokens for users signing-in to your application with Azure AD (work and school accounts), Microsoft (personal) accounts (MSA), or Azure AD B2C.

If you are already familiar with the Azure AD for developers (v1.0) endpoint (and ADAL.NET), you might want to read [What's different about the Microsoft identity platform \(v2.0\) endpoint?](#).

However, you still need to use ADAL.NET if your application needs to sign in users with earlier versions of [Active Directory Federation Services \(ADFS\)](#). For more information, see [ADFS support](#).

The following picture summarizes some of the differences between ADAL.NET and MSAL.NET

The diagram compares two code snippets: ADAL.NET on the left and MSAL.NET on the right. It highlights several differences with callout boxes:

- Different namespace:** ADAL.NET uses `Microsoft.IdentityModel.Clients.ActiveDirectory`, while MSAL.NET uses `Microsoft.Identity.Client`.
- Scopes instead of a resource:** ADAL.NET uses `const string resource = "GUID or AppID URI";`, while MSAL.NET uses `string[] scopes = { "User.Read" };`.
- PublicClientApplicationBuilder and ConfidentialClientApplicationBuilder:** MSAL.NET uses `IPublicClientApplication app = PublicClientApplicationBuilder.Create(clientId).Build();` and `AuthenticationResult result = null; var accounts = await app.GetAccountsAsync();`, which implement the fluent API instead of `AuthenticationContext`.
- No need to pass the clientId at every Token acquisition:** MSAL.NET's `AcquireTokenSilent` method takes `scopes` and `accounts.FirstOrDefault()` as parameters, eliminating the need to pass the clientId.
- More explicit exceptions:** MSAL.NET uses `try` blocks with specific exception types like `MsalUiRequiredException` and `MsalException` for error handling.
- Conditional access:** MSAL.NET includes comments for handling errors including conditional access.
- Other errors:** MSAL.NET includes comments for handling other errors.
- ADAL.NET:** A label indicating the original ADAL.NET code.

```
using Microsoft.IdentityModel.Clients.ActiveDirectory;
const string resource = "GUID or AppID URI";
AuthenticationContext app = new AuthenticationContext(authority);

AuthenticationResult result=null;

try
{
 result = await app.AcquireTokenSilentAsync(resource, clientId);
}

catch (AdalException exception)
{
 if (exception.ErrorCode == "user_interaction_required")
 {
 try
 {
 result = await app.AcquireTokenAsync(resource,
 clientId, redirectUri,
 new PlatformParameters(PromptBehavior.Auto));
 }
 catch
 {
 // Handle errors including Conditional access
 }
 }
 // Other errors
}
if (result!=null)
{
 httpClient.DefaultRequestHeaders.Authorization = new
 AuthenticationHeaderValue("Bearer", result.AccessToken);
}
```

```
using Microsoft.Identity.Client;
string[] scopes = { "User.Read" };
IPublicClientApplication app = PublicClientApplicationBuilder.Create(clientId)
 .Build();
AuthenticationResult result = null;
var accounts = await app.GetAccountsAsync();
try
{
 result = await app.AcquireTokenSilent(scopes, accounts.FirstOrDefault())
 .ExecuteAsync();
}
catch (MsalUiRequiredException exception)
{
 try
 {
 result = await app.AcquireTokenInteractive(scopes)
 .ExecuteAsync();
 }
 catch(MsalException)
 {
 // Handle errors including conditional access
 }
 // Other errors
}

if (result != null)
{
 httpClient.DefaultRequestHeaders.Authorization = new
 AuthenticationHeaderValue("Bearer", result.AccessToken);
}
```

## NuGet packages and Namespaces

ADAL.NET is consumed from the [Microsoft.IdentityModel.Clients.ActiveDirectory](#) NuGet package. the namespace to use is `Microsoft.IdentityModel.Clients.ActiveDirectory`.

To use MSAL.NET you will need to add the [Microsoft.Identity.Client](#) NuGet package, and use the `Microsoft.Identity.Client` namespace

### Scopes not resources

ADAL.NET acquires tokens for *resources*, but MSAL.NET acquires tokens for *scopes*. A number of MSAL.NET AcquireToken overrides require a parameter called scopes(`IEnumerable<string> scopes`). This parameter is a simple list of strings that declare the desired permissions and resources that are requested. Well known scopes are the [Microsoft Graph's scopes](#).

It's also possible in MSAL.NET to access v1.0 resources. See details in [Scopes for a v1.0 application](#).

### Core classes

- ADAL.NET uses [AuthenticationContext](#) as the representation of your connection to the Security Token Service (STS) or authorization server, through an Authority. On the contrary, MSAL.NET is designed around [client applications](#). It provides two separate classes: `PublicClientApplication` and `ConfidentialClientApplication`
- Acquiring Tokens: ADAL.NET and MSAL.NET have the same authentication calls (`AcquireTokenAsync` and `AcquireTokenSilentAsync` for ADAL.NET, and `AcquireTokenInteractive` and `AcquireTokenSilent` in MSAL.NET) but with different parameters required. One difference is the fact that, in MSAL.NET, you no longer have to pass in the `ClientID` of your application in every AcquireTokenXX call. Indeed, the `ClientID` is set only once when building the (`IPublicClientApplication` or `IConfidentialClientApplication`).

### IAccount not IUser

ADAL.NET manipulated users. However, a user is a human or a software agent, but it can possess/own/be responsible for one or more accounts in the Microsoft identity system (several Azure AD accounts, Azure AD B2C, Microsoft personal accounts).

MSAL.NET 2.x now defines the concept of Account (through the `IAccount` interface). This breaking change provides the right semantics: the fact that the same user can have several accounts, in different Azure AD directories. Also MSAL.NET provides better information in guest scenarios, as home account information is provided.

For more information about the differences between `IUser` and `IAccount`, see [MSAL.NET 2.x](#).

### Exceptions

#### Interaction required exceptions

MSAL.NET has more explicit exceptions. For example, when silent authentication fails in ADAL the procedure is to catch the exception and look for the `user_interaction_required` error code:

```
catch(AdalException exception)
{
 if (exception.ErrorCode == "user_interaction_required")
 {
 try
 {"try to authenticate interactively"}
 }
}
```

See details in [The recommended pattern to acquire a token](#) with ADAL.NET

Using MSAL.NET, you catch `MsalUiRequiredException` as described in [AcquireTokenSilent](#).

```

 catch(MsalUiRequiredException exception)
 {
 try {"try to authenticate interactively"}
 }

```

### Handling claim challenge exceptions

In ADAL.NET, claim challenge exceptions are handled in the following way:

- `AdalClaimChallengeException` is an exception (deriving from `AdalServiceException`) thrown by the service in case a resource requires more claims from the user (for instance two-factors authentication). The `Claims` member contains some JSON fragment with the claims, which are expected.
- Still in ADAL.NET, the public client application receiving this exception needs to call the `AcquireTokenInteractive` override having a claims parameter. This override of `AcquireTokenInteractive` does not even try to hit the cache as it is not necessary. The reason is that the token in the cache does not have the right claims (otherwise an `AdalClaimChallengeException` would not have been thrown). Therefore, there is no need to look at the cache. Note that the `ClaimChallengeException` can be received in a WebAPI doing OBO, whereas the `AcquireTokenInteractive` needs to be called in a public client application calling this Web API.
- for details, including samples see Handling [AdalClaimChallengeException](#)

In MSAL.NET, claim challenge exceptions are handled in the following way:

- The `Claims` are surfaced in the `MsalServiceException`.
- There is a `.WithClaim(claims)` method that can apply to the `AcquireTokenInteractive` builder.

### Supported grants

Not all the grants are yet supported in MSAL.NET and the v2.0 endpoint. The following is a summary comparing ADAL.NET and MSAL.NET's supported grants.

#### Public client applications

Here are the grants supported in ADAL.NET and MSAL.NET for Desktop and Mobile applications

GRANT	ADAL.NET	MSAL.NET
Interactive	<a href="#">Interactive Auth</a>	<a href="#">Acquiring tokens interactively in MSAL.NET</a>
Integrated Windows Authentication	<a href="#">Integrated authentication on Windows (Kerberos)</a>	<a href="#">Integrated Windows Authentication</a>
Username / Password	<a href="#">Acquiring tokens with username and password</a>	<a href="#">Username Password Authentication</a>
Device code flow	<a href="#">Device profile for devices without web browsers</a>	<a href="#">Device Code flow</a>

#### Confidential client applications

Here are the grants supported in ADAL.NET and MSAL.NET for Web Applications, Web APIs, and daemon applications:

TYPE OF APP	GRANT	ADAL.NET	MSAL.NET
Web App, Web API, daemon	Client Credentials	<a href="#">Client credential flows in ADAL.NET</a>	<a href="#">Client credential flows in MSAL.NET</a>

TYPE OF APP	GRANT	ADAL.NET	MSAL.NET
Web API	On behalf of	Service to service calls on behalf of the user with ADAL.NET	On behalf of in MSAL.NET
Web App	Auth Code	Acquiring tokens with authorization codes on web apps with ADAL.NET	Acquiring tokens with authorization codes on web apps with A MSAL.NET

## Cache persistence

ADAL.NET allows you to extend the `TokenCache` class to implement the desired persistence functionality on platforms without a secure storage (.NET Framework and .NET core) by using the `BeforeAccess`, and `BeforeWrite` methods. For details, see [Token Cache Serialization in ADAL.NET](#).

MSAL.NET makes the token cache a sealed class, removing the ability to extend it. Therefore, your implementation of token cache persistence must be in the form of a helper class that interacts with the sealed token cache. This interaction is described in [Token Cache Serialization in MSAL.NET](#).

## Signification of the common authority

In v1.0, if you use the <https://login.microsoftonline.com/common> authority, you will allow users to sign in with any AAD account (for any organization). See [Authority Validation in ADAL.NET](#)

If you use the <https://login.microsoftonline.com/common> authority in v2.0, you will allow users to sign in with any AAD organization or a Microsoft personal account (MSA). In MSAL.NET, if you want to restrict login to any AAD account (same behavior as with ADAL.NET), you need to use <https://login.microsoftonline.com/organizations>. For details, see the `authority` parameter in [public client application](#).

## v1.0 and v2.0 tokens

There are two versions of tokens:

- v1.0 tokens
- v2.0 tokens

The v1.0 endpoint (used by ADAL) only emits v1.0 tokens.

However, the v2.0 endpoint (used by MSAL) emits the version of the token that the Web API accepts. A property of the application manifest of the Web API enables developers to choose which version of token is accepted. See `accessTokenAcceptedVersion` in the [Application manifest](#) reference documentation.

For more information about v1.0 and v2.0 tokens, see [Azure Active Directory access tokens](#)

## Scopes for a Web API accepting v1.0 tokens

OAuth2 permissions are permission scopes that a v1.0 web API (resource) application exposes to client applications. These permission scopes may be granted to client applications during consent. See the section about `oauth2Permissions` in [Azure Active Directory application manifest](#).

### Scopes to request access to specific OAuth2 permissions of a v1.0 application

If you want to acquire tokens for specific scopes of a v1.0 application (for instance the AAD graph, which is <https://graph.windows.net>), you'd need to create `scopes` by concatenating a desired resource identifier with a desired OAuth2 permission for that resource.

For instance, to access in the name of the user a v1.0 Web API which App ID URI is `ResourceId`, you'd want to use:

```
var scopes = new [] { ResourceId+ "/user_impersonation" };
```

If you want to read and write with MSAL.NET Azure Active Directory using the AAD graph API (<https://graph.windows.net/>), you would create a list of scopes like in the following snippet:

```
ResourceId = "https://graph.windows.net/";
var scopes = new [] { ResourceId + "Directory.Read", ResourceID + "Directory.Write"}
```

**Warning: Should you have one or two slashes in the scope corresponding to a v1.0 Web API**

If you want to write the scope corresponding to the Azure Resource Manager API (<https://management.core.windows.net/>), you need to request the following scope (note the two slashes)

```
var scopes = new[] {"https://management.core.windows.net//user_impersonation"};
var result = await app.AcquireTokenInteractive(scopes).ExecuteAsync();

// then call the API: https://management.azure.com/subscriptions?api-version=2016-09-01
```

This is because the Resource Manager API expects a slash in its audience claim (`aud`), and then there is a slash to separate the API name from the scope.

The logic used by Azure AD is the following:

- For ADAL (v1.0) endpoint with a v1.0 access token (the only possible), `aud=resource`
- For MSAL (v2.0 endpoint) asking an access token for a resource accepting v2.0 tokens, `aud=resource.AppId`
- For MSAL (v2.0 endpoint) asking an access token for a resource accepting a v1.0 access token (which is the case above), Azure AD parses the desired audience from the requested scope by taking everything before the last slash and using it as the resource identifier. Therefore if `https://database.windows.net` expects an audience of `"https://database.windows.net/"`, you'll need to request a scope of `https://database.windows.net//.default`. See also issue #747: Resource url's trailing slash is omitted, which caused sql auth failure #747

### Scopes to request access to all the permissions of a v1.0 application

For instance, if you want to acquire a token for all the static scopes of a v1.0 application, one would need to use

```
ResourceId = "someAppIDURI";
var scopes = new [] { ResourceId+ ".default" };
```

### Scopes to request in the case of client credential flow / daemon app

In the case of client credential flow, the scope to pass would also be `/.default`. This scope tells to Azure AD: "all the app-level permissions that the admin has consented to in the application registration."

## ADAL to MSAL migration

In ADAL.NET v2.X, the refresh tokens were exposed allowing you to develop solutions around the use of these tokens by caching them and using the `AcquireTokenByRefreshToken` methods provided by ADAL 2.x. Some of those solutions were used in scenarios such as:

- Long running services that do actions including refreshing dashboards on behalf of the users whereas the users are no longer connected.
- WebFarm scenarios for enabling the client to bring the RT to the web service (caching is done client side, encrypted cookie, and not server side)

MSAL.NET does not expose refresh tokens, for security reasons: MSAL handles refreshing tokens for you.

Fortunately, MSAL.NET now has an API that allows you to migrate your previous refresh tokens (acquired with ADAL) into the `IConfidentialClientApplication`:

```
/// <summary>
/// Acquires an access token from an existing refresh token and stores it and the refresh token into
/// the application user token cache, where it will be available for further AcquireTokenSilent calls.
/// This method can be used in migration to MSAL from ADAL v2 and in various integration
/// scenarios where you have a RefreshToken available.
/// (see https://aka.ms/msal-net-migration-adal2-msal2)
/// </summary>
/// <param name="scopes">Scope to request from the token endpoint.
/// Setting this to null or empty will request an access token, refresh token and ID token with default
/// scopes</param>
/// <param name="refreshToken">The refresh token from ADAL 2.x</param>
IByRefreshToken.AcquireTokenByRefreshToken(IEnumerable<string> scopes, string refreshToken);
```

With this method, you can provide the previously used refresh token along with any scopes (resources) you desire. The refresh token will be exchanged for a new one and cached into your application.

As this method is intended for scenarios that are not typical, it is not readily accessible with the `IConfidentialClientApplication` without first casting it to `IByRefreshToken`.

This code snippet shows some migration code in a confidential client application.

`GetCachedRefreshTokenForSignedInUser` retrieve the refresh token that was stored in some storage by a previous version of the application that used to leverage ADAL 2.x. `GetTokenCacheForSignedInUser` deserializes a cache for the signed-in user (as confidential client applications should have one cache per user).

```
TokenCache userCache = GetTokenCacheForSignedInUser();
string rt = GetCachedRefreshTokenForSignedInUser();

IConfidentialClientApplication app;
app = ConfidentialClientApplicationBuilder.Create(clientId)
 .WithAuthority(Authority)
 .WithRedirectUri(RedirectUri)
 .WithClientSecret(ClientSecret)
 .Build();
IByRefreshToken appRt = app as IByRefreshToken;

AuthenticationResult result = await appRt.AcquireTokenByRefreshToken(null, rt)
 .ExecuteAsync()
 .ConfigureAwait(false);
```

You will see an access token and ID token returned in your `AuthenticationResult` while your new refresh token is stored in the cache.

You can also use this method for various integration scenarios where you have a refresh token available.

## Next steps

You can find more information about the scopes in [Scopes, permissions, and consent in the Microsoft identity platform endpoint](#)

# Differences between MSAL JS and ADAL JS

10/23/2019 • 3 minutes to read • [Edit Online](#)

Both Microsoft Authentication Library for JavaScript (MSAL.js) and Azure AD Authentication Library for JavaScript (ADAL.js) are used to authenticate Azure AD entities and request tokens from Azure AD. Up until now, most developers have worked with Azure AD for developers (v1.0) to authenticate Azure AD identities (work and school accounts) by requesting tokens using ADAL. Now, using MSAL.js, you can authenticate a broader set of Microsoft identities (Azure AD identities and Microsoft accounts, and social and local accounts through Azure AD B2C) through Microsoft identity platform (v2.0).

This article describes how to choose between the Microsoft Authentication Library for JavaScript (MSAL.js) and Azure AD Authentication Library for JavaScript (ADAL.js) and compares the two libraries.

## Choosing between ADAL.js and MSAL.js

In most cases you want to use the Microsoft identity platform and MSAL.js, which is the latest generation of Microsoft authentication libraries. Using MSAL.js, you acquire tokens for users signing in to your application with Azure AD (work and school accounts), Microsoft (personal) accounts (MSA), or Azure AD B2C.

If you are already familiar with the v1.0 endpoint (and ADAL.js), you might want to read [What's different about the v2.0 endpoint?](#).

However, you still need to use ADAL.js if your application needs to sign in users with earlier versions of [Active Directory Federation Services \(ADFS\)](#).

## Key differences in authentication with MSAL.js

### Core API

- ADAL.js uses `AuthenticationContext` as the representation of an instance of your application's connection to the authorization server or identity provider through an authority URL. On the contrary, MSAL.js API is designed around user agent client application(a form of public client application in which the client code is executed in a user-agent such as a web browser). It provides the `UserAgentApplication` class to represent an instance of the application's authentication context with the authorization server. For more details, see [Initialize using MSAL.js](#).
- In ADAL.js, the methods to acquire tokens are associated with a single authority set in the `AuthenticationContext`. In MSAL.js, the acquire token requests can take different authority values than what is set in the `UserAgentApplication`. This allows MSAL.js to acquire and cache tokens separately for multiple tenants and user accounts in the same application.
- The method to acquire and renew tokens silently without prompting users is named `acquireToken` in ADAL.js. In MSAL.js, this method is named `acquireTokenSilent` to be more descriptive of this functionality.

### Authority value common

In v1.0, using the `https://login.microsoftonline.com/common` authority will allow users to sign in with any Azure AD account (for any organization).

In v2.0, using the `https://login.microsoftonline.com/common` authority, will allow users to sign in with any Azure AD organization account or a Microsoft personal account (MSA). To restrict the sign in to only Azure AD accounts (same behavior as with ADAL.js), you need to use `https://login.microsoftonline.com/organizations`. For details, see the `authority` config option in [Initialize using MSAL.js](#).

## Scopes for acquiring tokens

- Scope instead of resource parameter in authentication requests to acquire tokens

v2.0 protocol uses scopes instead of resource in the requests. In other words, when your application needs to request tokens with permissions for a resource such as MS Graph, the difference in values passed to the library methods is as follows:

v1.0: resource = https://graph.microsoft.com

v2.0: scope = https://graph.microsoft.com/User.Read

You can request scopes for any resource API using the URI of the API in this format: appidURI/scope For example: https://mytenant.onmicrosoft.com/myapi/api.read

Only for the MS Graph API, a scope value `user.read` maps to https://graph.microsoft.com/User.Read and can be used interchangeably.

```
var request = {
 scopes = ["User.Read"];
};

acquireTokenPopup(request);
```

- Dynamic scopes for incremental consent.

When building applications using v1.0, you needed to register the full set of permissions(static scopes) required by the application for the user to consent to at the time of login. In v2.0, you can use the scope parameter to request the permissions at the time you want them. These are called dynamic scopes. This allows the user to provide incremental consent to scopes. So if at the beginning you just want the user to sign in to your application and you don't need any kind of access, you can do so. If later you need the ability to read the calendar of the user, you can then request the calendar scope in the acquireToken methods and get the user's consent. For example:

```
var request = {
 scopes = ["https://graph.microsoft.com/User.Read", "https://graph.microsoft.com/Calendar.Read"];
};

acquireTokenPopup(request);
```

- Scopes for V1.0 APIs

When getting tokens for V1.0 APIs using MSAL.js, you can request all the static scopes registered on the API by appending `.default` to the App ID URI of the API as scope. For example:

```
var request = {
 scopes = [appidURI + "/.default"];
};

acquireTokenPopup(request);
```

## Next steps

For more information, refer to [v1.0 and v2.0 comparison](#).

# ADAL to MSAL migration guide for Android

11/6/2019 • 11 minutes to read • [Edit Online](#)

This article highlights changes you need to make to migrate an app that uses the Azure Active Directory Authentication Library (ADAL) to use the Microsoft Authentication Library (MSAL).

## Difference highlights

ADAL works with the Azure Active Directory v1.0 endpoint. The Microsoft Authentication Library (MSAL) works with the Microsoft identity platform--formerly known as the Azure Active Directory v2.0 endpoint. The Microsoft identity platform differs from Azure Active Directory v1.0 in that it:

Supports:

- Organizational Identity (Azure Active Directory)
- Non-organizational identities such as Outlook.com, Xbox Live, and so on
- (B2C Only) Federated login with Google, Facebook, Twitter, and Amazon
- Is standards compatible with:
  - OAuth v2.0
  - OpenID Connect (OIDC)

The MSAL public API introduces important changes, including:

- A new model for accessing tokens:
  - ADAL provides access to tokens via the `AuthenticationContext`, which represents the server. MSAL provides access to tokens via the `PublicClientApplication`, which represents the client. Client developers don't need to create a new `PublicClientApplication` instance for every Authority they need to interact with. Only one `PublicClientApplication` configuration is required.
  - Support for requesting access tokens using scopes in addition to resource identifiers.
  - Support for incremental consent. Developers can request scopes as the user accesses more and more functionality in the app, including those not included during app registration.
  - Authorities are no longer validated at run-time. Instead, the developer declares a list of 'known authorities' during development.
- Token API changes:
  - In ADAL, `AcquireToken()` first makes a silent request. Failing that, it makes an interactive request. This behavior resulted in some developers relying only on `AcquireToken`, which resulted in the user being unexpectedly prompted for credentials at times. MSAL requires developers be intentional about when the user receives a UI prompt.
    - `AcquireTokenSilent` always results in a silent request that either succeeds or fails.
    - `AcquireToken` always results in a request that prompts the user via UI.
- MSAL supports sign in from either a default browser or an embedded web view:
  - By default, the default browser on the device is used. This allows MSAL to use authentication state (cookies) that may already be present for one or more signed in accounts. If no authentication state is present, authenticating during authorization via MSAL results in authentication state (cookies) being created for the benefit of other web applications that will be used in the same browser.
- New exception Model:

- Exceptions more clearly define the type of error that occurred and what the developer needs to do to resolve it.
- MSAL supports parameter objects for `AcquireToken` and `AcquireTokenSilent` calls.
- MSAL supports declarative configuration for:
  - Client ID, Redirect URI.
  - Embedded vs Default Browser
  - Authorities
  - HTTP settings such as read and connection timeout

## Your app registration and migration to MSAL

You don't need to change your existing app registration to use MSAL. If you want to take advantage of incremental/progressive consent, you may need to review the registration to identify the specific scopes that you want to request incrementally. More information on scopes and incremental consent follows.

In your app registration in the portal, you will see an **API permissions** tab. This provides a list of the APIs and permissions (scopes) that your app is currently configured to request access to. It also shows a list of the scope names associated with each API permission.

### User consent

With ADAL and the AAD v1 endpoint, user consent to resources they own was granted on first use. With MSAL and the Microsoft identity platform, consent can be requested incrementally. Incremental consent is useful for permissions that a user may consider high privilege, or may otherwise question if not provided with a clear explanation of why the permission is required. In ADAL, those permissions may have resulted in the user abandoning signing in to your app.

#### TIP

We recommend the use of incremental consent in scenarios where you need to provide additional context to your user about why your app needs a permission.

### Admin consent

Organization administrators can consent to permissions your application requires on behalf of all of the members of their organization. Some organizations only allow admins to consent to applications. Admin consent requires that you include all API permissions and scopes used by your application in your app registration.

#### TIP

Even though you can request a scope using MSAL for something not included in your app registration, we recommend that you update your app registration to include all resources and scopes that a user could ever grant permission to.

## Migrating from resource IDs to scopes

### Authenticate and request authorization for all permissions on first use

If you're currently using ADAL and don't need to use incremental consent, the simplest way to start using MSAL is to make an `acquireToken` request using the new `AcquireTokenParameter` object and setting the resource ID value.

#### Caution

It's not possible to set both scopes and a resource id. Attempting to set both will result in an `IllegalArgumentException`.

This will result in the same v1 behavior that you are used. All permissions requested in your app registration are requested from the user during their first interaction.

## Authenticate and request permissions only as needed

To take advantage of incremental consent, make a list of permissions (scopes) that your app uses from your app registration, and organize them into two lists based on:

- Which scopes you want to request during the user's first interaction with your app during sign-in.
- The permissions that are associated with an important feature of your app that you will also need to explain to the user.

Once you've organized the scopes, organize each list by which resource (API) you want to request a token for. As well as any other scopes that you wish the user to authorize at the same time.

The parameters object used to make your request to MSAL supports:

- `Scope` : The list of scopes that you want to request authorization for and receive an access token.
- `ExtraScopesToConsent` : An additional list of scopes that you want to request authorization for while you're requesting an access token for another resource. This list of scopes allows you to minimize the number of times that you need to request user authorization. Which means fewer user authorization or consent prompts.

## Migrate from AuthenticationContext to PublicClientApplications

### Constructing PublicClientApplication

When you use MSAL, you instantiate a `PublicClientApplication`. This object models your app identity and is used to make requests to one or more authorities. With this object you will configure your client identity, redirect URI, default authority, whether to use the device browser vs. embedded web view, the log level, and more.

You can declaratively configure this object with JSON, which you either provide as a file or store as a resource within your APK.

Although this object is not a singleton, internally it uses shared `Executors` for both interactive and silent requests.

### Business to Business

In ADAL, every organization that you request access tokens from requires a separate instance of the `AuthenticationContext`. In MSAL, this is no longer a requirement. You can specify the authority from which you want to request a token as part of your silent or interactive request.

### Migrate from authority validation to known authorities

MSAL does not have a flag to enable or disable authority validation. Authority validation is a feature in ADAL, and in the early releases of MSAL, that prevents your code from requesting tokens from a potentially malicious authority. MSAL now retrieves a list of authorities known to Microsoft and merges that list with the authorities that you've specified in your configuration.

#### TIP

If you're an Azure Business to Consumer (B2C) user, this means you no longer have to disable authority validation. Instead, include each of the your supported Azure AD B2C policies as authorities in your MSAL configuration.

If you attempt to use an authority that isn't known to Microsoft, and isn't included in your configuration, you will get an `UnknownAuthorityException`.

### Logging

You can now declaratively configure logging as part of your configuration, like this:

```
"logging": {
 "pii_enabled": false,
 "log_level": "WARNING",
 "logcat_enabled": true
}
```

## Migrate from UserInfo to Account

In ADAL, the `AuthenticationResult` provides a `UserInfo` object used to retrieve information about the authenticated account. The term "user", which meant a human or software agent, was applied in a way that made it difficult to communicate that some apps support a single user (whether a human or software agent) that has multiple accounts.

Consider a bank account. You may have more than one account at more than one financial institution. When you open an account, you (the user) are issued credentials, such as an ATM Card & PIN, that are used to access your balance, bill payments, and so on, for each account. Those credentials can only be used at the financial institution that issued them.

By analogy, like accounts at a financial institution, accounts in the Microsoft identity platform are accessed using credentials. Those credentials are either registered with, or issued by, Microsoft. Or by Microsoft on behalf of an organization.

Where the Microsoft identity platform differs from a financial institution, in this analogy, is that the Microsoft identity platform provides a framework that allows a user to use one account, and its associated credentials, to access resources that belong to multiple individuals and organizations. This is like being able to use a card issued by one bank, at yet another financial institution. This works because all of the organizations in question are using the Microsoft identity platform, which allows one account to be used across multiple organizations. Here's an example:

Sam works for Contoso.com but manages Azure virtual machines that belong to Fabrikam.com. For Sam to manage Fabrikam's virtual machines, he needs to be authorized to access them. This access can be granted by adding Sam's account to Fabrikam.com, and granting his account a role that allows him to work with the virtual machines. This would be done with the Azure portal.

Adding Sam's Contoso.com account as a member of Fabrikam.com would result in the creation of a new record in Fabrikam.com's Azure Active Directory for Sam. Sam's record in Azure Active Directory is known as a user object. In this case, that user object would point back to Sam's user object in Contoso.com. Sam's Fabrikam user object is the local representation of Sam, and would be used to store information about the account associated with Sam in the context of Fabrikam.com. In Contoso.com, Sam's title is Senior DevOps Consultant. In Fabrikam, Sam's title is Contractor-Virtual Machines. In Contoso.com, Sam is not responsible, nor authorized, to manage virtual machines. In Fabrikam.com, that's his only job function. Yet Sam still only has one set of credentials to keep track of, which are the credentials issued by Contoso.com.

Once a successful `acquireToken` call is made, you will see a reference to an `IAccount` object that can be used in later `acquireTokenSilent` requests.

### **IMultiTenantAccount**

If you have an app that accesses claims about an account from each of the tenants in which the account is represented, you can cast `IAccount` objects to `IMultiTenantAccount`. This interface provides a map of `ITenantProfiles`, keyed by tenant ID, that allows you to access the claims that belong to the account in each of the tenants you've requested a token from, relative to the current account.

The claims at the root of the `IAccount` and `IMultiTenantAccount` always contain the claims from the home tenant. If you have not yet made a request for a token within the home tenant, this collection will be empty.

## Other changes

### Use the new AuthenticationCallback

```
// Existing ADAL Interface
public interface AuthenticationCallback<T> {

 /**
 * This will have the token info.
 *
 * @param result returns <T>
 */
 void onSuccess(T result);

 /**
 * Sends error information. This can be user related error or server error.
 * Cancellation error is AuthenticationCancelError.
 *
 * @param exc return {@link Exception}
 */
 void onError(Exception exc);
}
```

```

// New Interface for Interactive AcquireToken
public interface AuthenticationCallback {

 /**
 * Authentication finishes successfully.
 *
 * @param authenticationResult {@link IAuthenticationResult} that contains the success response.
 */
 void onSuccess(final IAuthenticationResult authenticationResult);

 /**
 * Error occurs during the authentication.
 *
 * @param exception The {@link MsalException} contains the error code, error message and cause if applicable. The exception
 * returned in the callback could be {@link MsalClientException}, {@link
 * MsalServiceException}
 */
 void onError(final MsalException exception);

 /**
 * Will be called if user cancels the flow.
 */
 void onCancel();
}

// New Interface for Silent AcquireToken
public interface SilentAuthenticationCallback {

 /**
 * Authentication finishes successfully.
 *
 * @param authenticationResult {@link IAuthenticationResult} that contains the success response.
 */
 void onSuccess(final IAuthenticationResult authenticationResult);

 /**
 * Error occurs during the authentication.
 *
 * @param exception The {@link MsalException} contains the error code, error message and cause if applicable. The exception
 * returned in the callback could be {@link MsalClientException}, {@link
 * MsalServiceException} or
 * {@link MsalUiRequiredException}.
 */
 void onError(final MsalException exception);
}

```

## Migrate to the new exceptions

In ADAL, there's one type of exception, `AuthenticationException`, which includes a method for retrieving the `ADALError` enum value. In MSAL, there's a hierarchy of exceptions, and each has its own set of associated specific error codes.

### List of MSAL Exceptions

EXCEPTION	DESCRIPTION
<code>MsalException</code>	Default checked exception thrown by MSAL.

EXCEPTION	DESCRIPTION
MsalClientException	Thrown if the error is client side.
MsalArgumentException	Thrown if one or more inputs arguments are invalid.
MsalClientException	Thrown if the error is client side.
MsalServiceException	Thrown if the error is server side.
MsalUserCancelException	Thrown if the user canceled the authentication flow.
MsalUiRequiredException	Thrown if the token can't be refreshed silently.
MsalDeclinedScopeException	Thrown if one or more requested scopes were declined by the server.
MsalIntuneAppProtectionPolicyRequiredException	Thrown if the resource has MAMCA protection policy enabled.

## ADALError to MsalException ErrorCode

### ADAL Logging to MSAL Logging

```
// Legacy Interface
StringBuilder logs = new StringBuilder();
Logger.getInstance().setExternalLogger(new ILogger() {
 @Override
 public void Log(String tag, String message, String additionalMessage, LogLevel logLevel, ADALError
errorCode) {
 logs.append(message).append('\n');
 }
});
```

```
// New interface
StringBuilder logs = new StringBuilder();
Logger.getInstance().setExternalLogger(new ILoggerCallback() {
 @Override
 public void log(String tag, Logger.LogLevel logLevel, String message, boolean containsPII) {
 logs.append(message).append('\n');
 }
});

// New Log Levels:
public enum LogLevel
{
 /**
 * Error level logging.
 */
 ERROR,
 /**
 * Warning level logging.
 */
 WARNING,
 /**
 * Info level logging.
 */
 INFO,
 /**
 * Verbose level logging.
 */
 VERBOSE
}
```

# Migrate applications to MSAL for iOS and macOS

10/30/2019 • 15 minutes to read • [Edit Online](#)

The Azure Active Directory Authentication Library ([ADAL Objective-C](#)) was created to work with Azure Active Directory accounts via the v1.0 endpoint.

The Microsoft Authentication Library for iOS and macOS (MSAL) is built to work with all Microsoft identities such as Azure Active Directory (Azure AD) accounts, personal Microsoft accounts, and Azure AD B2C accounts via the Microsoft identity platform (formally the Azure AD v2.0 endpoint).

The Microsoft identity platform has a few key differences with Azure Active Directory v1.0. This article highlights these differences and provides guidance to migrate an app from ADAL to MSAL.

## ADAL and MSAL app capability differences

### Who can sign in

- ADAL only supports work and school accounts--also known as Azure AD accounts.
- MSAL supports personal Microsoft accounts (MSA accounts) such as Hotmail.com, Outlook.com, and Live.com.
- MSAL supports work and school accounts, and Azure AD B2C accounts.

### Standards compliance

- The Microsoft identity Platform endpoint follows OAuth 2.0 and OpenId Connect standards.

### Incremental and dynamic consent

- The Azure Active Directory v1.0 endpoint requires that all permissions be declared in advance during application registration. This means those permissions are static.
- The Microsoft identity platform allows you to request permissions dynamically. Apps can ask for permissions only as needed and request more as the app needs them.

For more about differences between Azure Active Directory v1.0 and the Microsoft identity platform, see [Why update to Microsoft identity platform \(v2.0\)?](#).

## ADAL and MSAL library differences

The MSAL public API reflects a few key differences between Azure AD v1.0 and the Microsoft identity platform.

### MSALPublicClientApplication instead of ADAuthenticationContext

`ADAuthenticationContext` is the first object an ADAL app creates. It represents an instantiation of ADAL. Apps create a new instance of `ADAuthenticationContext` for each Azure Active Directory cloud and tenant (authority) combination. The same `ADAuthenticationContext` can be used to get tokens for multiple public client applications.

In MSAL, the main interaction is through an `MSALPublicClientApplication` object, which is modeled after [OAuth 2.0 Public Client](#). One instance of `MSALPublicClientApplication` can be used to interact with multiple AAD clouds, and tenants, without needing to create a new instance for each authority. For most apps, one `MSALPublicClientApplication` instance is sufficient.

### Scopes instead of resources

In ADAL, an app had to provide a *resource* identifier like `https://graph.microsoft.com` to acquire tokens from the Azure Active Directory v1.0 endpoint. A resource can define a number of scopes, or oAuth2Permissions in the app manifest, that it understands. This allowed client apps to request tokens from that resource for a certain set of scopes pre-defined during app registration.

In MSAL, instead of a single resource identifier, apps provide a set of scopes per request. A scope is a resource identifier followed by a permission name in the form resource/permission. For example,

```
https://graph.microsoft.com/user.read
```

There are two ways to provide scopes in MSAL:

- Provide a list of all the permissions your app needs. For example:

```
[@[@"https://graph.microsoft.com/directory.read", @"https://graph.microsoft.com/directory.write"]]
```

In this case, the app requests the `directory.read` and `directory.write` permissions. The user will be asked to consent for those permissions if they haven't consented to them before for this app. The application might also receive additional permissions that the user has already consented to for the application. The user will only be prompted to consent for new permissions, or permissions that haven't been granted.

- The `/.default` scope.

This is the built-in scope for every application. It refers to the static list of permissions configured when the application was registered. Its behavior is similar to that of `resource`. This can be useful when migrating to ensure that a similar set of scopes and user experience is maintained.

To use the `/.default` scope, append `/.default` to the resource identifier. For example:

```
https://graph.microsoft.com/.default
```

If your resource ends with a slash (`/`), you should still append `/.default`, including the leading forward slash, resulting in a scope that has a double forward slash (`//`) in it.

You can read more information about using the `"/.default"` scope [here](#)

### Supporting different WebView types & browsers

ADAL only supports UIWebView/WKWebView for iOS, and WebView for macOS. MSAL for iOS supports more options for displaying web content when requesting an authorization code, and no longer supports `UIWebView`; which can improve the user experience and security.

By default, MSAL on iOS uses `ASWebAuthenticationSession`, which is the web component Apple recommends for authentication on iOS 12+ devices. It provides Single Sign-On (SSO) benefits through cookie sharing between apps and the Safari browser.

You can choose to use a different web component depending on app requirements and the end-user experience you want. See [supported web view types](#) for more options.

When migrating from ADAL to MSAL, `WKWebView` provides the user experience most similar to ADAL on iOS and macOS. We encourage you to migrate to `ASWebAuthenticationSession` on iOS, if possible. For macOS, we encourage you to use `WKWebView`.

### Account management API differences

When you call the ADAL methods `acquireToken()` or `acquireTokenSilent()`, you receive an `ADUserInformation` object containing a list of claims from the `id_token` that represents the account being authenticated. Additionally, `ADUserInformation` returns a `userId` based on the `upn` claim. After initial interactive token acquisition, ADAL expects developer to provide `userId` in all silent calls.

ADAL doesn't provide an API to retrieve known user identities. It relies on the app to save and manage those accounts.

MSAL provides a set of APIs to list all accounts known to MSAL without having to acquire a token.

Like ADAL, MSAL returns account information that holds a list of claims from the `id_token`. It's part of the `MSALAccount` object inside the `MSALResult` object.

MSAL provides a set of APIs to remove accounts, making the removed accounts inaccessible to the app. After the

account is removed, later token acquisition calls will prompt the user to do interactive token acquisition. Account removal only applies to the client application that started it, and doesn't remove the account from the other apps running on the device or from the system browser. This ensures that the user continues to have a SSO experience on the device even after signing out of an individual app.

Additionally, MSAL also returns an account identifier that can be used to request a token silently later. However, the account identifier (accessible through `identifier` property in the `MSALAccount` object) isn't displayable and you can't assume what format it is in nor should you try to interpret or parse it.

## Migrating the account cache

When migrating from ADAL, apps normally store ADAL's `userId`, which doesn't have the `identifier` required by MSAL. As a one-time migration step, an app can query an MSAL account using ADAL's `userId` with the following API:

```
- (nullable MSALAccount *)accountForUsername:(nonnull NSString *)username error:(NSError * _Nullable __autoreleasing * _Nullable)error;
```

This API reads both MSAL's and ADAL's cache to find the account by ADAL `userId` (UPN).

If the account is found, the developer should use the account to do silent token acquisition. The first silent token acquisition will effectively upgrade the account, and the developer will get a MSAL compatible account identifier in the MSAL result (`identifier`). After that, only `identifier` should be used for account lookups by using the following API:

```
- (nullable MSALAccount *)accountForIdentifier:(nonnull NSString *)identifier error:(NSError * _Nullable __autoreleasing * _Nullable)error;
```

Although it's possible to continue using ADAL's `userId` for all operations in MSAL, since `userId` is based on UPN, it's subject to multiple limitations that result in a bad user experience. For example, if the UPN changes, the user has to sign in again. We recommend all apps use the non-displayable account `identifier` for all operations.

Read more about [cache state migration](#).

## Token acquisition changes

MSAL introduces some token acquisition call changes:

- Like ADAL, `acquireTokenSilent` always results in a silent request.
- Unlike ADAL, `acquireToken` always results in user actionable UI either through the web view or the Microsoft Authenticator app. Depending on the SSO state inside webview/Microsoft Authenticator, the user may be prompted to enter their credentials.
- In ADAL, `acquireToken` with `AD_PROMPT_AUTO` first tries silent token acquisition, and only shows UI if the silent request fails. In MSAL, this logic can be achieved by first calling `acquireTokenSilent` and only calling `acquireToken` if silent acquisition fails. This allows developers to customize user experience before starting interactive token acquisition.

## Error handling differences

MSAL provides more clarity between errors that can be handled by your app and those that require intervention by the user. There are a limited number of errors developer must handle:

- `MSALErrorInteractionRequired` : The user must do an interactive request. This can be caused for various reasons such as an expired authentication session, Conditional Access policy has changed, a refresh token expired or was revoked, there are no valid tokens in the cache, and so on.
- `MSALErrorServerDeclinedScopes` : The request wasn't fully completed and some scopes weren't granted access. This can be caused by a user declining consent to one or more scopes.

Handling all other errors in the `MSALError` list is optional. You could use the information in those errors to improve the user experience.

See [Handling exceptions and errors using MSAL](#) for more about MSAL error handling.

## Broker support

MSAL, starting with version 0.3.0, provides support for brokered authentication using the Microsoft Authenticator app. Microsoft Authenticator also enables support for Conditional Access scenarios. Examples of Conditional Access scenarios include device compliance policies that require the user to enroll the device through Intune or register with AAD to get a token. And Mobile Application Management (MAM) Conditional Access policies, which require proof of compliance before your app can get a token.

To enable broker for your application:

1. Register a broker compatible redirect URI format for the application. The broker compatible redirect URI format is `msauth.<app.bundle.id>://auth`. Replace `<app.bundle.id>` with your application's bundle ID. If you're migrating from ADAL and your application was already broker capable, there's nothing extra you need to do. Your previous redirect URI is fully compatible with MSAL, so you can skip to step 3.
2. Add your application's redirect URI scheme to your `info.plist` file. For the default MSAL redirect URI, the format is `msauth.<app.bundle.id>`. For example:

```
<key>CFBundleURLSchemes</key>
<array>
 <string>msauth.<app.bundle.id></string>
</array>
```

3. Add following schemes to your app's `Info.plist` under `LSApplicationQueriesSchemes`:

```
<key>LSApplicationQueriesSchemes</key>
<array>
 <string>msauthv2</string>
 <string>msauthv3</string>
</array>
```

4. Add the following to your `AppDelegate.m` file to handle callbacks: Objective-C:

```
- (BOOL)application:(UIApplication *)app openURL:(NSURL *)url options:(NSDictionary<NSString *,id> *)options{
{
 return [MSALPublicClientApplication handleMSALResponse:url
sourceApplication:options[UIApplicationOpenURLOptionsSourceApplicationKey]];
}
```

Swift:

```
func application(_ app: UIApplication, open url: URL, options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
 return MSALPublicClientApplication.handleMSALResponse(url, sourceApplication:
options[UIApplication.OpenURLOptionsKey.sourceApplication] as? String)
}
```

## Business to business (B2B)

In ADAL, you create separate instances of `ADAuthenticationContext` for each tenant that the app requests tokens for. This is no longer a requirement in MSAL. In MSAL, you can create a single instance of `MSALPublicClientApplication` and use it for any AAD cloud and organization by specifying a different authority for `acquireToken` and `acquireTokenSilent` calls.

## SSO in partnership with other SDKs

MSAL for iOS can achieve SSO via a unified cache with the following SDKs:

- ADAL Objective-C 2.7.x+
- MSAL.NET for Xamarin 2.4.x+
- ADAL.NET for Xamarin 4.4.x+

SSO is achieved via iOS keychain sharing and is only available between apps published from the same Apple Developer account.

SSO through iOS keychain sharing is the only silent SSO type.

On macOS, MSAL can achieve SSO with other MSAL for iOS and macOS based applications and ADAL Objective-C-based applications.

MSAL on iOS also supports two other types of SSO:

- SSO through the web browser. MSAL for iOS supports `ASWebAuthenticationSession`, which provides SSO through cookies shared between other apps on the device and specifically the Safari browser.
- SSO through an Authentication broker. On an iOS device, Microsoft Authenticator acts as the Authentication broker. It can follow Conditional Access policies such as requiring a compliant device, and provides SSO for registered devices. MSAL SDKs starting with version 0.3.0 support a broker by default.

## Intune MAM SDK

The [Intune MAM SDK](#) supports MSAL for iOS starting with version [11.1.2](#).

## MSAL and ADAL in the same app

ADAL version 2.7.0, and above, can't coexist with MSAL in the same application. The main reason is because of the shared submodule common code. Because Objective-C doesn't support namespaces, if you add both ADAL and MSAL frameworks to your application, there will be two instances of the same class. There's no guarantee for which one gets picked at runtime. If both SDKs are using same version of the conflicting class, your app may still work. However, if it's a different version, your app might experience unexpected crashes that are difficult to diagnose.

Running ADAL and MSAL in the same production application isn't supported. However, if you're just testing and migrating your users from ADAL Objective-C to MSAL for iOS and macOS, you can continue using [ADAL Objective-C 2.6.10](#). It's the only version that works with MSAL in the same application. There will be no new feature updates for this ADAL version, so it should be only used for migration and testing purposes. Your app shouldn't rely on ADAL and MSAL coexistence long term.

ADAL and MSAL coexistence in the same application isn't supported. ADAL and MSAL coexistence between multiple applications is fully supported.

## Practical migration steps

### App registration migration

You don't need to change your existing AAD application to switch to MSAL and enable AAD accounts. However, if your ADAL-based application doesn't support brokered authentication, you'll need to register a new redirect URI for the application before you can switch to MSAL.

The redirect URI should be in this format: `msauth.<app.bundle.id>://auth`. Replace `<app.bundle.id>` with your application's bundle ID. Specify the redirect URI in the [Azure portal](#).

For iOS only, to support cert-based authentication, an additional redirect URI needs to be registered in your application and the Azure portal in the following format: `msauth://code/<broker-redirect-uri-in-url-encoded-form>`.

For example, `msauth://code/msauth.com.microsoft.mybundleId%3A%2F%2Fauth`

We recommend all apps register both redirect URIs.

If you wish to add support for incremental consent, select the APIs and permissions your app is configured to request access to in your app registration under the **API permissions** tab.

If you're migrating from ADAL and want to support both AAD and MSA accounts, your existing application registration needs to be updated to support both. We don't recommend you update your existing production app to support both AAD and MSA right away. Instead, create another client ID that supports both AAD and MSA for testing, and after you've verified that all scenarios work, update the existing app.

### Add MSAL to your app

You can add MSAL SDK to your app using your preferred package management tool. See [detailed instructions here](#).

### Update your app's Info.plist file

For iOS only, add your application's redirect URI scheme to your info.plist file. For ADAL broker compatible apps, it should be there already. The default MSAL redirect URI scheme will be in the format: `msauth.<app.bundle.id>`.

```
<key>CFBundleURLSchemes</key>
<array>
 <string>msauth.<app.bundle.id></string>
</array>
```

Add following schemes to your app's Info.plist under `LSApplicationQueriesSchemes`.

```
<key>LSApplicationQueriesSchemes</key>
<array>
 <string>msauthv2</string>
 <string>msauthv3</string>
</array>
```

### Update your AppDelegate code

For iOS only, add the following to your AppDelegate.m file:

Objective-C:

```
- (BOOL)application:(UIApplication *)app openURL:(NSURL *)url options:(NSDictionary<NSString *,id> *)options{
{
 return [MSALPublicClientApplication handleMSALResponse:url
sourceApplication:options[UIApplicationOpenURLOptionsSourceApplicationKey]];
}
```

Swift:

```
func application(_ app: UIApplication, open url: URL, options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
 return MSALPublicClientApplication.handleMSALResponse(url, sourceApplication:
options[UIApplication.OpenURLOptionsKey.sourceApplication] as? String)
}
```

If you are using Xcode 11, you should place MSAL callback into the `SceneDelegate` file instead. If you support both UISceneDelegate and UIApplicationDelegate for compatibility with older iOS, MSAL callback would need to

be placed into both files.

Objective-C:

```
- (void)scene:(UIScene *)scene openURLContexts:(NSSet<UIOpenURLContext *> *)URLContexts
{
 UIOpenURLContext *context = URLContexts.anyObject;
 NSURL *url = context.URL;
 NSString *sourceApplication = context.options.sourceApplication;

 [MSALPublicClientApplication handleMSALResponse:url sourceApplication:sourceApplication];
}
```

Swift:

```
func scene(_ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>) {

 guard let urlContext = URLContexts.first else {
 return
 }

 let url = urlContext.url
 let sourceApp = urlContext.options.sourceApplication

 MSALPublicClientApplication.handleMSALResponse(url, sourceApplication: sourceApp)
}
```

This allows MSAL to handle responses from the broker and web component. This wasn't necessary in ADAL since it "swizzled" app delegate methods automatically. Adding it manually is less error prone and gives the application more control.

### Enable token caching

By default, MSAL caches your app's tokens in the iOS or macOS keychain.

To enable token caching:

1. Ensure your application is properly signed
2. Go to your Xcode Project Settings > **Capabilities tab** > **Enable Keychain Sharing**
3. Click + and enter a following **Keychain Groups** entry: 3.a For iOS, enter `com.microsoft.adalcache` 3.b For macOS enter `com.microsoft.identity.universalstorage`

### Create MSALPublicClientApplication and switch to its acquireToken and acquireTokenSilent calls

You can create `MSALPublicClientApplication` using following code:

Objective-C:

```
NSError *error = nil;
MSALPublicClientApplicationConfig *configuration = [[MSALPublicClientApplicationConfig alloc]
initWithClientId:@"<your-client-id-here>"];

MSALPublicClientApplication *application =
[[MSALPublicClientApplication alloc] initWithConfiguration:configuration
 error:&error];
```

Swift:

```

let config = MSALPublicClientApplicationConfig(clientId: "<your-client-id-here>")
do {
 let application = try MSALPublicClientApplication(configuration: config)
 // continue on with application

} catch let error as NSError {
 // handle error here
}

```

Then call the account management API to see if there are any accounts in the cache:

Objective-C:

```

NSString *accountIdentifier = nil /*previously saved MSAL account identifier */;
NSError *error = nil;
MSALAccount *account = [application accountForIdentifier:accountIdentifier error:&error];

```

Swift:

```

// definitions that need to be initialized
let application: MSALPublicClientApplication!
let accountIdentifier: String! /*previously saved MSAL account identifier */

do {
 let account = try application.account(forIdentifier: accountIdentifier)
 // continue with account usage
} catch let error as NSError {
 // handle error here
}

```

or read all of the accounts:

Objective-C:

```

NSError *error = nil;
NSArray<MSALAccount *> *accounts = [application allAccounts:&error];

```

Swift:

```

let application: MSALPublicClientApplication!
do {
 let accounts = try application.allAccounts()
 // continue with account usage
} catch let error as NSError {
 // handle error here
}

```

If an account is found, call the MSAL `acquireTokenSilent` API:

Objective-C:

```

MSALSilentTokenParameters *silentParameters = [[MSALSilentTokenParameters alloc] initWithScopes:@[@"<your-
resource-here>/.default"] account:account];

[application acquireTokenSilentWithParameters:silentParameters
 completionBlock:^(MSALResult *result, NSError *error)
{
 if (result)
 {
 NSString *accessToken = result.accessToken;
 // Use your token
 }
 else
 {
 // Check the error
 if ([error.domain isEqualToString:MSALErrorDomain] && error.code == MSALErrorInteractionRequired)
 {
 // Interactive auth will be required
 }

 // Other errors may require trying again later, or reporting authentication problems to the user
 }
}];

```

Swift:

```

let application: MSALPublicClientApplication!
let account: MSALAccount!

let silentParameters = MSALSilentTokenParameters(scopes: ["<your-resource-here>/.default"],
 account: account)
application.acquireTokenSilent(with: silentParameters) {
 (result: MSALResult?, error: Error?) in
 if let accessToken = result?.accessToken {
 // use accessToken
 }
 else {
 // Check the error
 guard let error = error else {
 XCTAssertTrue("callback should contain a valid result or error")
 return
 }

 let nsError = error as NSError
 if (nsError.domain == MSALErrorDomain
 && nsError.code == MSALError.interactionRequired.rawValue) {
 // Interactive auth will be required
 }

 // Other errors may require trying again later, or reporting authentication problems to the user
 }
}

```

## Next steps

Learn more about [Authentication flows and application scenarios](#)

# ADAL to MSAL migration guide for Java

11/10/2019 • 4 minutes to read • [Edit Online](#)

This article highlights changes you need to make to migrate an app that uses the Azure Active Directory Authentication Library (ADAL) to use the Microsoft Authentication Library (MSAL).

Both Microsoft Authentication Library for Java (MSAL4J) and Azure AD Authentication Library for Java (ADAL4J) are used to authenticate Azure AD entities and request tokens from Azure AD. Until now, most developers have worked with Azure AD for developers platform (v1.0) to authenticate Azure AD identities (work and school accounts) by requesting tokens using Azure AD Authentication Library (ADAL).

MSAL offers the following benefits:

- Because it uses the newer Microsoft identity platform endpoint, you can authenticate a broader set of Microsoft identities such as Azure AD identities, Microsoft accounts, and social and local accounts through Azure AD Business to Consumer (B2C).
- Your users will get the best single-sign-on experience.
- Your application can enable incremental consent, and supporting conditional access is easier.

MSAL for java (MSAL4J) is the auth library we recommend you use with the Microsoft identity platform. No new features will be implemented on ADAL4J. All efforts going forward are focused on improving MSAL.

## Differences

If you have been working with the Azure AD for developers (v1.0) endpoint (and ADAL4J), you might want to read [What's different about the Microsoft identity platform \(v2.0\) endpoint?](#)

## Scopes not resources

ADAL4J acquires tokens for resources whereas MSAL4J acquires tokens for scopes. A number of MSAL4J classes require a scopes parameter. This parameter is a list of strings that declare the desired permissions and resources that are requested. See [Microsoft Graph's scopes](#) to see example scopes.

## Core classes

In ADAL4J, the `AuthenticationContext` class represents your connection to the Security Token Service (STS), or authorization server, through an Authority. However, MSAL4J is designed around client applications. It provides two separate classes: `PublicClientApplication` and `ConfidentialClientApplication` to represent client applications. The latter, `ConfidentialClientApplication`, represents an application that is designed to securely maintain a secret such as an application identifier for a daemon app.

The following table shows how ADAL4J functions map to the new MSAL4J functions:

ADAL4J METHOD	MSAL4J METHOD
<code>acquireToken(String resource, ClientCredential credential, AuthenticationCallback callback)</code>	<code>acquireToken(ClientCredentialParameters)</code>
<code>acquireToken(String resource, ClientAssertion assertion, AuthenticationCallback callback)</code>	<code>acquireToken(ClientCredentialParameters)</code>

ADAL4J METHOD	MSAL4J METHOD
acquireToken(String resource, AsymmetricKeyCredential credential, AuthenticationCallback callback)	acquireToken(ClientCredentialParameters)
acquireToken(String resource, String clientId, String username, String password, AuthenticationCallback callback)	acquireToken(UserPasswordParameters)
acquireToken(String resource, String clientId, String username, String password=null, AuthenticationCallback callback)	acquireToken(IntegratedWindowsAuthenticationParameters)
acquireToken(String resource, UserAssertion userAssertion, ClientCredential credential, AuthenticationCallback callback)	acquireToken(OnBehalfOfParameters)
acquireTokenByAuthorizationCode()	acquireToken(AuthorizationCodeParameters)
acquireDeviceCode() and acquireTokenByDeviceCode()	acquireToken(DeviceCodeParameters)
acquireTokenByRefreshToken()	acquireTokenSilently(SilentParameters)

## IAccount instead of IUser

ADAL4J manipulated users. Although a user represents a single human or software agent, it can have one or more accounts in the Microsoft identity system. For example, a user may have several Azure AD, Azure AD B2C, or Microsoft personal accounts.

MSAL4J defines the concept of Account via the `IAccount` interface. This is a breaking change from ADAL4J, but it is a good one because it captures the fact that the same user can have several accounts, and perhaps even in different Azure AD directories. MSAL4J provides better information in guest scenarios because home account information is provided.

## Cache persistence

ADAL4J did not have support for token cache. MSAL4J adds a [token cache](#) to simplify managing token lifetimes by automatically refreshing expired tokens when possible and preventing unnecessary prompts for the user to provide credentials when possible.

## Common Authority

In v1.0, if you use the `https://login.microsoftonline.com/common` authority, users can sign in with any Azure Active Directory (AAD) account (for any organization).

If you use the `https://login.microsoftonline.com/common` authority in v2.0, users can sign in with any AAD organization, or even a Microsoft personal account (MSA). In MSAL4J, if you want to restrict login to any AAD account, you need to use the `https://login.microsoftonline.com/organizations` authority (which is the same behavior as with ADAL4J). To specify an authority, set the `authority` parameter in the [PublicClientApplicationBuilder](#) method when you create your `PublicClientApplication` class.

## v1.0 and v2.0 tokens

The v1.0 endpoint (used by ADAL) only emits v1.0 tokens.

The v2.0 endpoint (used by MSAL) can emit v1.0 and v2.0 tokens. A property of the application manifest of the Web API enables developers to choose which version of token is accepted. See `accessTokenAcceptedVersion` in the

[application manifest](#) reference documentation.

For more information about v1.0 and v2.0 tokens, see [Azure Active Directory access tokens](#).

## ADAL to MSAL migration

In ADAL4J, the refresh tokens were exposed--which allowed developers to cache them. They would then use `AcquireTokenByRefreshToken()` to enable solutions such as implementing long-running services that refresh dashboards on behalf of the user when the user is no longer connected.

MSAL4J does not expose refresh tokens for security reasons. Instead, MSAL handles refreshing tokens for you.

MSAL4J has an API that allows you to migrate refresh tokens you acquired with ADAL4J into the ClientApplication: `acquireToken(RefreshTokenParameters)`. With this method, you can provide the previously used refresh token along with any scopes (resources) you desire. The refresh token will be exchanged for a new one and cached for use by your application.

The following code snippet shows some migration code in a confidential client application:

```
String rt = GetCachedRefreshTokenForSignedInUser(); // Get refresh token from where you have them stored
Set<String> scopes = Collections.singleton("SCOPE_FOR_REFRESH_TOKEN");

RefreshTokenParameters parameters = RefreshTokenParameters.builder(scopes, rt).build();

PublicClientApplication app = PublicClientApplication.builder(CLIENT_ID) // ClientId for your application
 .authority(AUTHORITY) //plug in your authority
 .build();

IAuthenticationResult result = app.acquireToken(parameters);
```

The `IAuthenticationResult` returns an access token and ID token, while your new refresh token is stored in the cache. The application will also now contain an IAccount:

```
Set<IAccount> accounts = app.getAccounts().join();
```

To use the tokens that are now in the cache, call:

```
SilentParameters parameters = SilentParameters.builder(scope, accounts.iterator().next()).build();
IAuthenticationResult result = app.acquireToken(parameters);
```

# Migrate iOS applications that use Microsoft Authenticator from ADAL.NET to MSAL.NET

10/30/2019 • 4 minutes to read • [Edit Online](#)

You've been using the Azure Active Directory Authentication Library for .NET (ADAL.NET) and the iOS broker. Now it's time to migrate to the [Microsoft Authentication Library for .NET](#) (MSAL.NET), which supports the broker on iOS from release 4.3 onward.

Where should you start? This article helps you migrate your Xamarin iOS app from ADAL to MSAL.

## Prerequisites

This article assumes that you already have a Xamarin iOS app that's integrated with the iOS broker. If you don't, move directly to MSAL.NET and begin the broker implementation there. For information on how to invoke the iOS broker in MSAL.NET with a new application, see [this documentation](#).

## Background

### What are brokers?

Brokers are applications provided by Microsoft on Android and iOS. (See the [Microsoft Authenticator](#) app on iOS and Android, and the Intune Company Portal app on Android.)

They enable:

- Single sign-on.
- Device identification, which is required by some [Conditional Access policies](#). For more information, see [Device management](#).
- Application identification verification, which is also required in some enterprise scenarios. For more information, see [Intune mobile application management \(MAM\)](#).

## Migrate from ADAL to MSAL

### Step 1: Enable the broker

Current ADAL code:	MSAL counterpart:
--------------------	-------------------

In ADAL.NET, broker support was enabled on a per-authentication context basis. It's disabled by default. You had to set a

`useBroker` flag to true in the `PlatformParameters` constructor to call the broker:

```
public PlatformParameters(
 UIViewController callerViewController,
 bool useBroker)
```

Also, in the platform-specific code, in this example, in the page renderer for iOS, set the `useBroker` flag to true:

```
page.BrokerParameters = new PlatformParameters(
 this,
 true,
 PromptBehavior.SelectAccount);
```

Then, include the parameters in the acquire token call:

```
AuthenticationResult result =
 await

AuthContext.AcquireTokenAsync(
 Resource,
 ClientId,
 new
 Uri(RedirectURI),
 platformParameters)

.ConfigureAwait(false);
```

In MSAL.NET, broker support is enabled on a per-PublicClientApplication basis. It's disabled by default. To enable it, use the

`WithBroker()` parameter (set to true by default) in order to call the broker:

```
var app = PublicClientApplicationBuilder
 .Create(ClientId)
 .WithBroker()
 .WithReplyUri(redirectUriOnIos)
 .Build();
```

In the acquire token call:

```
result = await app.AcquireTokenInteractive(scopes)

.WithParentActivityOrWindow(App.RootViewController
)
 .ExecuteAsync();
```

## Step 2: Set a UIViewController()

In ADAL.NET, you passed in a UIViewController as part of `PlatformParameters`. (See the example in Step 1.) In MSAL.NET, to give developers more flexibility, an object window is used, but it's not required in regular iOS usage. To use the broker, set the object window in order to send and receive responses from the broker.

Current ADAL code:

MSAL counterpart:

A UIViewController is passed into `PlatformParameters` in the iOS-specific platform.

```
page.BrokerParameters = new PlatformParameters(
 this,
 true,
 PromptBehavior.SelectAccount);
```

In MSAL.NET, you do two things to set the object window for iOS:

1. In `AppDelegate.cs`, set `App.RootViewController` to a new `UIViewController()`. This assignment ensures that there's a UIViewController with the call to the broker. If it isn't set correctly, you might get this error:

```
"uiviewcontroller_required_for_ios_broker":"UIViewController is null, so MSAL.NET cannot invoke the iOS broker. See https://aka.ms/msal-net-ios-broker"
```

2. On the `AcquireTokenInteractive` call, use `.WithParentActivityOrWindow(App.RootViewController)`, and pass in the reference to the object window you'll use.

#### For example:

In `App.cs`:

```
public static object RootViewController { get;
set; }
```

In `AppDelegate.cs`:

```
LoadApplication(new App());
App.RootViewController = new
UIViewController();
```

In the acquire token call:

```
result = await app.AcquireTokenInteractive(scopes)
 .WithParentActivityOrWindow(App.RootViewController)
 .ExecuteAsync();
```

### Step 3: Update `AppDelegate` to handle the callback

Both ADAL and MSAL call the broker, and the broker in turn calls back to your application through the `OpenUrl` method of the `AppDelegate` class. For more information, see [this documentation](#).

There are no changes here between ADAL.NET and MSAL.NET.

### Step 4: Register a URL scheme

ADAL.NET and MSAL.NET use URLs to invoke the broker and return the broker response back to the app. Register the URL scheme in the `Info.plist` file for your app as follows:

Current ADAL code:

MSAL counterpart:

The URL scheme is unique to your app.

The

`CFBundleURLSchemes` name must include

`msauth.`

as a prefix, followed by your `CFBundleURLName`

For example:  `$"msauth.(BundleId)"`

```
<key>CFBundleURLTypes</key>
<array>
 <dict>
 <key>CFBundleTypeRole</key>
 <string>Editor</string>
 <key>CFBundleURLName</key>
 <string>com.yourcompany.xforms</string>
 <key>CFBundleURLSchemes</key>
 <array>

 <string>msauth.com.yourcompany.xforms</string>
 </array>
 </dict>
</array>
```

#### NOTE

This URL scheme becomes part of the redirect URI that's used to uniquely identify the app when it receives the response from the broker.

### Step 5: Add the broker identifier to the LSApplicationQueriesSchemes section

ADAL.NET and MSAL.NET both use `-canOpenURL:` to check if the broker is installed on the device. Add the correct identifier for the iOS broker to the LSApplicationQueriesSchemes section of the info.plist file as follows:

Current ADAL code:	MSAL counterpart:
<p>Uses</p> <p><code>msauth</code></p> <pre>&lt;key&gt;LSApplicationQueriesSchemes&lt;/key&gt; &lt;array&gt;   &lt;string&gt;msauth&lt;/string&gt; &lt;/array&gt;</pre>	<p>Uses</p> <p><code>msauthv2</code></p> <pre>&lt;key&gt;LSApplicationQueriesSchemes&lt;/key&gt; &lt;array&gt;   &lt;string&gt;msauthv2&lt;/string&gt; &lt;/array&gt;</pre>

### Step 6: Register your redirect URI in the portal

ADAL.NET and MSAL.NET both add an extra requirement on the redirect URI when it targets the broker. Register the redirect URI with your application in the portal.

Current ADAL code:	MSAL counterpart:
<p>"&lt;app-scheme&gt;://&lt;your.bundle.id&gt;"</p> <p>Example:</p> <p><code>mytestiosapp://com.mycompany.myapp</code></p>	<p><code> \$"msauth.{BundleId}://auth"</code></p> <p>Example:</p> <pre>public static string redirectUriOnIos = "msauth.com.yourcompany.XForms://auth";</pre>

For more information about how to register the redirect URI in the portal, see [Leverage the broker in Xamarin.iOS](#)

applications.

## Next steps

Learn about [Xamarin iOS-specific considerations with MSAL.NET](#).

# Authentication flows

11/4/2019 • 12 minutes to read • [Edit Online](#)

This article describes the different authentication flows provided by Microsoft Authentication Library (MSAL).

These flows can be used in a variety of different application scenarios.

FLOW	DESCRIPTION	USED IN
Interactive	Gets the token through an interactive process that prompts the user for credentials through a browser or pop-up window.	Desktop apps, mobile apps
Implicit grant	Allows the app to get tokens without performing a back-end server credential exchange. This allows the app to sign in the user, maintain session, and get tokens to other web APIs, all within the client JavaScript code.	Single-page applications (SPA)
Authorization code	Used in apps that are installed on a device to gain access to protected resources, such as web APIs. This allows you to add sign-in and API access to your mobile and desktop apps.	Desktop apps, mobile apps, web apps
On-behalf-of	An application invokes a service or web API, which in turn needs to call another service or web API. The idea is to propagate the delegated user identity and permissions through the request chain.	Web APIs
Client credentials	Allows you to access web-hosted resources by using the identity of an application. Commonly used for server-to-server interactions that must run in the background, without immediate interaction with a user.	Daemon apps
Device code	Allows users to sign in to input-constrained devices such as a smart TV, IoT device, or printer.	Desktop/mobile apps
Integrated Windows Authentication	Allows applications on domain or Azure Active Directory (Azure AD) joined computers to acquire a token silently (without any UI interaction from the user).	Desktop/mobile apps
Username/password	Allows an application to sign in the user by directly handling their password. This flow isn't recommended.	Desktop/mobile apps

## How each flow emits tokens and codes

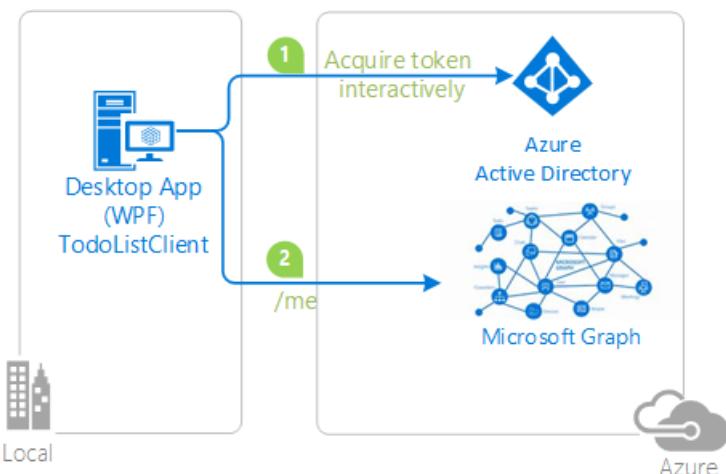
Depending on how your client is built, it can use one (or several) of the authentication flows supported by the Microsoft identity platform. These flows can produce a variety of tokens (id\_tokens, refresh tokens, access tokens) as well as authorization codes, and require different tokens to make them work. This chart provides an overview:

FLOW	REQUIRES	ID_TOKEN	ACCESS TOKEN	REFRESH TOKEN	AUTHORIZATION CODE
Authorization code flow		x	x	x	x
Implicit flow		x	x		
Hybrid OAuth 2.0 flow		x			x
Refresh token redemption	refresh token	x	x	x	
On-behalf-of flow	access token	x	x	x	
Device code flow		x	x	x	
Client credentials			x (app-only)		

Tokens issued via the implicit mode have a length limitation due to being passed back to the browser via the URL (where `response_mode` is `query` or `fragment`). Some browsers have a limit on the size of the URL that can be put in the browser bar and fail when it is too long. Thus, these tokens do not have `groups` or `wids` claims.

## Interactive

MSAL supports the ability to interactively prompt the user for their credentials to sign in, and obtain a token by using those credentials.



For more information on using MSAL.NET to interactively acquire tokens on specific platforms, see:

- [Xamarin Android](#)
- [Xamarin iOS](#)
- [Universal Windows Platform](#)

For more information on interactive calls in MSAL.js, see [Prompt behavior in MSAL.js interactive requests](#).

## Implicit grant

MSAL supports the [OAuth 2 implicit grant flow](#), which allows the app to get tokens from Microsoft identity platform without performing a back-end server credential exchange. This allows the app to sign in the user, maintain session, and get tokens to other web APIs, all within the client JavaScript code.



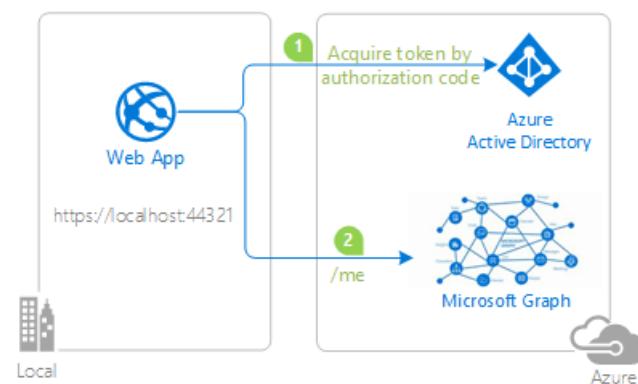
Many modern web applications are built as client-side, single page applications, written by using JavaScript or an SPA framework such as Angular, Vuejs, and Reactjs. These applications run in a web browser, and have different authentication characteristics than traditional server-side web applications. The Microsoft identity platform enables single page applications to sign in users, and get tokens to access back-end services or web APIs, by using the implicit grant flow. The implicit flow allows the application to get ID tokens to represent the authenticated user, and also access tokens needed to call protected APIs.

This authentication flow doesn't include application scenarios that use cross-platform JavaScript frameworks such as Electron and React-Native, because they require further capabilities for interaction with the native platforms.

## Authorization code

MSAL supports the [OAuth 2 authorization code grant](#). This grant can be used in apps that are installed on a device to gain access to protected resources, such as web APIs. This allows you to add sign-in and API access to your mobile and desktop apps.

When users sign in to web applications (websites), the web application receives an authorization code. The authorization code is redeemed to acquire a token to call web APIs. In ASP.NET and ASP.NET Core web apps, the only goal of `AcquireTokenByAuthorizationCode` is to add a token to the token cache. The token can then be used by the application (usually in the controllers, which just get a token for an API by using `AcquireTokenSilent`).



In the preceding diagram, the application:

1. Requests an authorization code, which is redeemed for an access token.
2. Uses the access token to call a web API.

### Considerations

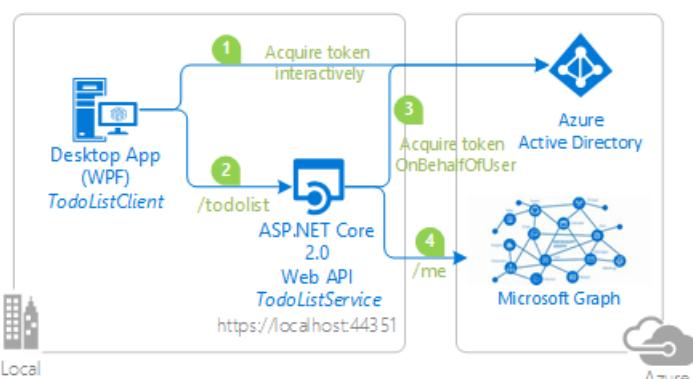
- You can use the authorization code only once to redeem a token. Don't try to acquire a token multiple times with the same authorization code (it's explicitly prohibited by the protocol standard specification). If you redeem the code several times intentionally, or because you are not aware that a framework also does it for you, you'll get the following error:

`AADSTS7002: Error validating credentials. AADSTS54005: OAuth2 Authorization code was already redeemed, please retry with a new valid code or use an existing refresh token.`

- If you're writing an ASP.NET or ASP.NET Core application, this might happen if you don't tell the framework that you've already redeemed the authorization code. For this, you need to call the `context.HandleCodeRedemption()` method of the `AuthorizationCodeReceived` event handler.
- Avoid sharing the access token with ASP.NET, which might prevent incremental consent happening correctly. For more information, see [issue #693](#).

## On-behalf-of

MSAL supports the [OAuth 2 on-behalf-of authentication flow](#). This flow is used when an application invokes a service or web API, which in turn needs to call another service or web API. The idea is to propagate the delegated user identity and permissions through the request chain. For the middle-tier service to make authenticated requests to the downstream service, it needs to secure an access token from the Microsoft identity platform, on behalf of the user.



In the preceding diagram:

1. The application acquires an access token for the web API.
2. A client (web, desktop, mobile, or single-page application) calls a protected web API, adding the access token as a bearer token in the authentication header of the HTTP request. The web API authenticates the user.
3. When the client calls the web API, the web API requests another token on-behalf-of the user.
4. The protected web API uses this token to call a downstream web API on-behalf-of the user. The web API can also later request tokens for other downstream APIs (but still on behalf of the same user).

## Client credentials

MSAL supports the [OAuth 2 client credentials flow](#). This flow allows you to access web-hosted resources by using the identity of an application. This type of grant is commonly used for server-to-server interactions that must run in the background, without immediate interaction with a user. These types of applications are often referred to as daemons or service accounts.

The client credentials grant flow permits a web service (a 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 a website. For a higher level of assurance, the Microsoft identity platform also allows the calling service to use a certificate (instead of a shared secret) as a credential.

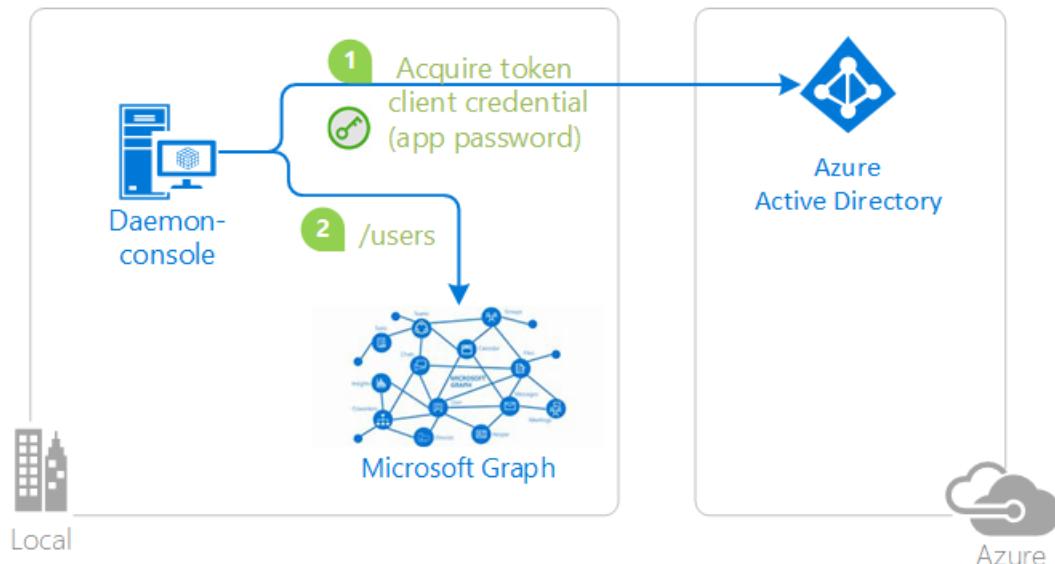
### NOTE

The confidential client flow isn't available on the mobile platforms (UWP, Xamarin.iOS, and Xamarin.Android), because these only support public client applications. Public client applications don't know how to prove the application's identity to the Identity Provider. A secure connection can be achieved on web app or web API back ends by deploying a certificate.

MSAL.NET supports two types of client credentials. These client credentials need to be registered with Azure AD.

The credentials are passed in to the constructors of the confidential client application in your code.

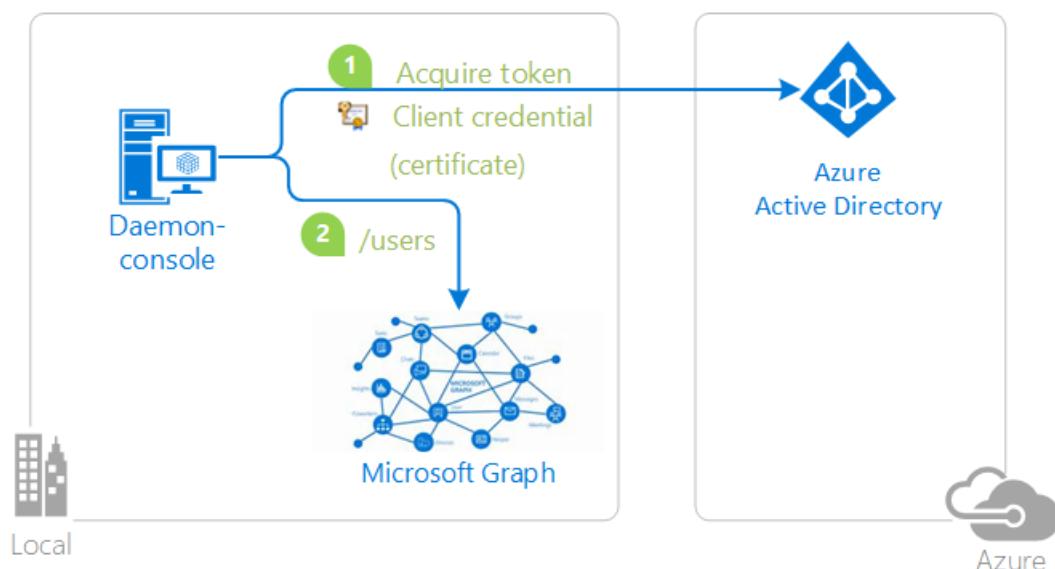
### Application secrets



In the preceding diagram, the application:

1. Acquires a token by using application secret or password credentials.
2. Uses the token to make requests of the resource.

### Certificates



In the preceding diagram, the application:

1. Acquires a token by using certificate credentials.
2. Uses the token to make requests of the resource.

These client credentials need to be:

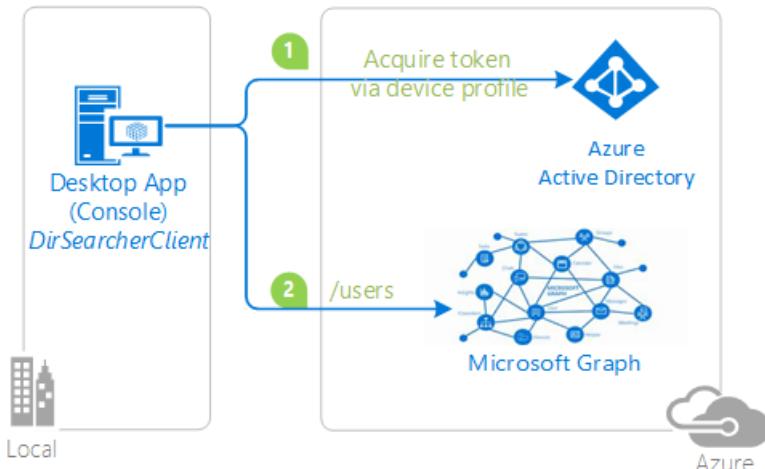
- Registered with Azure AD.
- Passed in at the construction of the confidential client application in your code.

## Device code

MSAL supports the [OAuth 2 device code flow](#), which allows users to sign in to input-constrained devices, such as a smart TV, IoT device, or printer. Interactive authentication with Azure AD requires a web browser. The device code

flow lets the user use another device (for example, another computer or a mobile phone) to sign in interactively, where the device or operating system doesn't provide a web browser.

By using the device code flow, the application obtains tokens through a two-step process especially designed for these devices or operating systems. Examples of such applications include those running on IoT devices or command-line tools (CLI).



In the preceding diagram:

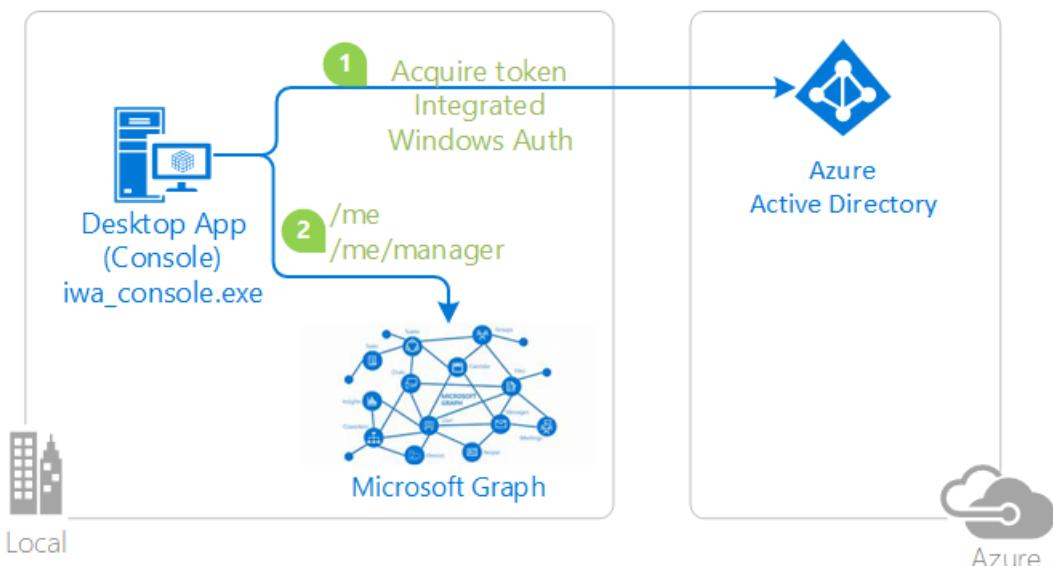
1. Whenever user authentication is required, the app provides a code, and asks the user to use another device (such as an internet-connected smartphone) to go to a URL (for example, <https://microsoft.com/devicelogin>). The user is then prompted to enter the code, and proceeds through a normal authentication experience, including consent prompts and multi-factor authentication if necessary.
2. Upon successful authentication, the command-line app receives the required tokens through a back channel, and uses them to perform the web API calls it needs.

### Constraints

- Device code flow is only available on public client applications.
- The authority passed in when constructing the public client application must be one of the following:
  - Tenanted (of the form `https://login.microsoftonline.com/{tenant}/` where `{tenant}` is either the GUID representing the tenant ID or a domain associated with the tenant).
  - For any work and school accounts (`https://login.microsoftonline.com/organizations/`).
- Microsoft personal accounts aren't yet supported by the Azure AD v2.0 endpoint (you can't use the `/common` or `/consumers` tenants).

## Integrated Windows Authentication

MSAL supports Integrated Windows Authentication (IWA) for desktop or mobile applications that run on a domain joined or Azure AD joined Windows computer. Using IWA, these applications can acquire a token silently (without any UI interaction from the user).



In the preceding diagram, the application:

1. Acquires a token by using Integrated Windows Authentication.
2. Uses the token to make requests of the resource.

### Constraints

IWA supports federated users only, meaning users created in Active Directory and backed by Azure AD. Users created directly in Azure AD, without Active Directory backing (managed users) can't use this authentication flow. This limitation doesn't affect the [username/password flow](#).

IWA is for apps written for .NET Framework, .NET Core, and Universal Windows Platform platforms.

IWA doesn't bypass multi-factor authentication. If multi-factor authentication is configured, IWA might fail if a multi-factor authentication challenge is required. Multi-factor authentication requires user interaction.

You don't control when the identity provider requests two-factor authentication to be performed. The tenant admin does. Typically, two-factor authentication is required when you sign in from a different country, when you're not connected via VPN to a corporate network, and sometimes even when you are connected via VPN. Azure AD uses AI to continuously learn if two-factor authentication is required. If IWA fails, you should fall back to an [interactive user prompt] (#interactive).

The authority passed in when constructing the public client application must be one of the following:

- Tenanted (of the form `https://login.microsoftonline.com/{tenant}/` where `{tenant}` is either the guid representing the tenant ID or a domain associated with the tenant).
- For any work and school accounts (`https://login.microsoftonline.com/organizations/`). Microsoft personal accounts are not supported (you can't use `/common` or `/consumers` tenants).

Because IWA is a silent flow, one of the following must be true:

- The user of your application must have previously consented to use the application.
- The tenant admin must have previously consented to all users in the tenant to use the application.

This means that one of the following is true:

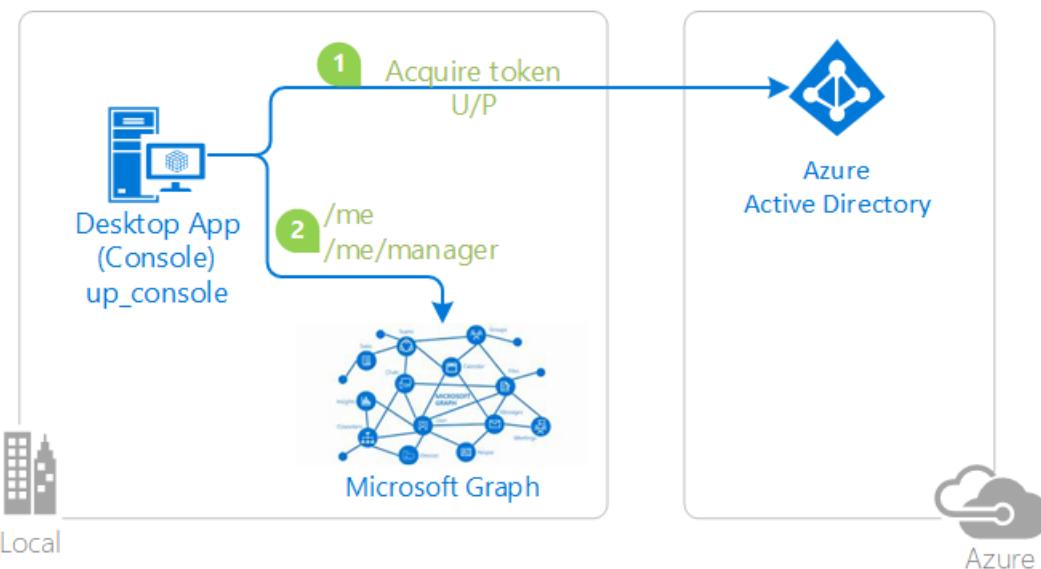
- You as a developer have selected **Grant** on the Azure portal for yourself.
- A tenant admin has selected **Grant/revoke admin consent for {tenant domain}** in the **API permissions** tab of the registration for the application (see [Add permissions to access web APIs](#)).
- You have provided a way for users to consent to the application (see [Requesting individual user consent](#)).
- You have provided a way for the tenant admin to consent for the application (see [admin consent](#)).

The IWA flow is enabled for .NET desktop, .NET Core, and Windows Universal Platform apps. On .NET Core you must provide the username to IWA, because .NET Core can't obtain usernames from the operating system.

For more information on consent, see [v2.0 permissions and consent](#).

## Username/password

MSAL supports the [OAuth 2 resource owner password credentials grant](#), which allows an application to sign in the user by directly handling their password. In your desktop application, you can use the username/password flow to acquire a token silently. No UI is required when using the application.



In the preceding diagram, the application:

1. Acquires a token by sending the username and password to the identity provider.
2. Calls a web API by using the token.

### WARNING

This flow isn't recommended. It requires a high degree of trust and user exposure. You should only use this flow when other, more secure, flows can't be used. For more information, see [What's the solution to the growing problem of passwords?](#)

The preferred flow for acquiring a token silently on Windows domain-joined machines is [Integrated Windows Authentication](#). Otherwise, you can also use [Device code flow](#).

Although this is useful in some cases (DevOps scenarios), if you want to use username/password in interactive scenarios where you provide your own UI, try to avoid it. By using username/password:

- Users who need to do multi-factor authentication won't be able to sign in (as there is no interaction).
- Users won't be able to do single sign-on.

### Constraints

Apart from the [Integrated Windows Authentication constraints](#), the following constraints also apply:

- The username/password flow isn't compatible with Conditional Access and multi-factor authentication. As a consequence, if your app runs in an Azure AD tenant where the tenant admin requires multi-factor authentication, you can't use this flow. Many organizations do that.
- It works only for work and school accounts (not Microsoft accounts).
- The flow is available on .NET desktop and .NET Core, but not on Universal Windows Platform.

## Azure AD B2C specifics

For more information on using MSAL.NET and Azure AD B2C, see [Using ROPC with Azure AD B2C \(MSAL.NET\)](#).

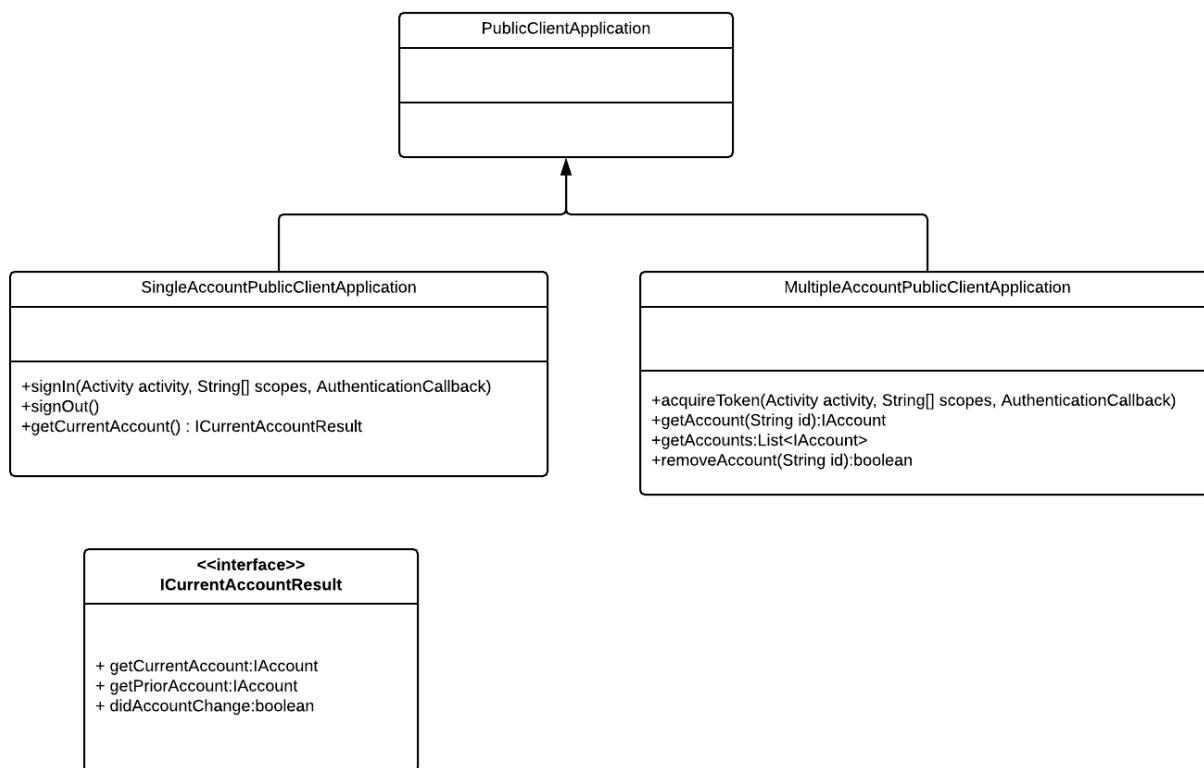
# Single and multiple account public client apps

9/30/2019 • 4 minutes to read • [Edit Online](#)

This article will help you understand the types used in single and multiple account public client apps, with a focus on single account public client apps.

The Azure Active Directory Authentication Library (ADAL) models the server. The Microsoft Authentication Library (MSAL) instead models your client application. The majority of Android apps are considered public clients. A public client is an app that can't securely keep a secret.

MSAL specializes the API surface of `PublicClientApplication` to simplify and clarify the development experience for apps that allow only one account to be used at a time. `PublicClientApplication` is subclassed by `SingleAccountPublicClientApplication` and `MultipleAccountPublicClientApplication`. The following diagram shows the relationship between these classes.



## Single account public client application

The `SingleAccountPublicClientApplication` class allows you to create an MSAL-based app that only allows a single account to be signed in at a time. `SingleAccountPublicClientApplication` differs from `PublicClientApplication` in the following ways:

- MSAL tracks the currently signed-in account.
  - If your app is using a broker (the default during Azure portal app registration) and is installed on a device where a broker is present, MSAL will verify that the account is still available on the device.
- `signIn` allows you to sign in an account explicitly and separately from requesting scopes.
- `acquireTokenSilent` doesn't require an account parameter. If you do provide an account, and the account you provide doesn't match the current account tracked by MSAL, an `MsalClientException` is thrown.

- `acquireToken` doesn't allow the user to switch accounts. If the user attempts to switch to a different account, an exception is thrown.
- `getCurrentAccount` returns a result object that provides the following:
  - A boolean indicating whether the account changed. An account may be changed as a result of being removed from the device, for example.
  - The prior account. This is useful if you need to do any local data cleanup when the account is removed from the device or when a new account is signed in.
  - The currentAccount.
- `signout` removes any tokens associated with your client from the device.

When an Android Authentication broker such as Microsoft Authenticator or Intune Company Portal is installed on the device and your app is configured to use the broker, `signOut` won't remove the account from the device.

## Single account scenario

The following pseudo code illustrates using `SingleAccountPublicClientApplication`.

```

// Construct Single Account Public Client Application
ISingleAccountPublicClientApplication app =
PublicClientApplication.createSingleAccountPublicClientApplication(getApplicationContext(), R.raw.msal_config);

String[] scopes = {"User.Read"};
IAccount mAccount = null;

// Acquire a token interactively
// The user will get a UI prompt before getting the token.
app.signIn(getActivity(), scopes, new AuthenticationCallback()
{

 @Override
 public void onSuccess(IAuthenticationResult authenticationResult)
 {
 mAccount = authenticationResult.getAccount();
 }

 @Override
 public void onError(MsalException exception)
 {

 }

 @Override
 public void onCancel()
 {
 }
}
);

// Load Account Specific Data
getDataForAccount(account);

// Get Current Account
ICurrentAccountResult currentAccountResult = app.getCurrentAccount();
if (currentAccountResult.didAccountChange())
{
 // Account Changed Clear existing account data
 clearDataForAccount(currentAccountResult.getPriorAccount());
 mAccount = currentAccountResult.getCurrentAccount();
 if (account != null)
 {
 //load data for new account
 getDataForAccount(account);
 }
}

// Sign out
if (app.signOut())
{
 clearDataForAccount(mAccount);
 mAccount = null;
}

```

## Multiple account public client application

The `MultipleAccountPublicClientApplication` class is used to create MSAL-based apps that allow multiple accounts to be signed in at the same time. It allows you to get, add, and remove accounts as follows:

### Add an account

Use one or more accounts in your application by calling `acquireToken` one or more times.

### Get accounts

- Call `getAccount` to get a specific account.

- Call `getAccounts` to get a list of accounts currently known to the app.

Your app won't be able to enumerate all Microsoft identity platform accounts on the device known to the broker app. It can only enumerate accounts that have been used by your app. Accounts that have been removed from the device won't be returned by these functions.

### Remove an account

Remove an account by calling `removeAccount` with an account identifier.

If your app is configured to use a broker, and a broker is installed on the device, the account won't be removed from the broker when you call `removeAccount`. Only tokens associated with your client are removed.

## Multiple account scenario

The following pseudo code shows how to create a multiple account app, list accounts on the device, and acquire tokens.

```

// Construct Multiple Account Public Client Application
IMultipleAccountPublicClientApplication app =
PublicClientApplication.createMultipleAccountPublicClientApplication(getApplicationContext(),
R.raw.msal_config);

String[] scopes = {"User.Read"};
IAccount mAccount = null;

// Acquire a token interactively
// The user will be required to interact with a UI to obtain a token
app.acquireToken(getActivity(), scopes, new AuthenticationCallback()
{

 @Override
 public void onSuccess(IAuthenticationResult authenticationResult)
 {
 mAccount = authenticationResult.getAccount();
 }

 @Override
 public void onError(MsalException exception)
 {
 }

 @Override
 public void onCancel()
 {
 }
});

...
// Get the default authority
String authority = app.getConfiguration().getDefaultAuthority().getAuthorityURL().toString();

// Get a list of accounts on the device
List<IAccount> accounts = app.getAccounts();

// Pick an account to obtain a token from without prompting the user to sign in
IAccount selectedAccount = accounts.get(0);

// Get a token without prompting the user
app.acquireTokenSilentAsync(scopes, selectedAccount, authority, new SilentAuthenticationCallback()
{
 @Override
 public void onSuccess(IAuthenticationResult authenticationResult)
 {
 mAccount = authenticationResult.getAccount();
 }

 @Override
 public void onError(MsalException exception)
 {
 }
});

```

# Acquire and cache tokens using the Microsoft authentication library (MSAL)

11/10/2019 • 6 minutes to read • [Edit Online](#)

Access tokens enable clients to securely call web APIs protected by Azure. There are many ways to acquire a token using Microsoft Authentication Library (MSAL). Some ways require user interactions through a web browser. Some don't require any user interactions. In general, the way to acquire a token depends on if the application is a public client application (desktop or mobile app) or a confidential client application (Web App, Web API, or daemon application like a Windows service).

MSAL caches a token after it has been acquired. Application code should try to get a token silently (from the cache), first, before acquiring a token by other means.

You can also clear the token cache, which is achieved by removing the accounts from the cache. This does not remove the session cookie which is in the browser, though.

## Scopes when acquiring tokens

Scopes are the permissions that a web API exposes for client applications to request access to. Client applications request the user's consent for these scopes when making authentication requests to get tokens to access the web APIs. MSAL allows you to get tokens to access Azure AD for developers (v1.0) and Microsoft identity platform (v2.0) APIs. v2.0 protocol uses scopes instead of resource in the requests. For more information, read [v1.0 and v2.0 comparison](#). Based on the web API's configuration of the token version it accepts, the v2.0 endpoint returns the access token to MSAL.

A number of MSAL acquire token methods require a *scopes* parameter. This parameter is a simple list of strings that declare the desired permissions and resources that are requested. Well known scopes are the [Microsoft Graph permissions](#).

It's also possible in MSAL to access v1.0 resources. For more information, read [Scopes for a v1.0 application](#).

### Request specific scopes for a web API

When your application needs to request tokens with specific permissions for a resource API, you will need to pass the scopes containing the app ID URI of the API in the below format: <app ID URI>/<scope>

For example, scopes for Microsoft Graph API: `https://graph.microsoft.com/User.Read`

Or, for example, scopes for a custom web API: `api://abcdefg-1234-abcd-efgh-1234567890/api.read`

For the Microsoft Graph API, only, a scope value `user.read` maps to `https://graph.microsoft.com/User.Read` format and can be used interchangeably.

#### NOTE

Certain web APIs such as Azure Resource Manager API (<https://management.core.windows.net/>) expect a trailing '/' in the audience claim (aud) of the access token. In this case, it is important to pass the scope as [https://management.core.windows.net//user\\_impersonation](https://management.core.windows.net//user_impersonation) (note the double slash), for the token to be valid in the API.

### Request dynamic scopes for incremental consent

When building applications using v1.0, you had to register the full set of permissions (static scopes) required by the application for the user to consent at the time of login. In v2.0, you can request additional permissions as

needed using the scope parameter. These are called dynamic scopes and allow the user to provide incremental consent to scopes.

For example, you can initially sign in the user and deny them any kind of access. Later, you can give them the ability to read the calendar of the user by requesting the calendar scope in the acquire token methods and get the user's consent.

For example: <https://graph.microsoft.com/User.Read> and <https://graph.microsoft.com/Calendar.Read>

## Acquiring tokens silently (from the cache)

MSAL maintains a token cache (or two caches for confidential client applications) and caches a token after it has been acquired. In many cases, attempting to silently get a token will acquire another token with more scopes based on a token in the cache. It's also capable of refreshing a token when it's getting close to expiration (as the token cache also contains a refresh token).

### Recommended call pattern for public client applications

Application code should try to get a token silently (from the cache), first. If the method call returns a "UI required" error or exception, try acquiring a token by other means.

However, there are two flows before which you **should not** attempt to silently acquire a token:

- [client credentials flow](#), which does not use the user token cache, but an application token cache. This method takes care of verifying this application token cache before sending a request to the STS.
- [authorization code flow](#) in Web Apps, as it redeems a code that the application got by signing-in the user, and having them consent for more scopes. Since a code is passed as a parameter, and not an account, the method cannot look in the cache before redeeming the code, which requires, anyway, a call to the service.

### Recommended call pattern in Web Apps using the Authorization Code flow

For Web applications that use the [OpenID Connect authorization code flow](#), the recommended pattern in the controllers is to:

- Instantiate a confidential client application with a token cache with customized serialization.
- Acquire the token using the authorization code flow

## Acquiring tokens

Generally, the method of acquiring a token depends on whether it's a public client or confidential client application.

### Public client applications

For public client applications (desktop or mobile app), you:

- Often acquire tokens interactively, having the user sign in through a UI or pop-up window.
- Can [get a token silently for the signed-in user](#) using Integrated Windows Authentication (IWA/Kerberos) if the desktop application is running on a Windows computer joined to a domain or to Azure.
- Can [get a token with a username and password](#) in .NET framework desktop client applications, but this is not recommended. Do not use username/password in confidential client applications.
- Can acquire a token through the [device code flow](#) in applications running on devices which don't have a web browser. The user is provided with a URL and a code, who then goes to a web browser on another device and enters the code and signs in. Azure AD then sends a token back to the browser-less device.

### Confidential client applications

For confidential client applications (Web App, Web API, or daemon application like a Windows service), you:

- Acquire tokens **for the application itself** and not for a user, using the [client credentials flow](#). This can be used

for syncing tools, or tools that process users in general and not a specific user.

- Use the [On-behalf-of flow](#) for a web API to call an API on behalf of the user. The application is identified with client credentials in order to acquire a token based on a user assertion (SAML for example, or a JWT token). This flow is used by applications that need to access resources of a particular user in service-to-service calls.
- Acquire tokens using the [authorization code flow](#) in web apps after the user signs in through the authorization request URL. OpenID Connect application typically use this mechanism, which lets the user sign in using Open ID connect and then access web APIs on behalf of the user.

## Authentication results

When your client requests an access token, Azure AD also returns an authentication result which includes some metadata about the access token. This information includes the expiry time of the access token and the scopes for which it's valid. This data allows your app to do intelligent caching of access tokens without having to parse the access token itself. The authentication result exposes:

- The [access token](#) for the web API to access resources. This is a string, usually a base64 encoded JWT but the client should never look inside the access token. The format isn't guaranteed to remain stable and it can be encrypted for the resource. People writing code depending on access token content on the client is one of the biggest sources of errors and client logic breaks.
- The [ID token](#) for the user (this is a JWT).
- The token expiration, which tells the date/time when the token expires.
- The tenant ID contains the tenant in which the user was found. For guest users (Azure AD B2B scenarios), the tenant ID is the guest tenant, not the unique tenant. When the token is delivered in the name of a user, the authentication result also contains information about this user. For confidential client flows where tokens are requested with no user (for the application), this user information is null.
- The scopes for which the token was issued.
- The unique ID for the user.

## Next steps

If you are using MSAL for Java, learn about [Custom token cache serialization in MSAL for Java](#).

Learn about [handling errors and exceptions](#).

# Scopes for a Web API accepting v1.0 tokens

8/15/2019 • 2 minutes to read • [Edit Online](#)

OAuth2 permissions are permission scopes that a Azure AD for developers (v1.0) web API (resource) application exposes to client applications. These permission scopes may be granted to client applications during consent. See the section about `oauth2Permissions` in the [Azure Active Directory application manifest reference](#).

## Scopes to request access to specific OAuth2 permissions of a v1.0 application

If you want to acquire tokens for specific scopes of a v1.0 application (for example the Azure AD graph, which is <https://graph.windows.net>), you need to create scopes by concatenating a desired resource identifier with a desired OAuth2 permission for that resource.

For example, to access on behalf of the user a v1.0 web API where the app ID URI is `ResourceId`:

```
var scopes = new [] { ResourceId + "/user_impersonation"};
```

```
var scopes = [ResourceId + "/user_impersonation"];
```

If you want to read and write with MSAL.NET Azure Active Directory using the Azure AD graph API (<https://graph.windows.net>), you would create a list of scopes as in the following:

```
string ResourceId = "https://graph.windows.net/";
var scopes = new [] { ResourceId + "Directory.Read", ResourceID + "Directory.Write"}
```

```
var ResourceId = "https://graph.windows.net/";
var scopes = [ResourceId + "Directory.Read", ResourceID + "Directory.Write"];
```

If you want to write the scope corresponding to the Azure Resource Manager API (<https://management.core.windows.net>), you need to request the following scope (note the two slashes):

```
var scopes = new[] {"https://management.core.windows.net//user_impersonation"};
var result = await app.AcquireTokenInteractive(scopes).ExecuteAsync();

// then call the API: https://management.azure.com/subscriptions?api-version=2016-09-01
```

### NOTE

You need to use two slashes because the Azure Resource Manager API expects a slash in its audience claim (aud), and then there is a slash to separate the API name from the scope.

The logic used by Azure AD is the following:

- For ADAL (v1.0) endpoint with a v1.0 access token (the only possible), aud=resource
- For MSAL (Microsoft identity platform (v2.0) endpoint) asking an access token for a resource accepting v2.0 tokens, aud=resource.AppId

- For MSAL (v2.0 endpoint) asking an access token for a resource accepting a v1.0 access token (which is the case above), Azure AD parses the desired audience from the requested scope by taking everything before the last slash and using it as the resource identifier. Therefore if https://database.windows.net expects an audience of "https://database.windows.net/", you'll need to request a scope of "https://database.windows.net//.default". See also GitHub issue [#747: Resource url's trailing slash is omitted, which caused sql auth failure.](#)

## Scopes to request access to all the permissions of a v1.0 application

If you want to acquire a token for all the static scopes of a v1.0 application, append ".default" to the app ID URI of the API:

```
ResourceId = "someAppIDURI";
var scopes = new [] { ResourceId+=".default"};
```

```
var ResourceId = "someAppIDURI";
var scopes = [ResourceId + "/.default"];
```

## Scopes to request for client credential flow / daemon app

In the case of client credential flow, the scope to pass would also be `[/.default]`. This tells to Azure AD: "all the app-level permissions that the admin has consented to in the application registration."

# Public client and confidential client applications

10/23/2019 • 2 minutes to read • [Edit Online](#)

Microsoft Authentication Library (MSAL) defines two types of clients: public clients and confidential clients. The two client types are distinguished by their ability to authenticate securely with the authorization server and maintain the confidentiality of their client credentials. In contrast, Azure AD Authentication Library (ADAL) uses what's called *authentication context* (which is a connection to Azure AD).

- **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). These values are difficult for end users to extract. A web app is the most common confidential client. The client ID is exposed through the web browser, but the secret is passed only in the back channel and never directly exposed.

Confidential client apps:



- **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.

Public client apps:



## NOTE

In MSAL.js, there is no separation of public and confidential client apps. MSAL.js represents client apps as user agent-based apps, public clients in which the client code is executed in a user agent like a web browser. These clients don't store secrets because the browser context is openly accessible.

## Comparing the client types

Here are some similarities and differences between public client and confidential client apps:

- Both kinds of app maintain a user token cache and can acquire a token silently (when the token is already in the token cache). Confidential client apps also have an app token cache for tokens that are for the app itself.
- Both types of app manage user accounts and can get an account from the user token cache, get an account from its identifier, or remove an account.
- Public client apps have four ways to acquire a token (four authentication flows). Confidential client apps have three ways to acquire a token (and one way to compute the URL of the identity provider authorize endpoint). For more information, see [Acquiring tokens](#).

If you've used ADAL, you might notice that, unlike ADAL's authentication context, in MSAL the client ID (also called the *application ID* or *app ID*) is passed once at the construction of the application. It doesn't need to be

passed again when the app acquires a token. This is true for both for public and confidential client apps. Constructors of confidential client apps are also passed client credentials: the secret they share with the identity provider.

## Next steps

Learn about:

- [Client application configuration options](#)
- [Instantiating client applications by using MSAL.NET](#)
- [Instantiating client applications by using MSAL.js](#)

# Application configuration options

10/23/2019 • 6 minutes to read • [Edit Online](#)

In your code, you initialize a new public or confidential client application (or user-agent for MSAL.js) to authenticate and acquire tokens. You can set a number of configuration options when you initialize the client app in Microsoft Authentication Library (MSAL). These options fall into two groups:

- Registration options, including:
  - **Authority** (composed of the identity provider [instance](#) and sign-in [audience](#) for the app, and possibly the tenant ID).
  - **Client ID**.
  - **Redirect URI**.
  - **Client secret** (for confidential client applications).
- [Logging options](#), including log level, control of personal data, and the name of the component using the library.

## Authority

The authority is a URL that indicates a directory that MSAL can request tokens from. Common authorities are:

- <https://login.microsoftonline.com/<tenant>/>, where <tenant> is the tenant ID of the Azure Active Directory (Azure AD) tenant or a domain associated with this Azure AD tenant. Used only to sign in users of a specific organization.
- <https://login.microsoftonline.com/common/>. Used to sign in users with work and school accounts or personal Microsoft accounts.
- <https://login.microsoftonline.com/organizations/>. Used to sign in users with work and school accounts.
- <https://login.microsoftonline.com/consumers/>. Used to sign in users with only personal Microsoft accounts (formerly known as Windows Live ID accounts).

The authority setting needs to be consistent with what's declared in the application registration portal.

The authority URL is composed of the instance and the audience.

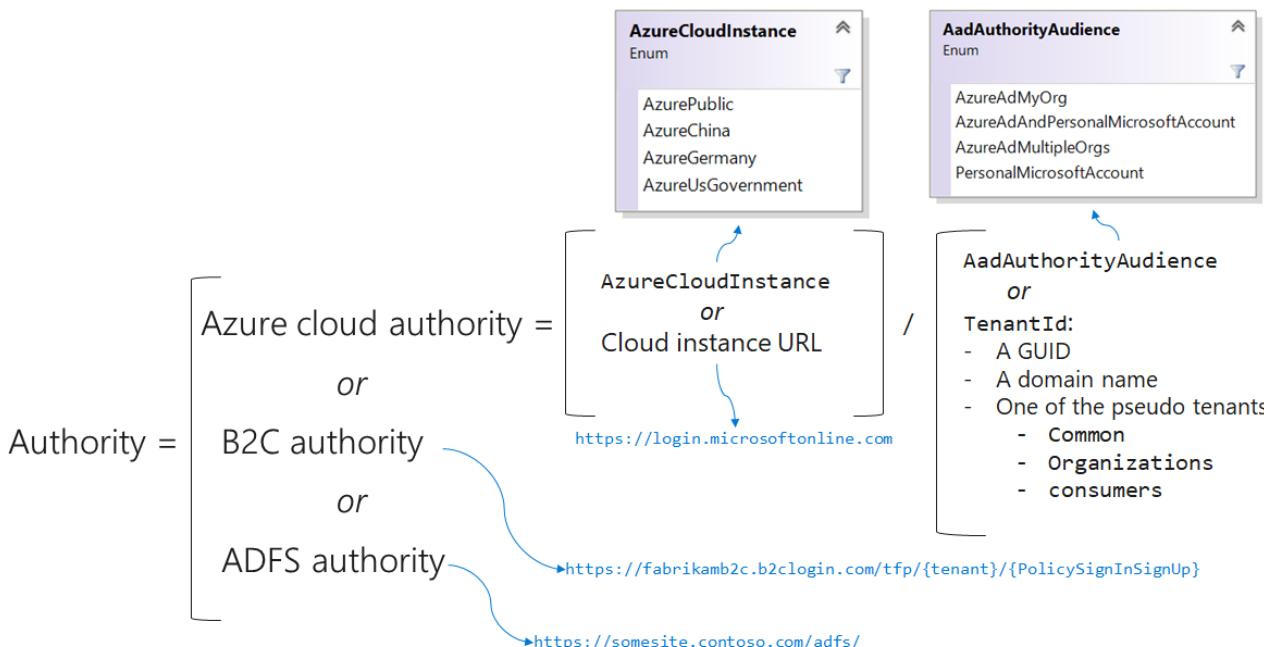
The authority can be:

- An Azure AD cloud authority.
- An Azure AD B2C authority. See [B2C specifics](#).
- An Active Directory Federation Services (AD FS) authority. See [AD FS support](#).

Azure AD cloud authorities have two parts:

- The identity provider *instance*
- The sign-in *audience* for the app

The instance and audience can be concatenated and provided as the authority URL. In versions of MSAL.NET earlier than MSAL 3.x, you had to compose the authority yourself, based on the cloud you wanted to target and the sign-in audience. This diagram shows how the authority URL is composed:



## Cloud instance

The *instance* is used to specify if your app is signing users from the Azure public cloud or from national clouds. Using MSAL in your code, you can set the Azure cloud instance by using an enumeration or by passing the URL to the [national cloud instance](#) as the `Instance` member (if you know it).

MSAL.NET will throw an explicit exception if both `Instance` and `AzureCloudInstance` are specified.

If you don't specify an instance, your app will target the Azure public cloud instance (the instance of URL <https://login.onmicrosoft.com>).

## Application audience

The sign-in audience depends on the business needs for your app:

- If you're a line of business (LOB) developer, you'll probably produce a single-tenant application that will be used only in your organization. In that case, you need to specify the organization, either by its tenant ID (the ID of your Azure AD instance) or by a domain name associated with the Azure AD instance.
- If you're an ISV, you might want to sign in users with their work and school accounts in any organization or in some organizations (multitenant app). But you might also want to have users sign in with their personal Microsoft accounts.

### How to specify the audience in your code/configuration

Using MSAL in your code, you specify the audience by using one of the following values:

- The Azure AD authority audience enumeration
- The tenant ID, which can be:
  - A GUID (the ID of your Azure AD instance), for single-tenant applications
  - A domain name associated with your Azure AD instance (also for single-tenant applications)
- One of these placeholders as a tenant ID in place of the Azure AD authority audience enumeration:
  - `organizations` for a multitenant application
  - `consumers` to sign in users only with their personal accounts
  - `common` to sign in users with their work and school accounts or their personal Microsoft accounts

MSAL will throw a meaningful exception if you specify both the Azure AD authority audience and the tenant ID.

If you don't specify an audience, your app will target Azure AD and personal Microsoft accounts as an audience. (That is, it will behave as though `common` were specified.)

## Effective audience

The effective audience for your application will be the minimum (if there's an intersection) of the audience you set in your app and the audience that's specified in the app registration. In fact, the [App registrations](#) experience lets you specify the audience (the supported account types) for the app. For more information, see [Quickstart: Register an application with the Microsoft identity platform](#).

Currently, the only way to get an app to sign in users with only personal Microsoft accounts is to configure both of these settings:

- Set the app registration audience to `Work and school accounts and personal accounts`.
- Set the audience in your code/configuration to `AadAuthorityAudience.PersonalMicrosoftAccount` (or `TenantID` = "consumers").

## Client ID

The client ID is the unique application (client) ID assigned to your app by Azure AD when the app was registered.

## Redirect URI

The redirect URI is the URI the identity provider will send the security tokens back to.

### Redirect URI for public client apps

If you're a public client app developer who's using MSAL:

- You'd want to use `.WithDefaultRedirectUri()` in desktop or UWP applications (MSAL.NET 4.1+). This method will set the public client application's redirect uri property to the default recommended redirect uri for public client applications.

PLATFORM	REDIRECT URI
Desktop app (.NET FW)	<code>https://login.microsoftonline.com/common/oauth2/nativeclient</code>
UWP	<p>value of <code>WebAuthenticationBroker.GetCurrentApplicationCallbackUri()</code> . This enables SSO with the browser by setting the value to the result of <code>WebAuthenticationBroker.GetCurrentApplicationCallbackU</code> ri() which you need to register</p>
.NET Core	<p><code>https://localhost</code> . This enables the user to use the system browser for interactive authentication since .NET Core doesn't have a UI for the embedded web view at the moment.</p>

- You don't need to add a redirect URI if you're building a Xamarin Android and iOS application that doesn't support broker (the redirect URI is automatically set to `msal{ClientId}://auth` for Xamarin Android and iOS)
- You need to configure the redirect URI in [App registrations](#):

The screenshot shows the Azure portal's 'Authentication' configuration page for an application. On the left, a sidebar lists 'Overview', 'Quickstart', 'Manage' (with 'Branding', 'Authentication' selected, 'Certificates & secrets', 'API permissions', 'Expose an API', 'Owners', and 'Manifest'), 'Support + Troubleshooting' (with 'Troubleshooting' and 'New support request'), and a 'Save' button.

The main content area is titled 'Redirect URIs'. It explains that these URLs will accept authentication responses after users log in. A note says they are also referred to as reply URLs. A link to 'Learn more about adding support for web, mobile and desktop clients' is provided.

TYPE	REDIRECT URI
Web	e.g. https://myapp.com/auth

A section titled 'Suggested Redirect URIs for public clients (mobile, desktop)' lists four options:

- msal5d6f2191-d895-4d0f-81cd-986806b1db59://auth (MSAL only) [Copy](#)
- urn:nietfwgoauth:2.0:oob [Copy](#)
- https://login.microsoftonline.com/common/oauth2/nativeclient [Copy](#)
- https://login.live.com/oauth20\_desktop.srf (LiveSDK) [Copy](#)

You can override the redirect URI by using the `RedirectUri` property (for example, if you use brokers). Here are some examples of redirect URIs for that scenario:

- `RedirectUriOnAndroid` = "msauth-5a434691-ccb2-4fd1-b97b-b64bcfbc03fc://com.microsoft.identity.client.sample";
- `RedirectUriOnIos` = \$"msauth.{Bundle.ID}://auth";

For additional iOS details, see [Migrate iOS applications that use Microsoft Authenticator from ADAL.NET to MSAL.NET](#) and [Leveraging the broker on iOS](#). For additional Android details, see [Brokered auth in Android](#).

### Redirect URI for confidential client apps

For web apps, the redirect URI (or reply URI) is the URI that Azure AD will use to send the token back to the application. This URI can be the URL of the web app/Web API if the confidential app is one of these. The redirect URI needs to be registered in app registration. This registration is especially important when you deploy an app that you've initially tested locally. You then need to add the reply URL of the deployed app in the application registration portal.

For daemon apps, you don't need to specify a redirect URI.

## Client secret

This option specifies the client secret for the confidential client app. This secret (app password) is provided by the application registration portal or provided to Azure AD during app registration with PowerShell AzureAD, PowerShell AzureRM, or Azure CLI.

## Logging

The other configuration options enable logging and troubleshooting. See the [Logging](#) article for details on how to use them.

## Next steps

Learn about [instantiating client applications by using MSAL.NET](#). Learn about [instantiating client applications by using MSAL.js](#).

# Initialize client applications using MSAL.NET

11/11/2019 • 4 minutes to read • [Edit Online](#)

This article describes initializing public client and confidential client applications using Microsoft Authentication Library for .NET (MSAL.NET). To learn more about the client application types and application configuration options, read the [overview](#).

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.

## Prerequisites

Before initializing an application, you first need to [register it](#) so that your app can be integrated with the Microsoft identity platform. After registration, you may need the following information (which can be found in the Azure portal):

- The client ID (a string representing a GUID)
- The identity provider URL (named the instance) and the sign-in audience for your application. These two parameters are collectively known as the authority.
- The tenant ID if you are writing a line of business application solely for your organization (also named single-tenant application).
- The application secret (client secret string) or certificate (of type `X509Certificate2`) if it's a confidential client app.
- For web apps, and sometimes for public client apps (in particular when your app needs to use a broker), you'll have also set the `redirectUri` where the identity provider will contact back your application with the security tokens.

## Ways to initialize applications

There are many different ways to instantiate client applications.

### Initializing a public client application 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();
```

### Initializing a confidential client application from code

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();

```

As you might know, in production, rather than using a client secret, you might want to share with Azure AD a certificate. The code would then be the following:

```

IConfidentialClientApplication app = ConfidentialClientApplicationBuilder.Create(clientId)
 .WithCertificate(certificate)
 .WithRedirectUri(redirectUri)
 .Build();

```

### Initializing a public client application from configuration options

The following code instantiates a public client application from a configuration object, which could be filled-in programmatically or read from a configuration file:

```

PublicClientApplicationOptions options = GetOptions(); // your own method
IPublicClientApplication app = PublicClientApplicationBuilder.CreateWithApplicationOptions(options)
 .Build();

```

### Initializing a confidential client application from configuration options

The same kind of pattern applies to confidential client applications. You can also add other parameters using `.WithXXX` modifiers (here a certificate).

```

ConfidentialClientApplicationOptions options = GetOptions(); // your own method
IConfidentialClientApplication app =
ConfidentialClientApplicationBuilder.CreateWithApplicationOptions(options)
 .WithCertificate(certificate)
 .Build();

```

## Builder modifiers

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

### Modifiers common to public and confidential client applications

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

MODIFIER	DESCRIPTION
<code>.WithAuthority()</code> 7 overrides	Sets the application default authority to an Azure AD authority, with the possibility of choosing the Azure Cloud, the audience, the tenant (tenant ID or domain name), or providing directly the authority URI.
<code>.WithAdfsAuthority(string)</code>	Sets the application default authority to be an ADFS authority.
<code>.WithB2CAuthority(string)</code>	Sets the application default authority to be an Azure AD B2C authority.
<code>.WithClientId(string)</code>	Overrides the client ID.

MODIFIER	DESCRIPTION
<code>.WithComponent(string)</code>	Sets the name of the library using MSAL.NET (for telemetry reasons).
<code>.WithDebugLoggingCallback()</code>	If called, the application will call <code>Debug.WriteLine</code> simply enabling debugging traces. See <a href="#">Logging</a> for more information.
<code>.WithExtraQueryParameters(IDictionary&lt;string, string&gt; eqp)</code>	Set the application level extra query parameters that will be sent in all authentication request. This is overridable at each token acquisition method level (with the same <code>.WithExtraQueryParameters</code> pattern ).
<code>.WithHttpClientFactory(IMsalHttpClientFactory httpClientFactory)</code>	Enables advanced scenarios such as configuring for an HTTP proxy, or to force MSAL to use a particular HttpClient (for instance in ASP.NET Core web apps/APIs).
<code>.WithLogging()</code>	If called, the application will call a callback with debugging traces. See <a href="#">Logging</a> for more information.
<code>.WithRedirectUri(string redirectUri)</code>	Overrides the default redirect URI. In the case of public client applications, this will be useful for scenarios involving the broker.
<code>.WithTelemetry(TelemetryCallback telemetryCallback)</code>	Sets the delegate used to send telemetry.
<code>.WithTenantId(string tenantId)</code>	Overrides the tenant ID, or the tenant description.

## Modifiers specific to Xamarin.iOS applications

The modifiers you can set on a public client application builder on Xamarin.iOS are:

MODIFIER	DESCRIPTION
<code>.WithIosKeychainSecurityGroup()</code>	<b>Xamarin.iOS only:</b> Sets the iOS key chain security group (for the cache persistence).

## Modifiers specific to confidential client applications

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

MODIFIER	DESCRIPTION
<code>.WithCertificate(X509Certificate2 certificate)</code>	Sets the certificate identifying the application with Azure AD.
<code>.WithClientSecret(string clientSecret)</code>	Sets the client secret (app password) identifying the application with Azure AD.

These modifiers are mutually exclusive. If you provide both, MSAL will throw a meaningful exception.

## Example of usage of modifiers

Let's assume that your application is a line-of-business application, which is only for your organization. Then you can write:

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
 .WithAadAuthority(AzureCloudInstance.AzurePublic, tenantId)
 .Build();
```

Where it becomes interesting is that programming for national clouds has now simplified. If you want your application to be a multi-tenant application in a national cloud, you could write, for instance:

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
 .WithAadAuthority(AzureCloudInstance.AzureUsGovernment, AadAuthorityAudience.AzureAdMultipleOrgs)
 .Build();
```

There is also an override for ADFS (ADFS 2019 is currently not supported):

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
 .WithAdfsAuthority("https://consoso.com/adfs")
 .Build();
```

Finally, if you are an Azure AD B2C developer, you can specify your tenant like this:

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
 .WithB2CAuthority("https://fabrikamb2c.b2clogin.com/tfp/{tenant}/{PolicySignInSignUp}")
 .Build();
```

# Confidential client assertions

10/23/2019 • 3 minutes to read • [Edit Online](#)

In order to prove their identity, confidential client applications exchange a secret with Azure AD. The secret can be:

- A client secret (application password).
- A certificate, which is used to build a signed assertion containing standard claims.

This secret can also be a signed assertion directly.

MSAL.NET has four methods to provide either credentials or assertions to the confidential client app:

- `.WithClientSecret()`
- `.WithCertificate()`
- `.WithClientAssertion()`
- `.WithClientClaims()`

## Signed assertions

A signed client assertion takes the form of a signed JWT with the payload containing the required authentication claims mandated by Azure AD, Base64 encoded. To use it:

```
string signedClientAssertion = ComputeAssertion();
app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
 .WithClientAssertion(signedClientAssertion)
 .Build();
```

The claims expected by Azure AD are:

CLAIM TYPE	VALUE	DESCRIPTION
aud	<a href="https://login.microsoftonline.com/{tenantId}/v2.0">https://login.microsoftonline.com/{tenantId}/v2.0</a>	The "aud" (audience) claim identifies the recipients that the JWT is intended for (here Azure AD) See [RFC 7519, Section 4.1.3]
exp	Thu Jun 27 2019 15:04:17 GMT+0200 (Romance Daylight Time)	The "exp" (expiration time) claim identifies the expiration time on or after which the JWT MUST NOT be accepted for processing. See [RFC 7519, Section 4.1.4]
iss	{ClientID}	The "iss" (issuer) claim identifies the principal that issued the JWT. The processing of this claim is application-specific. The "iss" value is a case-sensitive string containing a StringOrURI value. [RFC 7519, Section 4.1.1]

CLAIM TYPE	VALUE	DESCRIPTION
jti	(a Guid)	The "jti" (JWT ID) claim provides a unique identifier for the JWT. The identifier value MUST be assigned in a manner that ensures that there is a negligible probability that the same value will be accidentally assigned to a different data object; if the application uses multiple issuers, collisions MUST be prevented among values produced by different issuers as well. The "jti" claim can be used to prevent the JWT from being replayed. The "jti" value is a case-sensitive string. [RFC 7519, Section 4.1.7]
nbf	Thu Jun 27 2019 14:54:17 GMT+0200 (Romance Daylight Time)	The "nbf" (not before) claim identifies the time before which the JWT MUST NOT be accepted for processing. [RFC 7519, Section 4.1.5]
sub	{ClientID}	The "sub" (subject) claim identifies the subject of the JWT. The claims in a JWT are normally statements about the subject. The subject value MUST either be scoped to be locally unique in the context of the issuer or be globally unique. The See [RFC 7519, Section 4.1.2]

Here is an example of how to craft these claims:

```
private static IDictionary<string, string> GetClaims()
{
 //aud = https://login.microsoftonline.com/ + Tenant ID + /v2.0
 string aud = "https://login.microsoftonline.com/72f988bf-86f1-41af-hd4m-2d7cd011db47/v2.0";

 string ConfidentialClientID = "61dab2ba-145d-4b1b-8569-bf4b9aed5dhb" //client id
 const uint JwtToAadLifetimeInSeconds = 60 * 10; // Ten minutes
 DateTime validFrom = DateTime.UtcNow;
 var nbf = ConvertToTimeT(validFrom);
 var exp = ConvertToTimeT(validFrom + TimeSpan.FromSeconds(JwtToAadLifetimeInSeconds));

 return new Dictionary<string, string>()
 {
 { "aud", aud },
 { "exp", exp.ToString() },
 { "iss", ConfidentialClientID },
 { "jti", Guid.NewGuid().ToString() },
 { "nbf", nbf.ToString() },
 { "sub", ConfidentialClientID }
 };
}
```

Here is how to craft a signed client assertion:

```

string Encode(byte[] arg)
{
 char Base64PadCharacter = '=';
 char Base64Character62 = '+';
 char Base64Character63 = '/';
 char Base64UrlCharacter62 = '-';
 char Base64UrlCharacter63 = '_';

 string s = Convert.ToBase64String(arg);
 s = s.Split(Base64PadCharacter)[0]; // RemoveAccount any trailing padding
 s = s.Replace(Base64Character62, Base64UrlCharacter62); // 62nd char of encoding
 s = s.Replace(Base64Character63, Base64UrlCharacter63); // 63rd char of encoding

 return s;
}

string GetAssertion()
{
 //Signing with SHA-256
 string rsaSha256Signature = "http://www.w3.org/2001/04/xmldsig-more#rsa-sha256";
 X509Certificate2 certificate = ReadCertificate(config.CertificateName);

 //Create RSACryptoServiceProvider
 var x509Key = new X509AsymmetricSecurityKey(certificate);
 var privateKeyXmlParams = certificate.PrivateKey.XmlString(true);
 var rsa = new RSACryptoServiceProvider();
 rsa.FromXmlString(privateKeyXmlParams);

 //alg represents the desired signing algorithm, which is SHA-256 in this case
 //kid represents the certificate thumbprint
 var header = new Dictionary<string, string>()
 {
 { "alg", "RS256" },
 { "kid", Encode(Certificate.GetCertHash()) }
 };

 //Please see the previous code snippet on how to craft claims for the GetClaims() method
 string token = Encode(Encoding.UTF8.GetBytes(JObject.FromObject(header).ToString()) + "." +
 Encode(Encoding.UTF8.GetBytes(JObject.FromObject(GetClaims()))));

 string signature = Encode(rsa.SignData(Encoding.UTF8.GetBytes(token), new SHA256Cng()));
 string SignedAssertion = string.Concat(token, ".", signature);
 return SignedAssertion;
}

```

## WithClientClaims

```
WithClientClaims(X509Certificate2 certificate, IDictionary<string, string> claimsToSign, bool mergeWithDefaultClaims = true)
```

by default will produce a signed assertion containing the claims expected by Azure AD plus additional client claims that you want to send. Here is a code snippet on how to do that.

```

string ipAddress = "192.168.1.2";
X509Certificate2 certificate = ReadCertificate(config.CertificateName);
app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
 .WithAuthority(new Uri(config.Authority))
 .WithClientClaims(certificate,
 new Dictionary<string, string> { {
 "client_ip", ipAddress } })
 .Build();

```

If one of the claims in the dictionary that you pass in is the same as one of the mandatory claims, the additional claim's value will be taken into account. It will override the claims computed by MSAL.NET.

If you want to provide your own claims, including the mandatory claims expected by Azure AD, pass in `false` for the `mergeWithDefaultClaims` parameter.

# Initialize client applications using MSAL.js

10/30/2019 • 5 minutes to read • [Edit Online](#)

This article describes initializing Microsoft Authentication Library for JavaScript (MSAL.js) with an instance of a user-agent application. The user-agent application is a form of public client application in which the client code is executed in a user-agent such as a web browser. These clients do not store secrets, since the browser context is openly accessible. To learn more about the client application types and application configuration options, read the [overview](#).

## Prerequisites

Before initializing an application, you first need to [register it with the Azure portal](#) so that your app can be integrated with the Microsoft identity platform. After registration, you may need the following information (which can be found in the Azure portal):

- The client ID (a string representing a GUID for your application)
- The identity provider URL (named the instance) and the sign-in audience for your application. These two parameters are collectively known as the authority.
- The tenant ID if you are writing a line-of-business application solely for your organization (also named single-tenant application).
- For web apps, you'll have to also set the redirectUri where the identity provider will return to your application with the security tokens.

## Initializing applications

You can use MSAL.js as follows in a plain JavaScript/TypeScript application. Initialize MSAL authentication context by instantiating `UserAgentApplication` with a configuration object. The minimum required config to initialize MSAL.js is the `clientId` of your application which you should get from the application registration portal.

For authentication methods with redirect flows (`loginRedirect` and `acquireTokenRedirect`), you will need to explicitly register a callback for success or error through `handleRedirectCallback()` method. This is needed since redirect flows do not return promises as the methods with a pop-up experience do.

```
// Configuration object constructed
const config = {
 auth: {
 clientId: "abcd-ef12-gh34-ikkl-ashdjh1hsdg"
 }
}

// create UserAgentApplication instance
const myMSALObj = new UserAgentApplication(config);

function authCallback(error, response) {
 //handle redirect response
}

// (optional when using redirect methods) register redirect call back for Success or Error
myMSALObj.handleRedirectCallback(authCallback);
```

MSAL.js is designed to have a single instance and configuration of the `UserAgentApplication` to represent a single authentication context. Multiple instances are not recommended as they cause conflicting cache entries and

behavior in the browser.

## Configuration options

MSAL.js has a configuration object shown below that provides a grouping of configurable options available for creating an instance of `UserAgentApplication`.

```
type storage = "localStorage" | "sessionStorage";

// Protocol Support
export type AuthOptions = {
 clientId: string;
 authority?: string;
 validateAuthority?: boolean;
 redirectUri?: string | (() => string);
 postLogoutRedirectUri?: string | (() => string);
 navigateToLoginRequestUrl?: boolean;
};

// Cache Support
export type CacheOptions = {
 cacheLocation?: CacheLocation;
 storeAuthStateInCookie?: boolean;
};

// Library support
export type SystemOptions = {
 logger?: Logger;
 loadFrameTimeout?: number;
 tokenRenewalOffsetSeconds?: number;
 navigateFrameWait?: number;
};

// Developer App Environment Support
export type FrameworkOptions = {
 isAngular?: boolean;
 unprotectedResources?: Array<string>;
 protectedResourceMap?: Map<string, Array<string>>;
};

// Configuration Object
export type Configuration = {
 auth: AuthOptions,
 cache?: CacheOptions,
 system?: SystemOptions,
 framework?: FrameworkOptions
};
```

Below is the total set of configurable options that are supported currently in the config object:

- **clientID**: Required. The clientID of your application, you should get this from the application registration portal.
- **authority**: Optional. A URL indicating a directory that MSAL can request tokens from. Default value is:  
`https://login.microsoftonline.com/common`.
  - In Azure AD, it is of the form `https://<instance>/<audience>`, where `<instance>` is the identity provider domain (for example, `https://login.microsoftonline.com`) and `<audience>` is an identifier representing the sign-in audience. This can be the following values:
    - `https://login.microsoftonline.com/<tenant>` - tenant is a domain associated to the tenant, such as `contoso.onmicrosoft.com`, or the GUID representing the `TenantID` property of the directory used only to sign in users of a specific organization.

- `https://login.microsoftonline.com/common` - Used to sign in users with work and school accounts or a Microsoft personal account.
- `https://login.microsoftonline.com/organizations/` - Used to sign in users with work and school accounts.
- `https://login.microsoftonline.com/consumers/` - Used to sign in users with only personal Microsoft account (live).
- In Azure AD B2C, it is of the form `https://<instance>/tfp/<tenant>/<policyName>/`, where instance is the Azure AD B2C domain, tenant is the name of the Azure AD B2C tenant, policyName is the name of the B2C policy to apply.
- **validateAuthority**: Optional. Validate the issuer of tokens. Default is `true`. For B2C applications, since the authority value is known and can be different per policy, the authority validation will not work and has to be set to `false`.
- **redirectUri**: Optional. 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. Defaults to `window.location.href`.
- **postLogoutRedirectUri**: Optional. Redirects the user to `postLogoutRedirectUri` after sign out. The default is `redirectUri`.
- **navigateToLoginRequestUrl**: Optional. Ability to turn off default navigation to start page after login. Default is true. This is used only for redirect flows.
- **cacheLocation**: Optional. Sets browser storage to either `localStorage` or `sessionStorage`. The default is `sessionStorage`.
- **storeAuthStateInCookie**: Optional. This flag was introduced in MSAL.js v0.2.2 as a fix for the [authentication loop issues](#) on Microsoft Internet Explorer and Microsoft Edge. Enable the flag `storeAuthStateInCookie` to true to take advantage of this fix. When this is enabled, MSAL.js will store the auth request state required for validation of the auth flows in the browser cookies. By default this flag is set to `false`.
- **logger**: Optional. A Logger object with a callback instance that can be provided by the developer to consume and publish logs in a custom manner. For details on passing logger object, see [logging with msal.js](#).
- **loadFrameTimeout**: Optional. The number of milliseconds of inactivity before a token renewal response from Azure AD should be considered timed out. Default is 6 seconds.
- **tokenRenewalOffsetSeconds**: Optional. The number of milliseconds which sets the window of offset needed to renew the token before expiry. Default is 300 milliseconds.
- **navigateFrameWait**: Optional. The number of milliseconds which sets the wait time before hidden iframes navigate to their destination. Default is 500 milliseconds.

These are only applicable to be passed down from the MSAL Angular wrapper library:

- **unprotectedResources**: Optional. Array of URLs that are unprotected resources. MSAL will not attach a token to outgoing requests that have these URL. Defaults to `null`.
- **protectedResourceMap**: Optional. This is mapping of resources to scopes used by MSAL for automatically attaching access tokens in web API calls. A single access token is obtained for the resource. So you can map a specific resource path as follows: `{"https://graph.microsoft.com/v1.0/me", ["user.read"]}`, or the app URL of the resource as: `{"https://graph.microsoft.com/", ["user.read", "mail.send"]}`. This is required for CORS calls. Defaults to `null`.

# Handle MSAL exceptions and errors

10/30/2019 • 15 minutes to read • [Edit Online](#)

This article gives an overview of the different types of errors and recommendations for handling common sign-in errors.

## MSAL error handling basics

Exceptions in Microsoft Authentication Library (MSAL) are intended for app developers to troubleshoot and not for displaying to end users. Exception messages are not localized.

When processing exceptions and errors, you can use the exception type itself and the error code to distinguish between exceptions. For a list of error codes, see [Authentication and authorization error codes](#).

During silent or interactive token acquisition, apps may come across errors during the sign-in experience such as errors about consents, Conditional Access (MFA, Device Management, Location-based restrictions), token issuance and redemption, and user properties.

## MSAL for iOS and macOS errors

The complete list of errors is listed in [MSALError enum](#).

All MSAL produced errors are returned with `MSALErrorDomain` domain.

For system errors, MSAL returns the original `NSError` from the system API. For example, if token acquisition fails because of a lack of network connectivity, MSAL returns an error with the `NSURLLErrorDomain` domain and `NSURLErrorNotConnectedToInternet` code.

We recommend that you handle at least the following two MSAL errors on the client side:

- `MSALErrorInteractionRequired`: The user must do an interactive request. There are many conditions that can lead to this error such as an expired authentication session or the need for additional authentication requirements. Call the MSAL interactive token acquisition API to recover.
- `MSALErrorServerDeclinedScopes`: Some or all scopes were declined. Decide whether to continue with only the granted scopes, or stop the sign-in process.

### NOTE

The `MSALInternalError` enum should only be used for reference and debugging. Do not try to automatically handle these errors at runtime. If your app encounters any of the errors that fall under `MSALInternalError`, you may want to show a generic user facing message explaining what happened.

For example, `MSALInternalErrorBrokerResponseNotReceived` means that user didn't complete authentication and manually returned to the app. In this case, your app should show a generic error message explaining that authentication didn't complete and suggest that they try to authenticate again.

The following Objective-C sample code demonstrates best practices for handling some common error conditions:

Objective-C

```
MSALInteractiveTokenParameters *interactiveParameters = ...;
MSALSilentTokenParameters *silentParameters = ...;

MSALCompletionBlock completionBlock;
__block __weak MSALCompletionBlock weakCompletionBlock;

weakCompletionBlock = completionBlock = ^(MSALResult *result, NSError *error)
{
 if (!error)
 {
 // Use result.accessToken
 NSString *accessToken = result.accessToken;
 return;
 }

 if ([error.domain isEqualToString:MSALErrorDomain])
 {
 switch (error.code)
 {
 case MSALErrorInteractionRequired:
 {
 // Interactive auth will be required
 [application acquireTokenWithParameters:interactiveParameters
 completionBlock:weakCompletionBlock];

 break;
 }

 case MSALErrorServerDeclinedScopes:
 {
 // These are list of granted and declined scopes.
 NSArray *grantedScopes = error.userInfo[MSALGrantedScopesKey];
 NSArray *declinedScopes = error.userInfo[MSALDeclinedScopesKey];

 // To continue acquiring token for granted scopes only, do the following
 silentParameters.scopes = grantedScopes;
 [application acquireTokenSilentWithParameters:silentParameters
 completionBlock:weakCompletionBlock];

 // Otherwise, instead, handle error fittingly to the application context
 }
 }
 }
}
```

```

 break;
 }

 case MSALErrorServerProtectionPoliciesRequired:
 {
 // Integrate the Intune SDK and call the
 // remediateComplianceForIdentity:silent: API.
 // Handle this error only if you integrated Intune SDK.
 // See more info here: https://aka.ms/intuneMAMSDK

 break;
 }

 case MSALErrorUserCanceled:
 {
 // The user cancelled the web auth session.
 // You may want to ask the user to try again.
 // Handling of this error is optional.

 break;
 }

 case MSALErrorInternal:
 {
 // Log the error, then inspect the MSALInternalErrorCodeKey
 // in the userInfo dictionary.
 // Display generic error message to the end user
 // More detailed information about the specific error
 // under MSALInternalErrorCodeKey can be found in MSALInternalError enum.
 NSLog(@"%@", error);

 break;
 }

 default:
 NSLog(@"%@", error);

 break;
 }

 return;
}

// Handle no internet connection.
if ([error.domain isEqualToString:NSUTFURLConnectionDomain] && error.code == NSURLConnectionNotConnectedToInternet)
{
 NSLog(@"No internet connection.");
 return;
}

// Other errors may require trying again later,
// or reporting authentication problems to the user.
NSLog(@"%@", error);
};

// Acquire token silently
[application acquireTokenSilentWithParameters:silentParameters
 completionBlock:completionBlock];

// or acquire it interactively.
[application acquireTokenWithParameters:interactiveParameters
 completionBlock:completionBlock];

```

Swift

```

let interactiveParameters: MSALInteractiveTokenParameters = ...
let silentParameters: MSALSilentTokenParameters = ...

var completionBlock: MSALCompletionBlock!
completionBlock = { (result: MSALResult?, error: Error?) in

 if let result = result
 {
 // Use result.accessToken
 let accessToken = result.accessToken
 return
 }

 guard let error = error as NSError? else { return }

 if error.domain == MSLErrorDomain, let errorCode = MSALError(rawValue: error.code)
 {
 switch errorCode
 {
 case .interactionRequired:
 // Interactive auth will be required
 application.acquireToken(with: interactiveParameters, completionBlock: completionBlock)

 case .serverDeclinedScopes:
 let grantedScopes = error.userInfo[MSALGrantedScopesKey]
 let declinedScopes = error.userInfo[MSALDeclinedScopesKey]

 if let scopes = grantedScopes as? [String] {
 silentParameters.scopes = scopes
 application.acquireTokenSilent(with: silentParameters, completionBlock: completionBlock)
 }

 case .serverProtectionPoliciesRequired:
 // Integrate the Intune SDK and call the
 // remediateComplianceForIdentity:silent: API.
 // Handle this error only if you integrated Intune SDK
 // See more info here: https://aka.ms/intuneMAMSDK
 break

 case .userCanceled:
 // The user cancelled the web auth session.
 // You may want to ask the user to try again.
 // Handling of this error is optional.
 break

 case .internal:
 // Log the error, then inspect the MSALInternalErrorCodeKey
 // in the userInfo dictionary.
 // Display generic error message to the end user
 // More detailed information about the specific error
 // under MSALInternalErrorCodeKey can be found in MSALInternalError enum.
 print("Failed with error \(error);")

 default:
 print("Failed with unknown MSAL error \(error)")
 }
 }

 // Handle no internet connection.
 if error.domain == NSURLErrorDomain && error.code == NSURLErrorNotConnectedToInternet
 {
 print("No internet connection.")
 return
 }

 // Other errors may require trying again later,
 // or reporting authentication problems to the user.
 print("Failed with error \(error);")
}

// Acquire token silently
application.acquireToken(with: interactiveParameters, completionBlock: completionBlock)

// or acquire it interactively.
application.acquireTokenSilent(with: silentParameters, completionBlock: completionBlock)

```

## .NET exceptions

When processing exceptions, you can use the exception type itself and the `ErrorCode` member to distinguish between exceptions. `ErrorCode` values are constants of type [MsalError](#).

You can also have a look at the fields of [MsalClientException](#), [MsalServiceException](#), and [MsalUIRequiredException](#).

If [MsalServiceException](#) is thrown, try [Authentication and authorization error codes](#) to see if the code is listed there.

### Common exceptions

Here are the common exceptions that might be thrown and some possible mitigations:

EXCEPTION	ERROR CODE	MITIGATION
-----------	------------	------------

EXCEPTION	ERROR CODE	MITIGATION
<a href="#">MsalUiRequiredException</a>	AADSTS65001: The user or administrator has not consented to use the application with ID '{appId}' named '{appName}'. Send an interactive authorization request for this user and resource.	You need to get user consent first. If you aren't using .NET Core (which doesn't have any Web UI), call (once only) <code>AcquireTokenInteractive</code> . If you are using .NET core or don't want to do an <code>AcquireTokenInteractive</code> , the user can navigate to a URL to give consent: <a href="https://login.microsoftonline.com/common/oauth2/v2.0/authorize?client_id={client_id}&amp;response_type=code&amp;scope=user.read">https://login.microsoftonline.com/common/oauth2/v2.0/authorize?client_id={client_id}&amp;response_type=code&amp;scope=user.read</a> . to call <code>AcquireTokenInteractive</code> : <code>app.AcquireTokenInteractive(scopes).WithAccount(account).WithPrompt(prompt)</code>
<a href="#">MsalUiRequiredException</a>	AADSTS50079: The user is required to use multi-factor authentication (MFA).	There is no mitigation. If MFA is configured for your tenant and Azure Active Directory (AAD) decides to enforce it, you need to fallback to an interactive flow such as <code>AcquireTokenInteractive</code> or <code>AcquireTokenByDeviceCode</code> .
<a href="#">MsalServiceException</a>	AADSTS90010: The grant type isn't supported over the <code>/common</code> or <code>/consumers</code> endpoints. Use the <code>/organizations</code> or tenant-specific endpoint. You used <code>/common</code> .	As explained in the message from Azure AD, the authority needs to have a tenant or otherwise <code>/organizations</code> .
<a href="#">MsalServiceException</a>	AADSTS70002: The request body must contain the following parameter: <code>client_secret</code> or <code>client_assertion</code> .	This exception can be thrown if your application was not registered as a public client application in Azure AD. In the Azure portal, edit the manifest for your application and set <code>allowPublicClient</code> to <code>true</code> .
<a href="#">MsalClientException</a>	<code>unknown_user Message</code> : Could not identify logged in user	The library was unable to query the current Windows logged-in user or this user isn't AD or AAD joined (workplace joined users aren't supported). Mitigation 1: on UWP, check that the application has the following capabilities: Enterprise Authentication, Private Networks (Client and Server), User Account Information. Mitigation 2: Implement your own logic to fetch the username (for example, <code>john@contoso.com</code> ) and use the <code>AcquireTokenByIntegratedWindowsAuth</code> form that takes in the username.
<a href="#">MsalClientException</a>	<code>integrated_windows_auth_not_supported_managed_user</code>	This method relies on a protocol exposed by Active Directory (AD). If a user was created in Azure Active Directory without AD backing ("managed" user), this method will fail. Users created in AD and backed by AAD ("federated" users) can benefit from this non-interactive method of authentication. Mitigation: Use interactive authentication.

#### [MsalUiRequiredException](#)

One of common status codes returned from MSAL.NET when calling `AcquireTokenSilent()` is `MsalError.InvalidGrantError`. This status code means that the application should call the authentication library again, but in interactive mode (`AcquireTokenInteractive` or `AcquireTokenByDeviceCodeFlow` for public client applications, and do a challenge in Web apps). This is because additional user interaction is required before authentication token can be issued.

Most of the time when `AcquireTokenSilent` fails, it is because the token cache doesn't have tokens matching your request. Access tokens expire in 1 hour, and `AcquireTokenSilent` will try to fetch a new one based on a refresh token (in OAuth2 terms, this is the "Refresh Token" flow). This flow can also fail for various reasons, for example if a tenant admin configures more stringent login policies.

The interaction aims at having the user do an action. Some of those conditions are easy for users to resolve (for example, accept Terms of Use with a single click), and some can't be resolved with the current configuration (for example, the machine in question needs to connect to a specific corporate network). Some help the user setting-up Multi-factor authentication, or install Microsoft Authenticator on their device.

#### [MsalUiRequiredException](#) classification enumeration

MSAL exposes a `Classification` field, which you can read to provide a better user experience, for example to tell the user that their password expired or that they'll need to provide consent to use some resources. The supported values are part of the `UiRequiredExceptionClassification` enum:

CLASSIFICATION	MEANING	RECOMMENDED HANDLING
BasicAction	Condition can be resolved by user interaction during the interactive authentication flow.	Call <code>AcquireTokenInteractive()</code> .
AdditionalAction	Condition can be resolved by additional remedial interaction with the system, outside of the interactive authentication flow.	Call <code>AcquireTokenInteractive()</code> to show a message that explains the remedial action. Calling application may choose to hide flows that require <code>additional_action</code> if the user is unlikely to complete the remedial action.
MessageOnly	Condition can't be resolved at this time. Launching interactive authentication flow will show a message explaining the condition.	Call <code>AcquireTokenInteractive()</code> to show a message that explains the condition. <code>AcquireTokenInteractive()</code> will return <code>UserCanceled</code> error after the user reads the message and closes the window. Calling application may choose to hide flows that result in <code>message_only</code> if the user is unlikely to benefit from the message.

CLASSIFICATION	MEANING	RECOMMENDED HANDLING
ConsentRequired	User consent is missing, or has been revoked.	Call AcquireTokenInteractively() for user to give consent.
UserPasswordExpired	User's password has expired.	Call AcquireTokenInteractively() so that user can reset their password.
PromptNeverFailed	Interactive Authentication was called with the parameter prompt=never, forcing MSAL to rely on browser cookies and not to display the browser. This has failed.	Call AcquireTokenInteractively() without Prompt.None
AcquireTokenSilentFailed	MSAL SDK doesn't have enough information to fetch a token from the cache. This can be because no tokens are in the cache or an account wasn't found. The error message has more details.	Call AcquireTokenInteractively().
None	No further details are provided. Condition may be resolved by user interaction during the interactive authentication flow.	Call AcquireTokenInteractively().

## Code example

```

AuthenticationResult res;
try
{
 res = await application.AcquireTokenSilent(scopes, account)
 .ExecuteAsync();
}
catch (MsalUiRequiredException ex) when (ex.ErrorCode == MsalError.InvalidGrantError)
{
 switch (ex.Classification)
 {
 case UiRequiredExceptionClassification.None:
 break;
 case UiRequiredExceptionClassification.MessageOnly:
 // You might want to call AcquireTokenInteractive(). Azure AD will show a message
 // that explains the condition. AcquireTokenInteractively() will return UserCanceled error
 // after the user reads the message and closes the window. The calling application may choose
 // to hide features or data that result in message_only if the user is unlikely to benefit
 // from the message
 try
 {
 res = await application.AcquireTokenInteractive(scopes)
 .ExecuteAsync();
 }
 catch (MsalClientException ex2) when (ex2.ErrorCode == MsalError.AuthenticationCanceledError)
 {
 // Do nothing. The user has seen the message
 }
 break;

 case UiRequiredExceptionClassification.BasicAction:
 // Call AcquireTokenInteractive() so that the user can, for instance accept terms
 // and conditions

 case UiRequiredExceptionClassification.AdditionalAction:
 // You might want to call AcquireTokenInteractive() to show a message that explains the remedial action.
 // The calling application may choose to hide flows that require additional_action if the user
 // is unlikely to complete the remedial action (even if this means a degraded experience)

 case UiRequiredExceptionClassification.ConsentRequired:
 // Call AcquireTokenInteractive() for user to give consent.

 case UiRequiredExceptionClassification.UserPasswordExpired:
 // Call AcquireTokenInteractive() so that user can reset their password

 case UiRequiredExceptionClassification.PromptNeverFailed:
 // You used WithPrompt(Prompt.Never) and this failed

 case UiRequiredExceptionClassification.AcquireTokenSilentFailed:
 default:
 // May be resolved by user interaction during the interactive authentication flow.
 res = await application.AcquireTokenInteractive(scopes)
 .ExecuteAsync(); break;
 }
}

```

## JavaScript errors

MSAL.js provides error objects that abstract and classify the different types of common errors. It also provides interface to access specific details of the errors such as error messages to handle them appropriately.

### Error object

```

export class AuthError extends Error {
 // This is a short code describing the error
 errorCode: string;
 // This is a descriptive string of the error,
 // and may also contain the mitigation strategy
 errorMessage: string;
 // Name of the error class
 this.name = "AuthError";
}

```

By extending the error class, you have access to the following properties:

- **AuthError.message**: Same as the errorMessage.
- **AuthError.stack**: Stack trace for thrown errors. Allows tracing to origin point of error.

## Error Types

The following error types are available:

- `AuthError`: Base error class for the MSAL.js library, also used for unexpected errors.
- `ClientAuthError`: Error class, which denotes an issue with Client authentication. Most errors that come from the library will be ClientAuthErrors. These errors result from things like calling a login method when login is already in progress, the user cancels the login, and so on.
- `ClientConfigurationError`: Error class, extends `ClientAuthError` thrown before requests are made when the given user config parameters are malformed or missing.
- `ServerError`: Error class, represents the error strings sent by the authentication server. These may be errors such as invalid request formats or parameters, or any other errors that prevent the server from authenticating or authorizing the user.
- `InteractionRequiredAuthError`: Error class, extends `ServerError` to represent server errors, which require an interactive call. This error is thrown by `acquireTokenSilent` if the user is required to interact with the server to provide credentials or consent for authentication/authorization. Error codes include `"interaction_required"`, `"login_required"`, and `"consent_required"`.

For error handling in authentication flows with redirect methods (`loginRedirect`, `acquireTokenRedirect`), you'll need to register the callback, which is called with success or failure after the redirect using `handleRedirectCallback()` method as follows:

```

function authCallback(error, response) {
 //handle redirect response
}

var myMSALObj = new Msal.UserAgentApplication(msalConfig);

// Register Callbacks for redirect flow
myMSALObj.handleRedirectCallback(authCallback);
myMSALObj.acquireTokenRedirect(request);

```

The methods for pop-up experience (`loginPopup`, `acquireTokenPopup`) return promises, so you can use the promise pattern (`.then` and `.catch`) to handle them as shown:

```

myMSALObj.acquireTokenPopup(request).then(
 function (response) {
 // success response
 }).catch(function (error) {
 console.log(error);
 });

```

## Interaction required, errors

An error is returned when you attempt to use a non-interactive method of acquiring a token such as `acquireTokenSilent`, but MSAL couldn't do it silently.

Possible reasons are:

- you need to sign in
- you need to consent
- you need to go through a multi-factor authentication experience.

The remediation is to call an interactive method such as `acquireTokenPopup` or `acquireTokenRedirect`:

```

// Request for Access Token
myMSALObj.acquireTokenSilent(request).then(function (response) {
 // call API
}).catch(function (error) {
 // call acquireTokenPopup in case of acquireTokenSilent failure
 // due to consent or interaction required
 if (error.errorCode === "consent_required"
 || error.errorCode === "interaction_required"
 || error.errorCode === "login_required") {
 myMSALObj.acquireTokenPopup(request).then(
 function (response) {
 // call API
 }).catch(function (error) {
 console.log(error);
 });
 }
});

```

## Conditional Access and claims challenges

When getting tokens silently, your application may receive errors when a [Conditional Access claims challenge](#) such as MFA policy is required by an API you're trying to access.

The pattern for handling this error is to interactively acquire a token using MSAL. Interactively acquiring a token prompts the user and gives them the opportunity to satisfy the required Conditional Access policy.

In certain cases when calling an API requiring Conditional Access, you can receive a claims challenge in the error from the API. For instance if the Conditional Access policy is to have a managed device (Intune) the error will be something like [AADSTS53000: Your device is required to be managed to access this resource](#) or something similar. In this case, you can pass the claims in the acquire token call so that the user is prompted to satisfy the appropriate policy.

### .NET

When calling an API requiring Conditional Access from MSAL.NET, your application will need to handle claim challenge exceptions. This will appear as an [MsalServiceException](#) where the [Claims](#) property won't be empty.

To handle the claim challenge, you'll need to use the `.WithClaim()` method of the `PublicClientApplicationBuilder` class.

### JavaScript

When getting tokens silently (using `acquireTokenSilent()`) using MSAL.js, your application may receive errors when a [Conditional Access claims challenge](#) such as MFA policy is required by an API you're trying to access.

The pattern to handle this error is to make an interactive call to acquire token in MSAL.js such as `acquireTokenPopup` or `acquireTokenRedirect` as in the following example:

```

myMSALObj.acquireTokenSilent(accessTokenRequest).then(function (accessTokenResponse) {
 // call API
}).catch(function (error) {
 if (error instanceof InteractionRequiredAuthError) {
 // Extract claims from error message
 accessTokenRequest.claimsRequest = extractClaims(error.errorMessage);
 // call acquireTokenPopup in case of InteractionRequiredAuthError failure
 myMSALObj.acquireTokenPopup(accessTokenRequest).then(function (accessTokenResponse) {
 // call API
 }).catch(function (error) {
 console.log(error);
 });
 }
});

```

Interactively acquiring the token prompts the user and gives them the opportunity to satisfy the required Conditional Access policy.

When calling an API requiring Conditional Access, you can receive a claims challenge in the error from the API. In this case, you can pass the claims returned in the error to the `claimsRequest` field of the `AuthenticationParameters.ts` class to satisfy the appropriate policy.

See [Requesting Additional Claims](#) for more detail.

### MSAL for iOS and macOS

MSAL for iOS and macOS allows you to request specific claims in both interactive and silent token acquisition scenarios.

To request custom claims, specify `claimsRequest` in `MSALSilentTokenParameters` or `MSALInteractiveTokenParameters`.

See [Request custom claims using MSAL for iOS and macOS](#) for more info.

## Retrying after errors and exceptions

You're expected to implement your own retry policies when calling MSAL. MSAL makes HTTP calls to the AAD service, and occasional failures can occur, for example the network can go down or the server is overloaded.

### HTTP error codes 500-600

MSAL.NET implements a simple retry-once mechanism for errors with HTTP error codes 500-600.

### HTTP 429

When the Service Token Server (STS) is overloaded with too many requests, it returns HTTP error 429 with a hint about how long until you can try again in the `Retry-After` response field.

## .NET

[MsalServiceException](#) surfaces `System.Net.Http.Headers.HttpResponseHeaders` as a property `namedHeaders`. You can use additional information from the error code to improve the reliability of your applications. In the case described, you can use the `RetryAfter` property (of type `RetryConditionHeaderValue`) and compute when to retry.

Here is an example for a daemon application using the client credentials flow. You can adapt this to any of the methods for acquiring a token.

```
do
{
 retry = false;
 TimeSpan? delay;
 try
 {
 result = await publicClientApplication.AcquireTokenForClient(scopes, account)
 .ExecuteAsync();
 }
 catch (MsalServiceException serviceException)
 {
 if (ex.ErrorCode == "temporarily_unavailable")
 {
 RetryConditionHeaderValue retryAfter = serviceException.Headers.RetryAfter;
 if (retryAfter.Delta.HasValue)
 {
 delay = retryAfter.Delta;
 }
 else if (retryAfter.Date.HasValue)
 {
 delay = retryAfter.Date.Value.Offset;
 }
 }
 }

 ...
 if (delay.HasValue)
 {
 Thread.Sleep((int)delay.Value.TotalMilliseconds); // sleep or other
 retry = true;
 }
} while (retry);
```

# Logging in MSAL applications

11/10/2019 • 7 minutes to read • [Edit Online](#)

Microsoft Authentication Library (MSAL) apps generate log messages that can help diagnose issues. An app can configure logging with a few lines of code, and have custom control over the level of detail and whether or not personal and organizational data is logged. We recommend you create an MSAL logging callback and provide a way for users to submit logs when they have authentication issues.

## Logging levels

MSAL provides several levels of logging detail:

- Error: Indicates something has gone wrong and an error was generated. Use for debugging and identifying problems.
- Warning: There hasn't necessarily been an error or failure, but are intended for diagnostics and pinpointing problems.
- Info: MSAL will log events intended for informational purposes not necessarily intended for debugging.
- Verbose: Default. MSAL logs the full details of library behavior.

## Personal and organizational data

By default, the MSAL logger doesn't capture any highly sensitive personal or organizational data. The library provides the option to enable logging personal and organizational data if you decide to do so.

## Logging in MSAL.NET

### NOTE

See the [MSAL.NET wiki](#) for samples of MSAL.NET logging and more.

In MSAL 3.x, logging is set per application at app creation using the `.withLogging` builder modifier. This method takes optional parameters:

- `Level` enables you to decide which level of logging you want. Setting it to Errors will only get errors
- `PiiLoggingEnabled` enables you to log personal and organizational data if set to true. By default this is set to false, so that your application does not log personal data.
- `LogCallback` is set to a delegate that does the logging. If `PiiLoggingEnabled` is true, this method will receive the messages twice: once with the `containsPii` parameter equals false and the message without personal data, and a second time with the `containsPii` parameter equals to true and the message might contain personal data. In some cases (when the message does not contain personal data), the message will be the same.
- `DefaultLoggingEnabled` enables the default logging for the platform. By default it's false. If you set it to true it uses Event Tracing in Desktop/UWP applications, NSLog on iOS and logcat on Android.

```

class Program
{
 private static void Log(LogLevel level, string message, bool containsPii)
 {
 if (containsPii)
 {
 Console.ForegroundColor = ConsoleColor.Red;
 }
 Console.WriteLine($"{level} {message}");
 Console.ResetColor();
 }

 static void Main(string[] args)
 {
 var scopes = new string[] { "User.Read" };

 var application = PublicClientApplicationBuilder.Create("<clientID>")
 .WithLogging(Log, LogLevel.Info, true)
 .Build();

 AuthenticationResult result = application.AcquireTokenInteractive(scopes)
 .ExecuteAsync().Result;
 }
}

```

## Logging in MSAL for Android using Java

Turn logging on at app creation by creating a logging callback. The callback takes these parameters:

- `tag` is a string passed to the callback by the library. It is associated with the log entry and can be used to sort logging messages.
- `logLevel` enables you to decide which level of logging you want. The supported log levels are: `Error`, `Warning`, `Info`, and `Verbose`.
- `message` is the content of the log entry.
- `containsPII` specifies whether messages containing personal data, or organizational data are logged. By default, this is set to false, so that your application doesn't log personal data. If `containsPII` is `true`, this method will receive the messages twice: once with the `containsPII` parameter set to `false` and the `message` without personal data, and a second time with the `containsPii` parameter set to `true` and the message might contain personal data. In some cases (when the message does not contain personal data), the message will be the same.

```

private StringBuilder mLogs;

mLogs = new StringBuilder();
Logger.getInstance().setExternalLogger(new ILoggerCallback()
{
 @Override
 public void log(String tag, Logger.LogLevel logLevel, String message, boolean containsPII)
 {
 mLogs.append(message).append('\n');
 }
});

```

By default, the MSAL logger will not capture any personal identifiable information or organizational identifiable information. To enable the logging of personal identifiable information or organizational identifiable information:

```
Logger.getInstance().setEnablePII(true);
```

To disable logging personal data and organization data:

```
Logger.getInstance().setEnablePII(false);
```

By default logging to logcat is disabled. To enable:

```
Logger.getInstance().setEnableLogcatLog(true);
```

## Logging in MSAL.js

Enable logging in MSAL.js (Javascript) by passing a logger object during the configuration for creating a `UserAgentApplication` instance. This logger object has the following properties:

- `localCallback`: a Callback instance that can be provided by the developer to consume and publish logs in a custom manner. Implement the localCallback method depending on how you want to redirect logs.
- `level` (optional): the configurable log level. The supported log levels are: `Error`, `Warning`, `Info`, and `Verbose`. The default is `Info`.
- `piiLoggingEnabled` (optional): if set to true, logs personal and organizational data. By default this is false so that your application doesn't log personal data. Personal data logs are never written to default outputs like Console, Logcat, or NSLog.
- `correlationId` (optional): a unique identifier, used to map the request with the response for debugging purposes. Defaults to RFC4122 version 4 guid (128 bits).

```
function loggerCallback(logLevel, message, containsPii) {
 console.log(message);
}

var msalConfig = {
 auth: {
 clientId: "<Enter your client id>",
 },
 system: {
 logger: new Msal.Logger(
 loggerCallback ,{
 level: Msal.LogLevel.Verbose,
 piiLoggingEnabled: false,
 correlationId: '1234'
 }
)
 }
}

var UserAgentApplication = new Msal.UserAgentApplication(msalConfig);
```

## Logging in MSAL for iOS and macOS

Set a callback to capture MSAL logging and incorporate it in your own application's logging. The signature for the callback looks like this:

```

/*!
 * The LogCallback block for the MSAL logger
 *
 @param level The level of the log message
 @param message The message being logged
 @param containsPII If the message might contain Personally Identifiable Information (PII)
 this will be true. Log messages possibly containing PII will not be
 sent to the callback unless PIILoggingEnabled is set to YES on the
 logger.

*/
typedef void (^MSALLogCallback)(MSALLogLevel level, NSString *message, BOOL containsPII);

```

For example:

Objective-C

```

[MSALGlobalConfig.loggerConfig.setLogCallback:^(MSALLogLevel level, NSString *message, BOOL containsPII)
{
 if (!containsPII)
 {
#ifndef DEBUG
 // IMPORTANT: MSAL logs may contain sensitive information. Never output MSAL logs with NSLog, or
 print, directly unless you're running your application in debug mode. If you're writing MSAL logs to file,
 you must store the file securely.
 NSLog(@"MSAL log: %@", message);
#endif
 }
}];

```

Swift

```

MSALGlobalConfig.loggerConfig.setLogCallback { (level, message, containsPII) in
 if let message = message, !containsPII
 {
#ifndef DEBUG
 // IMPORTANT: MSAL logs may contain sensitive information. Never output MSAL logs with NSLog, or print,
 print("MSAL log: \(message)")
#endif
 }
}

```

## Personal data

By default, MSAL doesn't capture or log any personal data (PII). The library allows app developers to turn this on through a property in the `MSALLogger` class. By turning on `pii.Enabled`, the app takes responsibility for safely handling highly sensitive data and following regulatory requirements.

Objective-C

```

// By default, the `MSALLogger` doesn't capture any PII

// PII will be logged
MSALGlobalConfig.loggerConfig.piiEnabled = YES;

// PII will NOT be logged
MSALGlobalConfig.loggerConfig.piiEnabled = NO;

```

Swift

```
// By default, the `MSALLogger` doesn't capture any PII

// PII will be logged
MSALGlobalConfig.loggerConfig.piiEnabled = true

// PII will NOT be logged
MSALGlobalConfig.loggerConfig.piiEnabled = false
```

## Logging levels

To set the logging level when you log using MSAL for iOS and macOS, use one of the following values:

LEVEL	DESCRIPTION
MSALLogLevelNothing	Disable all logging
MSALLogLevelError	Default level, prints out information only when errors occur
MSALLogLevelWarning	Warnings
MSALLogLevelInfo	Library entry points, with parameters and various keychain operations
MSALLogLevelVerbose	API tracing

For example:

Objective-C

```
MSALGlobalConfig.loggerConfig.logLevel = MSALLogLevelVerbose;
```

Swift

```
MSALGlobalConfig.loggerConfig.logLevel = .verbose
```

## Log message format

The message portion of MSAL log messages is in the format of

```
TID = <thread_id> MSAL <sdk_ver> <OS> <OS_ver> [timestamp - correlation_id] message
```

For example:

```
TID = 551563 MSAL 0.2.0 iOS Sim 12.0 [2018-09-24 00:36:38 - 36764181-EF53-4E4E-B3E5-16FE362CFC44]
acquireToken returning with error: (MSALErrorDomain, -42400) User cancelled the authorization session.
```

Providing correlation IDs and timestamps are helpful for tracking down issues. Timestamp and correlation ID information is available in the log message. The only reliable place to retrieve them is from MSAL logging messages.

## Logging in MSAL for Java

MSAL for Java (MSAL4J) allows you to use the logging library that you are already using with your app, as long as it is compatible with SLF4J. MSAL4j uses the [Simple Logging Facade for Java](#) (SLF4J) as a simple facade or abstraction for various logging frameworks, such as [java.util.logging](#), [Logback](#) and [Log4j](#). SLF4J allows the end-user to plug in the desired logging framework at deployment time.

For example, to use Logback as the logging framework in your application, add the Logback dependency to the Maven pom file for your application:

```
<dependency>
 <groupId>ch.qos.logback</groupId>
 <artifactId>logback-classic</artifactId>
 <version>1.2.3</version>
</dependency>
```

Then add the Logback configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">

</configuration>
```

SLF4J automatically binds to Logback at deployment time. MSAL logs will be written to the console.

For instructions on how to bind to other logging frameworks, see the [SLF4J manual](#).

### Personal and organization information

By default, MSAL logging does not capture or log any personal or organizational data. In the following example, logging personal or organizational data is off by default:

```
PublicClientApplication app2 = PublicClientApplication.builder(PUBLIC_CLIENT_ID)
 .authority(AUTHORITY)
 .build();
```

Turn on personal and organizational data logging by setting `logPii()` on the client application builder. If you turn on personal or organizational data logging, your app must take responsibility for safely handling highly-sensitive data and complying with any regulatory requirements.

In the following example, logging personal or organizational data is enabled:

```
PublicClientApplication app2 = PublicClientApplication.builder(PUBLIC_CLIENT_ID)
 .authority(AUTHORITY)
 .logPii(true)
 .build();
```

# Single sign-on with MSAL.js

10/30/2019 • 5 minutes to read • [Edit Online](#)

Single Sign-On (SSO) enables users to enter their credentials once to sign in and establish a session which can be reused across multiple applications without requiring to authenticate again. This provides a seamless experience to the user and reduces the repeated prompts for credentials.

Azure AD provides SSO capabilities to applications by setting a session cookie when the user authenticates the first time. The MSAL.js library allows applications to leverage this in a few ways.

## SSO between browser tabs

When your application is open in multiple tabs and you first sign in the user on one tab, the user is also signed in on the other tabs without being prompted. MSAL.js caches the ID token for the user in the browser `localStorage` and will sign the user in to the application on the other open tabs.

By default, MSAL.js uses `sessionStorage` which does not allow the session to be shared between tabs. To get SSO between tabs, make sure to set the `cacheLocation` in MSAL.js to `localStorage` as shown below.

```
const config = {
 auth: {
 clientId: "abcd-ef12-gh34-ikkl-ashdjhhlhsdg"
 },
 cache: {
 cacheLocation: 'localStorage'
 }
}

const myMSALObj = new UserAgentApplication(config);
```

## SSO between apps

When a user authenticates, a session cookie is set on the Azure AD domain in the browser. MSAL.js relies on this session cookie to provide SSO for the user between different applications. MSAL.js also caches the ID tokens and access tokens of the user in the browser storage per application domain. As a result, the SSO behavior varies for different cases:

### Applications on the same domain

When applications are hosted on the same domain, the user can sign into an app once and then get authenticated to the other apps without a prompt. MSAL.js leverages the tokens cached for the user on the domain to provide SSO.

### Applications on different domain

When applications are hosted on different domains, the tokens cached on domain A cannot be accessed by MSAL.js in domain B.

This means that when users signed in on domain A navigate to an application on domain B, they will be redirected or prompted with the Azure AD page. Since Azure AD still has the user session cookie, it will sign in the user and they will not have to re-enter the credentials. If the user has multiple user accounts in session with Azure AD, the user will be prompted to pick the relevant account to sign in with.

### Automatically select account on Azure AD

In certain cases, the application has access to the user's authentication context and wants to avoid the Azure AD account selection prompt when multiple accounts are signed in. This can be done a few different ways:

## Using Session ID (SID)

Session ID is an [optional claim](#) that can be configured in the ID tokens. This claim allows the application to identify the user's Azure AD session independent of the user's account name or username. You can pass the SID in the request parameters to the `acquireTokenSilent` call. This will allow Azure AD to bypass the account selection. SID is bound to the session cookie and will not cross browser contexts.

```
var request = {
 scopes: ["user.read"],
 sid: sid
}

userAgentApplication.acquireTokenSilent(request).then(function(response) {
 const token = response.accessToken
})
.catch(function (error) {
 //handle error
});
```

### NOTE

SID can be used only with silent authentication requests made by `acquireTokenSilent` call in MSAL.js. You can find the steps to configure optional claims in your application manifest [here](#).

## Using Login Hint

If you do not have SID claim configured or need to bypass the account selection prompt in interactive authentication calls, you can do so by providing a `login_hint` in the request parameters and optionally a `domain_hint` as `extraQueryParameters` in the MSAL.js interactive methods (`loginPopup`, `loginRedirect`, `acquireTokenPopup` and `acquireTokenRedirect`). For example:

```
var request = {
 scopes: ["user.read"],
 loginHint: preferred_username,
 extraQueryParameters: {domain_hint: 'organizations'}
}

userAgentApplication.loginRedirect(request);
```

You can get the values for `login_hint` and `domain_hint` by reading the claims returned in the ID token for the user.

- **loginHint** should be set to the `preferred_username` claim in the ID token.
- **domain\_hint** is only required to be passed when using the /common authority. The domain hint is determined by tenant ID(tid). If the `tid` claim in the ID token is `9188040d-6c67-4c5b-b112-36a304b66dad` it is consumers. Otherwise, it is organizations.

Read [here](#) for more information on the values for login hint and domain hint.

### NOTE

You cannot pass SID and `login_hint` at the same time. This will result in error response.

## SSO without MSAL.js login

By design, MSAL.js requires that a login method is called to establish a user context before getting tokens for APIs. Since login methods are interactive, the user sees a prompt.

There are certain cases in which applications have access to the authenticated user's context or ID token through authentication initiated in another application and want to leverage SSO to acquire tokens without first signing in through MSAL.js.

An example of this is: A user is signed into a parent web application which hosts another JavaScript application running as an add-on or plugin.

The SSO experience in this scenario can be achieved as follows:

Pass the `sid` if available (or `login_hint` and optionally `domain_hint`) as request parameters to the MSAL.js `acquireTokenSilent` call as follows:

```
var request = {
 scopes: ["user.read"],
 loginHint: preferred_username,
 extraQueryParameters: {domain_hint: 'organizations'}
}

userAgentApplication.acquireTokenSilent(request).then(function(response) {
 const token = response.accessToken
})
.catch(function (error) {
 //handle error
});
```

## SSO in ADAL.js to MSAL.js update

MSAL.js brings feature parity with ADAL.js for Azure AD authentication scenarios. To make the migration from ADAL.js to MSAL.js easy and to avoid prompting your users to sign in again, the library reads the ID token representing user's session in ADAL.js cache, and seamlessly signs in the user in MSAL.js.

To take advantage of the single sign-on (SSO) behavior when updating from ADAL.js, you will need to ensure the libraries are using `localStorage` for caching tokens. Set the `cacheLocation` to `localStorage` in both the MSAL.js and ADAL.js configuration at initialization as follows:

```
// In ADAL.js
window.config = {
 clientId: 'g075edef-0efa-453b-997b-de1337c29185',
 cacheLocation: 'localStorage'
};

var authContext = new AuthenticationContext(config);

// In latest MSAL.js version
const config = {
 auth: {
 clientId: "abcd-ef12-gh34-ikkl-ashdjhhlhsdg"
 },
 cache: {
 cacheLocation: 'localStorage'
 }
};

const myMSALObj = new UserAgentApplication(config);
```

Once this is configured, MSAL.js will be able to read the cached state of the authenticated user in ADAL.js and use that to provide SSO in MSAL.js.

## Next steps

Learn more about the [single sign-on session and token lifetime](#) values in Azure AD.

# How to: Configure SSO on macOS and iOS

10/30/2019 • 5 minutes to read • [Edit Online](#)

The Microsoft Authentication Library (MSAL) for macOS and iOS supports Single Sign-on (SSO) between macOS/iOS apps and browsers. This article covers the following SSO scenarios:

- [Silent SSO between multiple apps](#)

This type of SSO works between multiple apps distributed by the same Apple Developer. It provides silent SSO (that is, the user isn't prompted for credentials) by reading refresh tokens written by other apps from the keychain, and exchanging them for access tokens silently.

- [SSO through Authentication broker](#)

## IMPORTANT

This flow is not available on macOS.

Microsoft provides apps, called brokers, that enable SSO between applications from different vendors as long as the mobile device is registered with Azure Active Directory (AAD). This type of SSO requires a broker application be installed on the user's device.

- [SSO between MSAL and Safari](#)

SSO is achieved through the [ASWebAuthenticationSession](#) class. It uses existing sign-in state from other apps and the Safari browser. It's not limited to apps distributed by the same Apple Developer, but it requires some user interaction.

If you use the default web view in your app to sign in users, you'll get automatic SSO between MSAL-based applications and Safari. To learn more about the web views that MSAL supports, visit [Customize browsers and WebViews](#).

## IMPORTANT

This type of SSO is currently not available on macOS. MSAL on macOS only supports WKWebView which doesn't have SSO support with Safari.

- [Silent SSO between ADAL and MSAL macOS/iOS apps](#)

MSAL Objective-C supports migration and SSO with ADAL Objective-C-based apps. The apps must be distributed by the same Apple Developer.

See [SSO between ADAL and MSAL apps on macOS and iOS](#) for instructions for cross-app SSO between ADAL and MSAL-based apps.

## Silent SSO between apps

MSAL supports SSO sharing through iOS keychain access groups.

To enable SSO across your applications, you'll need to do the following steps, which are explained in more detail below:

1. Ensure that all your applications use the same Client ID or Application ID.
2. Ensure that all of your applications share the same signing certificate from Apple so that you can share keychains.
3. Request the same keychain entitlement for each of your applications.
4. Tell the MSAL SDKs about the shared keychain you want us to use if it's different from the default one.

### Use the same Client ID and Application ID

For the Microsoft identity platform to know which applications can share tokens, those applications need to share the same Client ID or Application ID. This is the unique identifier that was provided to you when you registered your first application in the portal.

The way the Microsoft identity platform tells apps that use the same Application ID apart is by their **Redirect URIs**. Each application can have multiple Redirect URIs registered in the onboarding portal. Each app in your suite will have a different redirect URI. For example:

App1 Redirect URI: `msauth.com.contoso.mytestapp1://auth`

`msauth.com.contoso.mytestapp2://auth`

App3 Redirect URI: `msauth.com.contoso.mytestapp3://auth`

#### IMPORTANT

The format of redirect uris must be compatible with the format MSAL supports, which is documented in [MSAL Redirect URI format requirements](#).

### Setup keychain sharing between applications

Refer to Apple's [Adding Capabilities](#) article to enable keychain sharing. What is important is that you decide what you want your keychain to be called and add that capability to all of your applications that will be involved in SSO.

When you have the entitlements set up correctly, you'll see a `entitlements.plist` file in your project directory that contains something like this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>keychain-access-groups</key>
 <array>
 <string>$(AppIdentifierPrefix)com.myapp.mytestapp</string>
 <string>$(AppIdentifierPrefix)com.myapp.mycache</string>
 </array>
</dict>
</plist>
```

Once you have the keychain entitlement enabled in each of your applications, and you're ready to use SSO, configure `MSALPublicClientApplication` with your keychain access group as in the following example:

Objective-C:

```
NSError *error = nil;
MSALPublicClientApplicationConfig *configuration = [[MSALPublicClientApplicationConfig alloc]
initWithClientId:@"<my-client-id>"];
configuration.cacheConfig.keychainSharingGroup = @"my.keychain.group";

MSALPublicClientApplication *application = [[MSALPublicClientApplication alloc]
initWithConfiguration:configuration error:&error];
```

Swift:

```

let config = MSALPublicClientApplicationConfig(clientId: "<my-client-id>")
config.cacheConfig.keychainSharingGroup = "my.keychain.group"

do {
 let application = try MSALPublicClientApplication(configuration: config)
 // continue on with application
} catch let error as NSError {
 // handle error here
}

```

### WARNING

When you share a keychain across your applications, any application can delete users or even all of the tokens across your application. This is particularly impactful if you have applications that rely on tokens to do background work. Sharing a keychain means that you must be very careful when your app uses Microsoft identity SDK remove operations.

That's it! The Microsoft identity SDK will now share credentials across all your applications. The account list will also be shared across application instances.

## SSO through Authentication broker on iOS

MSAL provides support for brokered authentication with Microsoft Authenticator. Microsoft Authenticator provides SSO for AAD registered devices, and also helps your application follow Conditional Access policies.

The following steps are how you enable SSO using an authentication broker for your app:

1. Register a broker compatible Redirect URI format for the application in your app's Info.plist. The broker compatible Redirect URI format is `msauth.<app.bundle.id>://auth`. Replace `<app.bundle.id>` with your application's bundle ID. For example:

```

<key>CFBundleURLSchemes</key>
<array>
 <string>msauth.<app.bundle.id></string>
</array>

```

2. Add following schemes to your app's Info.plist under `LSApplicationQueriesSchemes`:

```

<key>LSApplicationQueriesSchemes</key>
<array>
 <string>msauthv2</string>
 <string>msauthv3</string>
</array>

```

3. Add the following to your `AppDelegate.m` file to handle callbacks:

Objective-C:

```

- (BOOL)application:(UIApplication *)app openURL:(NSURL *)url options:(NSDictionary<NSString *,id> *)options
{
 return [MSALPublicClientApplication handleMSALResponse:url
sourceApplication:options[UIApplicationOpenURLOptionsSourceApplicationKey]];
}

```

Swift:

```
func application(_ app: UIApplication, open url: URL, options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
 return MSALPublicClientApplication.handleMSALResponse(url, sourceApplication: options[UIApplication.OpenURLOptionsKey.sourceApplication] as? String)
}
```

If you are using Xcode 11, you should place MSAL callback into the `SceneDelegate` file instead. If you support both UISceneDelegate and UIApplicationDelegate for compatibility with older iOS, MSAL callback would need to be placed into both files.

Objective-C:

```
- (void)scene:(UIScene *)scene openURLContexts:(NSSet<UIOpenURLContext *> *)URLContexts {
 UIOpenURLContext *context = URLContexts.anyObject;
 NSURL *url = context.URL;
 NSString *sourceApplication = context.options.sourceApplication;

 [MSALPublicClientApplication handleMSALResponse:url sourceApplication:sourceApplication];
}
```

Swift:

```
func scene(_ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>) {
 guard let urlContext = URLContexts.first else {
 return
 }

 let url = urlContext.url
 let sourceApp = urlContext.options.sourceApplication

 MSALPublicClientApplication.handleMSALResponse(url, sourceApplication: sourceApp)
}
```

## Next steps

Learn more about [Authentication flows and application scenarios](#)

# How to: SSO between ADAL and MSAL apps on macOS and iOS

10/10/2019 • 6 minutes to read • [Edit Online](#)

The Microsoft Authentication Library (MSAL) for iOS can share SSO state with [ADAL Objective-C](#) between applications. You can migrate your apps to MSAL at your own pace, ensuring that your users will still benefit from cross-app SSO--even with a mix of ADAL and MSAL-based apps.

If you're looking for guidance of setting up SSO between apps using the MSAL SDK, see [Silent SSO between multiple apps](#). This article focuses on SSO between ADAL and MSAL.

The specifics implementing SSO depend on ADAL version that you're using.

## ADAL 2.7.x

This section covers SSO differences between MSAL and ADAL 2.7.x

### Cache format

ADAL 2.7.x can read the MSAL cache format. You don't need to do anything special for cross-app SSO with version ADAL 2.7.x. However, you need to be aware of differences in account identifiers that those two libraries support.

### Account identifier differences

MSAL and ADAL use different account identifiers. ADAL uses UPN as its primary account identifier. MSAL uses a non-displayable account identifier that is based of an object ID and a tenant ID for AAD accounts, and a `sub` claim for other types of accounts.

When you receive an `MSALAccount` object in the MSAL result, it contains an account identifier in the `identifier` property. The application should use this identifier for subsequent silent requests.

In addition to `identifier`, `MSALAccount` object contains a displayable identifier called `username`. That translates to `userId` in ADAL. `username` is not considered a unique identifier and can change anytime, so it should only be used for backward compatibility scenarios with ADAL. MSAL supports cache queries using either `username` or `identifier`, where querying by `identifier` is recommended.

Following table summarizes account identifier differences between ADAL and MSAL:

ACCOUNT IDENTIFIER	MSAL	ADAL 2.7.X	OLDER ADAL (BEFORE ADAL 2.7.X)
displayable identifier	<code>username</code>	<code>userId</code>	<code>userId</code>
unique non-displayable identifier	<code>identifier</code>	<code>homeAccountId</code>	N/A
No account id known	Query all accounts through <code>allAccounts:</code> API in <code>MSALPublicClientApplication</code>	N/A	N/A

This is the `MSALAccount` interface providing those identifiers:

```

@protocol MSALAccount <NSObject>

/*
Displayable user identifier. Can be used for UI and backward compatibility with ADAL.
*/
@property (readonly, nullable) NSString *username;

/*
Unique identifier for the account.
Save this for account lookups from cache at a later point.
*/
@property (readonly, nullable) NSString *identifier;

/*
Host part of the authority string used for authentication based on the issuer identifier.
*/
@property (readonly, nonnull) NSString *environment;

/*
ID token claims for the account.
Can be used to read additional information about the account, e.g. name
Will only be returned if there has been an id token issued for the client Id for the account's source tenant.
*/
@property (readonly, nullable) NSDictionary<NSString *, NSString *> *accountClaims;

@end

```

## SSO from MSAL to ADAL

If you have an MSAL app and an ADAL app, and the user first signs into the MSAL-based app, you can get SSO in the ADAL app by saving the `username` from the `MSALAccount` object and passing it to your ADAL-based app as `userId`. ADAL can then find the account information silently with the `acquireTokenSilentWithResource:clientId:redirectUri:userId:completionBlock:` API.

## SSO from ADAL to MSAL

If you have an MSAL app and an ADAL app, and the user first signs into the ADAL-based app, you can use ADAL user identifiers for account lookups in MSAL. This also applies when migrating from ADAL to MSAL.

### ADAL's `homeAccountId`

ADAL 2.7.x returns the `homeAccountId` in the `ADUserInformation` object in the result via this property:

```

/*! Unique AAD account identifier across tenants based on user's home OID/home tenantId. */
@property (readonly) NSString *homeAccountId;

```

`homeAccountId` in ADAL's is equivalent of `identifier` in MSAL. You can save this identifier to use in MSAL for account lookups with the `accountForIdentifier:error:` API.

### ADAL's `userId`

If `homeAccountId` is not available, or you only have the displayable identifier, you can use ADAL's `userId` to lookup the account in MSAL.

In MSAL, first look up an account by `username` or `identifier`. Always use `identifier` for querying if you have it, and only use `username` as a fallback. If the account is found, use the account in the `acquireTokenSilent` calls.

Objective-C:

```

NSString *msalIdentifier = @"previously.saved.msal.account.id";
MSALAccount *account = nil;

if (msalIdentifier)
{
 // If you have MSAL account id returned either from MSAL as identifier or ADAL as homeAccountId, use it
 account = [application accountForIdentifier:@"my.account.id.here" error:nil];
}
else
{
 // Fallback to ADAL userId for migration
 account = [application accountForUsername:@"adal.user.id" error:nil];
}

if (!account)
{
 // Account not found.
 return;
}

MSALSilentTokenParameters *silentParameters = [[MSALSilentTokenParameters alloc]
initWithScopes:@[@"user.read"] account:account];
[application acquireTokenSilentWithParameters:silentParameters completionBlock:completionBlock];

```

Swift:

```

let msalIdentifier: String?
var account: MSALAccount

do {
 if let msalIdentifier = msalIdentifier {
 account = try application.account(forIdentifier: msalIdentifier)
 }
 else {
 account = try application.account(forUsername: "adal.user.id")
 }

 let silentParameters = MSALSilentTokenParameters(scopes: ["user.read"], account: account)
 application.acquireTokenSilent(with: silentParameters) {
 (result: MSALResult?, error: Error?) in
 // handle result
 }
} catch let error as NSError {
 // handle error or account not found
}

```

MSAL supported account lookup APIs:

```

/*
 Returns account for the given account identifier (received from an account object returned in a previous
 acquireToken call)

@param error The error that occurred trying to get the accounts, if any, if you're
 not interested in the specific error pass in nil.
*/
- (nullable MSALAccount *)accountForIdentifier:(nonnull NSString *)identifier
 error:(NSError * _Nullable __autoreleasing * _Nullable)error;

/*
 Returns account for the given username (received from an account object returned in a previous
 acquireToken call or ADAL)

@param username The displayable value in UserPrincipleName(UPN) format
@param error The error that occurred trying to get the accounts, if any, if you're
 not interested in the specific error pass in nil.
*/
- (MSALAccount *)accountForUsername:(NSString *)username
 error:(NSError * _autoreleasing *)error;

```

## ADAL 2.x-2.6.6

This section covers SSO differences between MSAL and ADAL 2.x-2.6.6.

Older ADAL versions don't natively support the MSAL cache format. However, to ensure smooth migration from ADAL to MSAL, MSAL can read the older ADAL cache format without prompting for user credentials again.

Because `homeAccountId` isn't available in older ADAL versions, you'd need to look up accounts using the `username`:

```

/*
 Returns account for the given username (received from an account object returned in a previous
 acquireToken call or ADAL)

@param username The displayable value in UserPrincipleName(UPN) format
@param error The error that occurred trying to get the accounts, if any. If you're not interested in
 the specific error pass in nil.
*/
- (MSALAccount *)accountForUsername:(NSString *)username
 error:(NSError * _autoreleasing *)error;

```

For example:

Objective-C:

```

MSALAccount *account = [application accountForUsername:@"adal.user.id" error:nil];
MSALSilentTokenParameters *silentParameters = [[MSALSilentTokenParameters alloc]
initWithScopes:@[@"user.read"] account:account];
[application acquireTokenSilentWithParameters:silentParameters completionBlock:completionBlock];

```

Swift:

```

do {
 let account = try application.account(forUsername: "adal.user.id")
 let silentParameters = MSALSilentTokenParameters(scopes: ["user.read"], account: account)
 application.acquireTokenSilent(with: silentParameters) {
 (result: MSALResult?, error: Error?) in
 // handle result
 }
} catch let error as NSError {
 // handle error or account not found
}

```

Alternatively, you can read all of the accounts, which will also read account information from ADAL:

Objective-C:

```

NSArray *accounts = [application allAccounts:nil];

if ([accounts count] == 0)
{
 // No account found.
 return;
}
if ([accounts count] > 1)
{
 // You might want to display an account picker to user in actual application
 // For this sample we assume there's only ever one account in cache
 return;
}
```
MSALSilentTokenParameters *silentParameters = [[MSALSilentTokenParameters alloc]
initWithScopes:@[@"user.read"] account:accounts[0]];
[application acquireTokenSilentWithParameters:silentParameters completionBlock:completionBlock];

```

Swift:

```

do {
    let accounts = try application.allAccounts()
    if accounts.count == 0 {
        // No account found.
        return
    }
    if accounts.count > 1 {
        // You might want to display an account picker to user in actual application
        // For this sample we assume there's only ever one account in cache
        return
    }

    let silentParameters = MSALSilentTokenParameters(scopes: ["user.read"], account: accounts[0])
    application.acquireTokenSilent(with: silentParameters) {
        (result: MSALResult?, error: Error?) in
        // handle result or error
    }
} catch let error as NSError {
    // handle error
}

```

Next steps

Learn more about [Authentication flows and application scenarios](#)

Active Directory Federation Services support in MSAL.NET

11/8/2019 • 2 minutes to read • [Edit Online](#)

Active Directory Federation Services (AD FS) in Windows Server enables you to add OpenID Connect and OAuth 2.0 based authentication and authorization to applications you are developing. Those applications can, then, authenticate users directly against AD FS. For more information, read [AD FS Scenarios for Developers](#).

Microsoft Authentication Library for .NET (MSAL.NET) supports two scenarios for authenticating against AD FS:

- MSAL.NET talks to Azure Active Directory, which itself is *federated* with AD FS.
- MSAL.NET talks **directly** to an ADFS authority. This is only supported from AD FS 2019 and above. One of the scenarios this highlights is [Azure Stack](#) support

MSAL connects to Azure AD, which is federated with AD FS

MSAL.NET supports connecting to Azure AD, which signs in managed-users (users managed in Azure AD) or federated users (users managed by another identity provider such as AD FS). MSAL.NET does not know about the fact that users are federated. As far as it's concerned, it talks to Azure AD.

The [authority](#) you use in this case is the usual authority (authority host name + tenant, common, or organizations).

Acquiring a token interactively

When you call the `AcquireTokenInteractive` method, the user experience is typically:

1. The user enters their account ID.
2. Azure AD displays briefly the message "Taking you to your organization's page".
3. The user is redirected to the sign-in page of the identity provider. The sign-in page is usually customized with the logo of the organization.

Supported AD FS versions in this federated scenario are AD FS v2, AD FS v3 (Windows Server 2012 R2), and AD FS v4 (AD FS 2016).

Acquiring a token using `AcquireTokenByIntegratedAuthentication` or `AcquireTokenByUsernamePassword`

When acquiring a token using the `AcquireTokenByIntegratedAuthentication` Or `AcquireTokenByUsernamePassword` methods, MSAL.NET gets the identity provider to contact based on the username. MSAL.NET receives a [SAML 1.1 token](#) after contacting the identity provider. MSAL.NET then provides the SAML token to Azure AD as a user assertion (similar to the [on-behalf-of flow](#)) to get back a JWT.

MSAL connects directly to AD FS

MSAL.NET supports connecting to AD FS 2019, which is Open ID Connect compliant and understands PKCE and scopes. This support requires that a service pack [KB 4490481](#) is applied to Windows Server. When connecting directly to AD FS, the authority you'll want to use to build your application is similar to

`https://mysite.contoso.com/adfs/`.

Currently, there are no plans to support a direct connection to:

- AD FS 16, as it doesn't support PKCE and still uses resources, not scope
- AD FS v2, which is not OIDC-compliant.

If you need to support scenarios requiring a direct connection to AD FS 2016, use the latest version of [Azure Active Directory Authentication Library](#). When you have upgraded your on-premise system to AD FS 2019, you'll be able to use MSAL.NET.

See also

For the federated case, see [Configure Azure Active Directory sign in behavior for an application by using a Home Realm Discovery policy](#)

Use MSAL for Android with B2C

10/23/2019 • 4 minutes to read • [Edit Online](#)

Microsoft Authentication Library (MSAL) enables application developers to authenticate users with social and local identities by using [Azure Active Directory B2C \(Azure AD B2C\)](#). Azure AD B2C is an identity management service. Use it to customize and control how customers sign up, sign in, and manage their profiles when they use your applications.

Configure known authorities and redirect URI

In MSAL for Android, B2C policies (user journeys) are configured as individual authorities.

Given a B2C application that has two policies:

- Sign-up / Sign-in
 - Called `B2C_1_SISOPolicy`
- Edit Profile
 - Called `B2C_1_EditProfile`

The configuration file for the app would declare two `authorities`. One for each policy. The `type` property of each authority is `B2C`.

`app/src/main/res/raw/msal_config.json`

```
{  
  "client_id": "<your_client_id_here>",  
  "redirect_uri": "<your_redirect_uri_here>",  
  "authorities": [{  
    "type": "B2C",  
    "authority_url": "https://contoso.b2clogin.com/tfp/contoso.onmicrosoft.com/B2C_1_SISOPolicy/",  
    "default": true  
  },  
  {  
    "type": "B2C",  
    "authority_url": "https://contoso.b2clogin.com/tfp/contoso.onmicrosoft.com/B2C_1_EditProfile/"  
  }]  
}
```

The `redirect_uri` must be registered in the app configuration, and also in `AndroidManifest.xml` to support redirection during the [authorization code grant flow](#).

Initialize IPublicClientApplication

`IPublicClientApplication` is constructed by a factory method to allow the application configuration to be parsed asynchronously.

```

PublicClientApplication.createMultipleAccountPublicClientApplication(
    context, // Your application Context
    R.raw.msal_config, // Id of app JSON config
    new IPublicClientApplication.ApplicationCreatedListener() {
        @Override
        public void onCreated(IMultipleAccountPublicClientApplication pca) {
            // Application has been initialized.
        }

        @Override
        public void onError(MsalException exception) {
            // Application could not be created.
            // Check Exception message for details.
        }
    }
);

```

Interactively acquire a token

To acquire a token interactively with MSAL, build an `AcquireTokenParameters` instance and supply it to the `acquireToken` method. The token request below uses the `default` authority.

```

IMultipleAccountPublicClientApplication pca = ...; // Initialization not shown

AcquireTokenParameters parameters = new AcquireTokenParameters.Builder()
    .startAuthorizationFromActivity(activity)
    .withScopes(Arrays.asList("https://contoso.onmicrosoft.com/contosob2c/read")) // Provide your registered
scope here
    .withPrompt(Prompt.LOGIN)
    .callback(new AuthenticationCallback() {
        @Override
        public void onSuccess(IAuthenticationResult authenticationResult) {
            // Token request was successful, inspect the result
        }

        @Override
        public void onError(MsalException exception) {
            // Token request was unsuccessful, inspect the exception
        }

        @Override
        public void onCancel() {
            // The user cancelled the flow
        }
    }).build();

pca.acquireToken(parameters);

```

Silently renew a token

To acquire a token silently with MSAL, build an `AcquireTokenSilentParameters` instance and supply it to the `acquireTokenSilentAsync` method. Unlike the `acquireToken` method, the `authority` must be specified to acquire a token silently.

```

IMultilpeAccountPublicClientApplication pca = ...; // Initialization not shown
AcquireTokenSilentParameters parameters = new AcquireTokenSilentParameters.Builder()
    .withScopes(Arrays.asList("https://contoso.onmicrosoft.com/contosob2c/read")) // Provide your registered
scope here
    .forAccount(account)
    // Select a configured authority (policy), mandatory for silent auth requests
    .fromAuthority("https://contoso.b2clogin.com/tfp/contoso.onmicrosoft.com/B2C_1_SISOPolicy/")
    .callback(new SilentAuthenticationCallback() {
        @Override
        public void onSuccess(IAuthenticationResult authenticationResult) {
            // Token request was successful, inspect the result
        }

        @Override
        public void onError(MsalException exception) {
            // Token request was unsuccesful, inspect the exception
        }
    })
    .build();

pca.acquireTokenSilentAsync(parameters);

```

Specify a policy

Because policies in B2C are represented as separate authorities, invoking a policy other than the default is achieved by specifying a `fromAuthority` clause when constructing `acquireToken` or `acquireTokenSilent` parameters. For example:

```

AcquireTokenParameters parameters = new AcquireTokenParameters.Builder()
    .startAuthorizationFromActivity(activity)
    .withScopes(Arrays.asList("https://contoso.onmicrosoft.com/contosob2c/read")) // Provide your registered
scope here
    .withPrompt(Prompt.LOGIN)
    .callback(...) // provide callback here
    .fromAuthority("<url_of_policy_defined_in_configuration_json>")
    .build();

```

Handle password change policies

The local account sign-up or sign-in user flow shows a '**Forgot password?**' link. Clicking this link doesn't automatically trigger a password reset user flow.

Instead, the error code `AADB2C90118` is returned to your app. Your app should handle this error code by running a specific user flow that resets the password.

To catch a password reset error code, the following implementation can be used inside your

`AuthenticationCallback`:

```

new AuthenticationCallback() {

    @Override
    public void onSuccess(IAuthenticationResult authenticationResult) {
        // ..
    }

    @Override
    public void onError(MsalException exception) {
        final String B2C_PASSWORD_CHANGE = "AADB2C90118";

        if (exception.getMessage().contains(B2C_PASSWORD_CHANGE)) {
            // invoke password reset flow
        }
    }

    @Override
    public void onCancel() {
        // ..
    }
}

```

Use IAuthenticationResult

A successful token acquisition results in a `IAuthenticationResult` object. It contains the access token, user claims, and metadata.

Get the access token and related properties

```

// Get the raw bearer token
String accessToken = authenticationResult.getAccessToken();

// Get the scopes included in the access token
String[] accessTokenScopes = authenticationResult.getScope();

// Gets the access token's expiry
Date expiry = authenticationResult.getExpiresOn();

// Get the tenant for which this access token was issued
String tenantId = authenticationResult.getTenantId();

```

Get the authorized account

```

// Get the account from the result
IAccount account = authenticationResult.getAccount();

// Get the id of this account - note for B2C, the policy name is a part of the id
String id = account.getId();

// Get the IdToken Claims
//
// For more information about B2C token claims, see reference documentation
// https://docs.microsoft.com/azure/active-directory-b2c/active-directory-b2c-reference-tokens
Map<String, ?> claims = account.getClaims();

// Get the 'preferred_username' claim through a convenience function
String username = account.getUsername();

// Get the tenant id (tid) claim through a convenience function
String tenantId = account.getTenantId();

```

IdToken claims

Claims returned in the IdToken are populated by the Security Token Service (STS), not by MSAL. Depending on the identity provider (IdP) used, some claims may be absent. Some IdPs don't currently provide the `preferred_username` claim. Because this claim is used by MSAL for caching, a placeholder value, `MISSING FROM THE TOKEN RESPONSE`, is used in its place. For more information on B2C IdToken claims, see [Overview of tokens in Azure Active Directory B2C](#).

Managing accounts and policies

B2C treats each policy as a separate authority. Thus the access tokens, refresh tokens, and ID tokens returned from each policy are not interchangeable. This means each policy returns a separate `IAccount` object whose tokens can't be used to invoke other policies.

Each policy adds an `IAccount` to the cache for each user. If a user signs in to an application and invokes two policies, they'll have two `IAccount`s. To remove this user from the cache, you must call `removeAccount()` for each policy.

When you renew tokens for a policy with `acquireTokenSilent`, provide the same `IAccount` that was returned from previous invocations of the policy to `AcquireTokenSilentParameters`. Providing an account returned by another policy will result in an error.

Next steps

Learn more about Azure Active Directory B2C (Azure AD B2C) at [What is Azure Active Directory B2C?](#)

Use Microsoft Authentication Library to interoperate with Azure Active Directory B2C

9/18/2019 • 2 minutes to read • [Edit Online](#)

Microsoft Authentication Library (MSAL) enables application developers to authenticate users with social and local identities by using [Azure Active Directory B2C \(Azure AD B2C\)](#). Azure AD B2C is an identity management service. By using it, you can customize and control how customers sign up, sign in, and manage their profiles when they use your applications.

Azure AD B2C also enables you to brand and customize the UI of your applications to provide a seamless experience to your customers.

This tutorial demonstrates how to use MSAL to interoperate with Azure AD B2C.

Prerequisites

If you haven't already created your own [Azure AD B2C tenant](#), create one now. You also can use an existing Azure AD B2C tenant.

JavaScript

The following steps demonstrate how a single-page application can use Azure AD B2C to sign up, sign in, and call a protected web API.

Step 1: Register your application

To implement authentication, you first need to register your application. See [Register your application](#) for detailed steps.

Step 2: Download the sample application

Download the sample as a zip file, or clone it from GitHub:

```
git clone https://github.com/Azure-Samples/active-directory-b2c-javascript-msal-singlepageapp.git
```

Step 3: Configure authentication

1. Open the **index.html** file in the sample.
2. Configure the sample with the client ID and key that you recorded earlier while registering your application.
Change the following lines of code by replacing the values with the names of your directory and APIs:

```

// The current application coordinates were pre-registered in a B2C tenant.

var appConfig = {
    b2cScopes: ["https://fabrikamb2c.onmicrosoft.com/helloapi/demo.read"],
    webApi: "https://fabrikamb2chello.azurewebsites.net/hello"
};

const msalConfig = {
    auth: {
        clientId: "e760cab2-b9a1-4c0d-86fb-ff7084abd902" //This is your client/application ID
        authority: "https://fabrikamb2c.b2clogin.com/fabrikamb2c.onmicrosoft.com/b2c_1_susi", //This is
        your tenant info
        validateAuthority: false
    },
    cache: {
        cacheLocation: "localStorage",
        storeAuthStateInCookie: true
    }
};
// create UserAgentApplication instance
const myMSALObj = new Msal.UserAgentApplication(msalConfig);

```

The name of the [user flow](#) in this tutorial is **B2C_1_signupsignin1**. If you're using a different user flow name, set the **authority** value to that name.

Step 4: Configure your application to use [b2clogin.com](#)

You can use [b2clogin.com](#) in place of [login.microsoftonline.com](#) as a redirect URL. You do so in your Azure AD B2C application when you set up an identity provider for sign-up and sign-in.

Use of [b2clogin.com](#) in the context of <https://your-tenant-name.b2clogin.com/your-tenant-guid> has the following effects:

- Microsoft services consume less space in the cookie header.
- Your URLs no longer include a reference to Microsoft. For example, your Azure AD B2C application probably refers to [login.microsoftonline.com](#).

To use [b2clogin.com](#), you need to update the configuration of your application.

- Set the **validateAuthority** property to [false](#), so that redirects using [b2clogin.com](#) can occur.

The following example shows how you might set the property:

```

// The current application coordinates were pre-registered in a B2C directory.

const msalConfig = {
    auth:{ 
        clientId: "Enter_the_Application_Id_here",
        authority: "https://contoso.b2clogin.com/tfp/contoso.onmicrosoft.com/B2C_1_signupsignin1",
        b2cScopes: ["https://contoso.onmicrosoft.com/demoapi/demo.read"],
        webApi: 'https://contosohello.azurewebsites.net/hello',
        validateAuthority: false;
    };
// create UserAgentApplication instance
const myMSALObj = new UserAgentApplication(msalConfig);

```

NOTE

Your Azure AD B2C application probably refers to `login.microsoftonline.com` in several places, such as your user flow references and token endpoints. Make sure that your authorization endpoint, token endpoint, and issuer have been updated to use `your-tenant-name.b2clogin.com`.

Follow [this MSAL JavaScript sample](#) on how to use the MSAL Preview for JavaScript (MSAL.js). The sample gets an access token and calls an API secured by Azure AD B2C.

Next steps

Learn more about:

- [Custom policies](#)
- [User interface customization](#)

Use MSAL.NET to sign in users with social identities

10/30/2019 • 6 minutes to read • [Edit Online](#)

You can use MSAL.NET to sign in users with social identities by using [Azure Active Directory B2C \(Azure AD B2C\)](#).

Azure AD B2C is built around the notion of policies. In MSAL.NET, specifying a policy translates to providing an authority.

- When you instantiate the public client application, you need to specify the policy in authority.
- When you want to apply a policy, you need to call an override of `AcquireTokenInteractive` containing an `authority` parameter.

This page is for MSAL 3.x. If you are interested in MSAL 2.x, please see [Azure AD B2C specifics in MSAL 2.x](#).

Authority for a Azure AD B2C tenant and policy

The authority to use is `https://{azureADB2CHostname}/tfp/{tenant}/{policyName}` where:

- `azureADB2CHostname` is the name of the Azure AD B2C tenant plus the host (for example `{your-tenant-name}.b2clogin.com`),
- `tenant` is the full name of the Azure AD B2C tenant (for example, `{your-tenant-name}.onmicrosoft.com`) or the GUID for the tenant,
- `policyName` the name of the policy or user flow to apply (for instance "b2c_1_susi" for sign-up/sign-in).

For more information on the Azure AD B2C authorities, see this [documentation](#).

Instantiating the application

When building the application, you need to provide the authority.

```
// Azure AD B2C Coordinates
public static string Tenant = "fabrikamb2c.onmicrosoft.com";
public static string AzureADB2CHostname = "fabrikamb2c.b2clogin.com";
public static string ClientID = "90c0fe63-bcf2-44d5-8fb7-b8bbc0b29dc6";
public static string PolicySignUpSignIn = "b2c_1_susi";
public static string PolicyEditProfile = "b2c_1_edit_profile";
public static string PolicyResetPassword = "b2c_1_reset";

public static string AuthorityBase = $"https://{AzureADB2CHostname}/tfp/{Tenant}/";
public static string Authority = $"{AuthorityBase}{PolicySignUpSignIn}";
public static string AuthorityEditProfile = $"{AuthorityBase}{PolicyEditProfile}";
public static string AuthorityPasswordReset = $"{AuthorityBase}{PolicyResetPassword}";

application = PublicClientApplicationBuilder.Create(ClientID)
    .WithB2CAuthority(Authority)
    .Build();
```

Acquire a token to apply a policy

Acquiring a token for an Azure AD B2C protected API in a public client application requires you to use the overrides with an authority:

```

IEnumerable<IAccount> accounts = await application.GetAccountsAsync();
AuthenticationResult ar = await application.AcquireTokenInteractive(scopes)
    .WithAccount(GetAccountByPolicy(accounts, policy))
    .WithParentActivityOrWindow(ParentActivityOrWindow)
    .ExecuteAsync();

```

with:

- `policy` being one of the previous strings (for instance `PolicySignUpSignIn`).
- `ParentActivityOrWindow` is required for Android (the Activity), and optional for other platforms which support the parent UI, such as windows in Windows and UIViewController in iOS. See more information [here on the UI dialog](#).
- `GetAccountByPolicy(IEnumerable<IAccount>, string)` is a method that finds an account for a given policy. For example:

```

private IAccount GetAccountByPolicy(IEnumerable<IAccount> accounts, string policy)
{
    foreach (var account in accounts)
    {
        string userIdentifier = account.HomeAccountId.ObjectId.Split('.')[0];
        if (userIdentifier.EndsWith(policy.ToLower()))
            return account;
    }
    return null;
}

```

Applying a policy or user flow (for example, letting the end user edit their profile or reset their password) is currently done by calling `AcquireTokenInteractive`. In the case of these two policies, you don't use the returned token / authentication result.

Special case of EditProfile and ResetPassword policies

When you want to provide an experience where your end users sign in with a social identity and then edit their profile, you want to apply the Azure AD B2C Edit Profile policy. The way to do this is by calling

`AcquireTokenInteractive` with the specific authority for that policy, and a Prompt set to `Prompt.NoPrompt` to prevent the account selection dialog from being displayed (as the user is already signed-in and has an active cookie session).

```

private async void EditProfileButton_Click(object sender, RoutedEventArgs e)
{
    IEnumerable<IAccount> accounts = await app.GetAccountsAsync();
    try
    {
        var authResult = await app.AcquireToken(scopes:App.ApiScopes)
            .WithAccount(GetUserByPolicy(accounts, App.PolicyEditProfile)),
            .WithPrompt(Prompt.NoPrompt),
            .WithB2CAuthority(App.AuthorityEditProfile)
            .ExecuteAsync();
        DisplayBasicTokenInfo(authResult);
    }
    catch
    {
        . . .
    }
}

```

Resource owner password credentials (ROPC) with Azure AD B2C

For more details on the ROPC flow, please see this [documentation](#).

This flow is **not recommended** because your application asking a user for their password is not secure. For more information about this problem, see [this article](#).

By using username/password, you are giving up a number of things:

- Core tenets of modern identity: password gets fished, replayed. Because we have this concept of a share secret that can be intercepted. This is incompatible with passwordless.
- Users who need to do MFA won't be able to sign in (as there is no interaction).
- Users won't be able to do single sign-on.

Configure the ROPC flow in Azure AD B2C

In your Azure AD B2C tenant, create a new user flow and select **Sign in using ROPC**. This will enable the ROPC policy for your tenant. See [Configure the resource owner password credentials flow](#) for more details.

`IPublicClientApplication` contains a method:

```
AcquireTokenByUsernamePassword(
    IEnumerable<string> scopes,
    string username,
    SecureString password)
```

This method takes as parameters:

- The *scopes* to request an access token for.
- A *username*.
- A *SecureString password* for the user.

Remember to use the authority that contains the ROPC policy.

Limitations of the ROPC flow

- The ROPC flow **only works for local accounts** (where you register with Azure AD B2C using an email or username). This flow does not work if federating to any of the identity providers supported by Azure AD B2C (Facebook, Google, etc.).

Google Auth and Embedded Webview

If you are a Azure AD B2C developer using Google as an identity provider we recommend you use the system browser, as Google does not allow [authentication from embedded webviews](#). Currently, `login.microsoftonline.com` is a trusted authority with Google. Using this authority will work with embedded webview. However using `b2clogin.com` is not a trusted authority with Google, so users will not be able to authenticate.

We will provide an update to this [issue](#) if things change.

Caching with Azure AD B2C in MSAL.Net

Known issue with Azure AD B2C

MSAL.Net supports a [token cache](#). The token caching key is based on the claims returned by the Identity Provider. Currently MSAL.Net needs two claims to build a token cache key:

- `tid` which is the Azure AD Tenant ID, and
- `preferred_username`

Both these claims are missing in many of the Azure AD B2C scenarios.

The customer impact is that when trying to display the username field, are you getting "Missing from the token response" as the value? If so, this is because Azure AD B2C does not return a value in the IdToken for the preferred_username because of limitations with the social accounts and external identity providers (IdPs). Azure AD returns a value for preferred_username because it knows who the user is, but for Azure AD B2C, because the user can sign in with a local account, Facebook, Google, GitHub, etc. there is not a consistent value for Azure AD B2C to use for preferred_username. To unblock MSAL from rolling out cache compatibility with ADAL, we decided to use "Missing from the token response" on our end when dealing with the Azure AD B2C accounts when the IdToken returns nothing for preferred_username. MSAL must return a value for preferred_username to maintain cache compatibility across libraries.

Workarounds

Mitigation for the missing tenant ID

The suggested workaround is to use the [Caching by Policy](#)

Alternatively, you can use the `tid` claim, if you are using the [B2C custom policies](#), because it provides the capability to return additional claims to the application. To learn more about [Claims Transformation](#)

Mitigation for "Missing from the token response"

One option is to use the "name" claim as the preferred username. The process is mentioned in this [B2C doc](#) -> "In the Return claim column, choose the claims you want returned in the authorization tokens sent back to your application after a successful profile editing experience. For example, select Display Name, Postal Code."

Next steps

More details about acquiring tokens interactively with MSAL.NET for Azure AD B2C applications are provided in the following sample.

| SAMPLE | PLATFORM | DESCRIPTION |
|---|-----------------------------------|---|
| active-directory-b2c-xamarin-native | Xamarin iOS, Xamarin Android, UWP | A simple Xamarin Forms app showcasing how to use MSAL.NET to authenticate users via Azure AD B2C, and access a Web API with the resulting tokens. |

How to: Configure MSAL for iOS and macOS to use different identity providers

11/4/2019 • 4 minutes to read • [Edit Online](#)

This article will show you how to configure your Microsoft authentication library app for iOS and macOS (MSAL) for different authorities such as Azure Active Directory (Azure AD), Business-to-Consumer (B2C), sovereign clouds, and guest users. Throughout this article, you can generally think of an authority as an identity provider.

Default authority configuration

`MSALPublicClientApplication` is configured with a default authority URL of `https://login.microsoftonline.com/common`, which is suitable for most Azure Active Directory (AAD) scenarios. Unless you're implementing advanced scenarios like national clouds, or working with B2C, you won't need to change it.

NOTE

Modern authentication with Active Directory Federation Services as identity provider (ADFS) is not supported (see [ADFS for Developers](#) for details). ADFS is supported through federation.

Change the default authority

In some scenarios, such as business-to-consumer (B2C), you may need to change the default authority.

B2C

To work with B2C, the [Microsoft Authentication Library \(MSAL\)](#) requires a different authority configuration. MSAL recognizes one authority URL format as B2C by itself. The recognized B2C authority format is

`https://<host>/tfp/<tenant>/<policy>`, for example

`https://login.microsoftonline.com/tfp/contoso.onmicrosoft.com/B2C_1_SignInPolicy`. However, you can also use any other supported B2C authority URLs by declaring authority as B2C authority explicitly.

To support an arbitrary URL format for B2C, `MSALB2CAuthority` can be set with an arbitrary URL, like this:

Objective-C

```
NSURL *authorityURL = [NSURL URLWithString:@"arbitrary URL"];
MSALB2CAuthority *b2cAuthority = [[MSALB2CAuthority alloc] initWithURL:authorityURL
                                                               error:&b2cAuthorityError];
```

Swift

```
guard let authorityURL = URL(string: "arbitrary URL") else {
    // Handle error
    return
}
let b2cAuthority = try MSALB2CAuthority(url: authorityURL)
```

All B2C authorities that don't use the default B2C authority format must be declared as known authorities.

Add each different B2C authority to the known authorities list even if authorities only differ in policy.

Objective-C

```
MSALPublicClientApplicationConfig *b2cApplicationConfig = [[MSALPublicClientApplicationConfig alloc]
    initWithClientId:@"your-client-id"
    redirectUri:@"your-redirect-uri"
    authority:b2cAuthority];
b2cApplicationConfig.knownAuthorities = @[b2cAuthority];
```

Swift

```
let b2cApplicationConfig = MSALPublicClientApplicationConfig(clientId: "your-client-id", redirectUri: "your-redirect-uri", authority: b2cAuthority)
b2cApplicationConfig.knownAuthorities = [b2cAuthority]
```

When your app requests a new policy, the authority URL needs to be changed because the authority URL is different for each policy.

To configure a B2C application, set `@property MSALAuthority *authority` with an instance of `MSALB2CAuthority` in `MSALPublicClientApplicationConfig` before creating `MSALPublicClientApplication`, like this:

Objective-C

```
// Create B2C authority URL
NSURL *authorityURL = [NSURL
URLWithString:@"https://login.microsoftonline.com/tfp/contoso.onmicrosoft.com/B2C_1_SignInPolicy"];

MSALB2CAuthority *b2cAuthority = [[MSALB2CAuthority alloc] initWithURL:authorityURL
    error:&b2cAuthorityError];

if (!b2cAuthority)
{
    // Handle error
    return;
}

// Create MSALPublicClientApplication configuration
MSALPublicClientApplicationConfig *b2cApplicationConfig = [[MSALPublicClientApplicationConfig alloc]
    initWithClientId:@"your-client-id"
    redirectUri:@"your-redirect-uri"
    authority:b2cAuthority];

// Initialize MSALPublicClientApplication
MSALPublicClientApplication *b2cApplication =
[[MSALPublicClientApplication alloc] initWithConfiguration:b2cApplicationConfig error:&error];

if (!b2cApplication)
{
    // Handle error
    return;
}
```

Swift

```

do{
    // Create B2C authority URL
    guard let authorityURL = URL(string:
        "https://login.microsoftonline.com/tfp/contoso.onmicrosoft.com/B2C_1_SignInPolicy") else {
        // Handle error
        return
    }
    let b2cAuthority = try MSALB2CAuthority(url: authorityURL)

    // Create MSALPublicClientApplication configuration
    let b2cApplicationConfig = MSALPublicClientApplicationConfig(clientId: "your-client-id", redirectUri:
        "your-redirect-uri", authority: b2cAuthority)

    // Initialize MSALPublicClientApplication
    let b2cApplication = try MSALPublicClientApplication(configuration: b2cApplicationConfig)
} catch {
    // Handle error
}

```

Sovereign clouds

If your app runs in a sovereign cloud, you may need to change the authority URL in the `MSALPublicClientApplication`. The following example sets the authority URL to work with the German AAD cloud:

Objective-C

```

NSURL *authorityURL = [NSURL URLWithString:@"https://login.microsoftonline.de/common"];
MSALAuthority *sovereignAuthority = [MSALAuthority authorityWithURL:authorityURL error:&authorityError];

if (!sovereignAuthority)
{
    // Handle error
    return;
}

MSALPublicClientApplicationConfig *applicationConfig = [[MSALPublicClientApplicationConfig alloc]
    initWithClientId:@"your-client-id"
    redirectUri:@"your-redirect-uri"
    authority:sovereignAuthority];

MSALPublicClientApplication *sovereignApplication = [[MSALPublicClientApplication alloc]
initWithConfiguration:applicationConfig error:&error];

if (!sovereignApplication)
{
    // Handle error
    return;
}

```

Swift

```

do{
    guard let authorityURL = URL(string: "https://login.microsoftonline.de/common") else {
        //Handle error
        return
    }
    let sovereignAuthority = try MSALAuthority(url: authorityURL)

    let applicationConfig = MSALPublicClientApplicationConfig(clientId: "your-client-id", redirectUri: "your-
redirect-uri", authority: sovereignAuthority)

    let sovereignApplication = try MSALPublicClientApplication(configuration: applicationConfig)
} catch {
    // Handle error
}

```

You may need to pass different scopes to each sovereign cloud. Which scopes to send depends on the resource that you're using. For example, you might use `"https://graph.microsoft.com/user.read"` in worldwide cloud, and `"https://graph.microsoft.de/user.read"` in German cloud.

Siging a user into a specific tenant

When the authority URL is set to `"login.microsoftonline.com/common"`, the user will be signed into their home tenant. However, some apps may need to sign the user into a different tenant and some apps only work with a single tenant.

To sign the user into a specific tenant, configure `MSALPublicClientApplication` with a specific authority. For example:

```
https://login.microsoftonline.com/469fdeb4-d4fd-4fde-991e-308a78e4bea4
```

The following shows how to sign a user into a specific tenant:

Objective-C

```

NSURL *authorityURL = [NSURL URLWithString:@"https://login.microsoftonline.com/469fdeb4-d4fd-4fde-991e-
308a78e4bea4"];
MSALAADAuthority *tenantedAuthority = [[MSALAADAuthority alloc] initWithURL:authorityURL
error:&authorityError];

if (!tenantedAuthority)
{
    // Handle error
    return;
}

MSALPublicClientApplicationConfig *applicationConfig = [[MSALPublicClientApplicationConfig alloc]
initWithClientId:@"your-client-id"
redirectUri:@"your-redirect-uri"
authority:tenantedAuthority];

MSALPublicClientApplication *application =
[[MSALPublicClientApplication alloc] initWithConfiguration:applicationConfig error:&error];

if (!application)
{
    // Handle error
    return;
}

```

Swift

```

do{
    guard let authorityURL = URL(string: "https://login.microsoftonline.com/469fdeb4-d4fd-4fde-991e-
308a78e4bea4") else {
        //Handle error
        return
    }
    let tenantedAuthority = try MSALAADAuthority(url: authorityURL)

    let applicationConfig = MSALPublicClientApplicationConfig(clientId: "your-client-id", redirectUri: "your-
redirect-uri", authority: tenantedAuthority)

    let application = try MSALPublicClientApplication(configuration: applicationConfig)
} catch {
    // Handle error
}

```

Supported authorities

MSALAuthority

The `MSALAuthority` class is the base abstract class for the MSAL authority classes. Don't try to create instance of it using `alloc` or `new`. Instead, either create one of its subclasses directly (`MSALAADAuthority`, `MSALB2CAuthority`) or use the factory method `authorityWithURL:error:` to create subclasses using an authority URL.

Use the `url` property to get a normalized authority URL. Extra parameters and path components or fragments that aren't part of authority won't be in the returned normalized authority URL.

The following are subclasses of `MSALAuthority` that you can instantiate depending on the authority want to use.

MSALAADAuthority

`MSALAADAuthority` represents an AAD authority. The authority url should be in the following format, where `<port>` is optional: `https://<host>:<port>/<tenant>`

MSALB2CAuthority

`MSALB2CAuthority` represents a B2C authority. By default, the B2C authority url should be in the following format, where `<port>` is optional: `https://<host>:<port>/tfp/<tenant>/<policy>`. However, MSAL also supports other arbitrary B2C authority formats.

Next steps

Learn more about [Authentication flows and application scenarios](#)

Using web browsers in MSAL.NET

10/23/2019 • 8 minutes to read • [Edit Online](#)

Web browsers are required for interactive authentication. By default, MSAL.NET supports the [system web browser](#) on Xamarin.iOS and Xamarin.Android. But [you can also enable the Embedded Web browser](#) depending on your requirements (UX, need for single sign-on (SSO), security) in [Xamarin.iOS](#) and [Xamarin.Android](#) apps. And you can even [choose dynamically](#) which web browser to use based on the presence of Chrome or a browser supporting Chrome custom tabs in Android. MSAL.NET only supports the system browser in .NET Core desktop applications.

Web browsers in MSAL.NET

Interaction happens in a Web browser

It's important to understand that when acquiring a token interactively, the content of the dialog box isn't provided by the library but by the STS (Security Token Service). The authentication endpoint sends back some HTML and JavaScript that controls the interaction, which is rendered in a web browser or web control. Allowing the STS to handle the HTML interaction has many advantages:

- The password (if one was typed) is never stored by the application, nor the authentication library.
- Enables redirections to other identity providers (for instance login-in with a work school account or a personal account with MSAL, or with a social account with Azure AD B2C).
- Lets the STS control Conditional Access, for example, by having the user do multiple factor authentication (MFA) during the authentication phase (entering a Windows Hello pin, or being called on their phone, or on an authentication app on their phone). In cases where the required multi factor authentication isn't set up yet, the user can set it up just in time in the same dialog. The user enters their mobile phone number and is guided to install an authentication application and scan a QR tag to add their account. This server driven interaction is a great experience!
- Lets the user change their password in this same dialog when the password has expired (providing additional fields for the old password and the new password).
- Enables branding of the tenant, or the application (images) controlled by the Azure AD tenant admin / application owner.
- Enables the users to consent to let the application access resources / scopes in their name just after the authentication.

Embedded vs System Web UI

MSAL.NET is a multi-framework library and has framework-specific code to host a browser in a UI control (for example, on .Net Classic it uses WinForms, on Xamarin it uses native mobile controls etc.). This control is called [embedded](#) web UI. Alternatively, MSAL.NET is also able to kick off the system OS browser.

Generally, it's recommended that you use the platform default, and this is typically the system browser. The system browser is better at remembering the users that have logged in before. If you need to change this behavior, use

```
WithUseEmbeddedWebView(bool)
```

At a glance

| FRAMEWORK | EMBEDDED | SYSTEM | DEFAULT |
|--------------|----------|--------|----------|
| .NET Classic | Yes | Yes^ | Embedded |

| FRAMEWORK | EMBEDDED | SYSTEM | DEFAULT |
|-----------------|----------|--------|----------|
| .NET Core | No | Yes^ | System |
| .NET Standard | No | Yes^ | System |
| UWP | Yes | No | Embedded |
| Xamarin.Android | Yes | Yes | System |
| Xamarin.iOS | Yes | Yes | System |
| Xamarin.Mac | Yes | No | Embedded |

^ Requires "http://localhost" redirect URI

System web browser on Xamarin.iOS, Xamarin.Android

By default, MSAL.NET supports the system web browser on Xamarin.iOS, Xamarin.Android, and .NET Core. For all the platforms that provide UI (that is, not .NET Core), a dialog is provided by the library embedding a Web browser control. MSAL.NET also uses an embedded web view for the .NET Desktop and WAB for the UWP platform. However, it leverages by default the **system web browser** for Xamarin iOS and Xamarin Android applications. On iOS, it even chooses the web view to use depending on the version of the Operating System (iOS12, iOS11, and earlier).

Using the system browser has the significant advantage of sharing the SSO state with other applications and with web applications without needing a broker (Company portal / Authenticator). The system browser was used, by default, in MSAL.NET for the Xamarin iOS and Xamarin Android platforms because, on these platforms, the system web browser occupies the whole screen, and the user experience is better. The system web view isn't distinguishable from a dialog. On iOS, though, the user might have to give consent for the browser to call back the application, which can be annoying.

System browser experience on .NET Core

On .NET Core, MSAL.NET will start the system browser as a separate process. MSAL.NET doesn't have control over this browser, but once the user finishes authentication, the web page is redirected in such a way that MSAL.NET can intercept the Uri.

You can also configure apps written for .NET Classic to use this browser, by specifying

```
await pca.AcquireTokenInteractive(s_scopes)
    .WithUseEmbeddedWebView(false)
```

MSAL.NET cannot detect if the user navigates away or simply closes the browser. Apps using this technique are encouraged to define a timeout (via `cancellationToken`). We recommend a timeout of at least a few minutes, to take into account cases where the user is prompted to change password or perform multi-factor-authentication.

How to use the Default OS Browser

MSAL.NET needs to listen on `http://localhost:port` and intercept the code that AAD sends when the user is done authenticating (See [Authorization code](#) for details)

To enable the system browser:

1. During app registration, configure `http://localhost` as a redirect uri (not currently supported by B2C)

2. When you construct your PublicClientApplication, specify this redirect uri:

```
IPublicClientApplication pca = PublicClientApplicationBuilder
    .Create("<CLIENT_ID>")
    // or use a known port if you wish "http://localhost:1234"
    .WithRedirectUri("http://localhost")
    .Build();
```

NOTE

If you configure `http://localhost`, internally MSAL.NET will find a random open port and use it.

Linux and MAC

On Linux, MSAL.NET will open the default OS browser using the `xdg-open` tool. To troubleshoot, run the tool from a terminal for example, `xdg-open "https://www.bing.com"`

On Mac, the browser is opened by invoking `open <url>`

Customizing the experience

NOTE

Customization is available in MSAL.NET 4.1.0 or later.

MSAL.NET is able to respond with an HTTP message when a token is received or in case of error. You can display an HTML message or redirect to an url of your choice:

```
var options = new SystemWebViewOptions()
{
    HtmlMessageError = "<p> An error occurred: {0}. Details {1}</p>",
    BrowserRedirectSuccess = new Uri("https://www.microsoft.com");
}

await pca.AcquireTokenInteractive(s_scopes)
    .WithUseEmbeddedWebView(false)
    .WithSystemWebViewOptions(options)
    .ExecuteAsync();
```

Opening a specific browser (Experimental)

You may customize the way MSAL.NET opens the browser. For example instead of using whatever browser is the default, you can force open a specific browser:

```
var options = new SystemWebViewOptions()
{
    OpenBrowserAsync = SystemWebViewOptions.OpenWithEdgeBrowserAsync
}
```

UWP doesn't use the System Webview

For desktop applications, however, launching a System Webview leads to a subpar user experience, as the user sees the browser, where they might already have other tabs opened. And when authentication has happened, the user gets a page asking them to close this window. If the user doesn't pay attention, they can close the entire process (including other tabs, which are unrelated to the authentication). Leveraging the system browser on desktop would also require opening local ports and listening on them, which might require advanced permissions for the application. You, as a developer, user, or administrator, might be reluctant about this requirement.

Enable embedded webviews on iOS and Android

You can also enable embedded webviews in Xamarin.iOS and Xamarin.Android apps. Starting with MSAL.NET 2.0.0-preview, MSAL.NET also supports using the **embedded** webview option. For ADAL.NET, embedded webview is the only option supported.

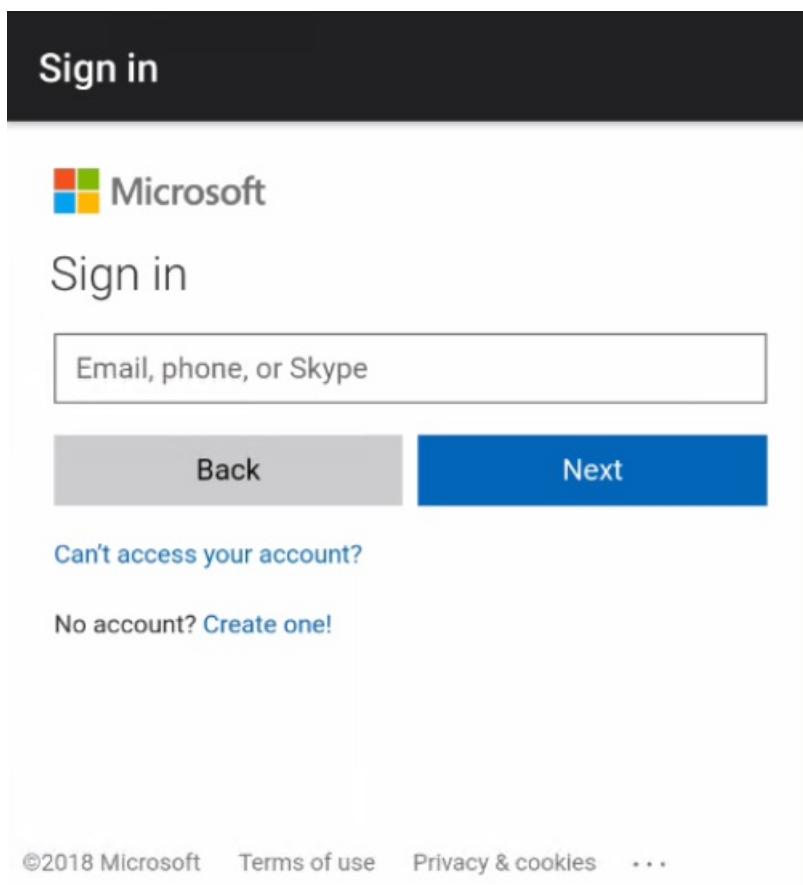
As a developer using MSAL.NET targeting Xamarin, you may choose to use either embedded webviews or system browsers. This is your choice depending on the user experience and security concerns you want to target.

Currently, MSAL.NET doesn't yet support the Android and iOS brokers. Therefore if you need to provide single sign-on (SSO), the system browser might still be a better option. Supporting brokers with the embedded web browser is on the MSAL.NET backlog.

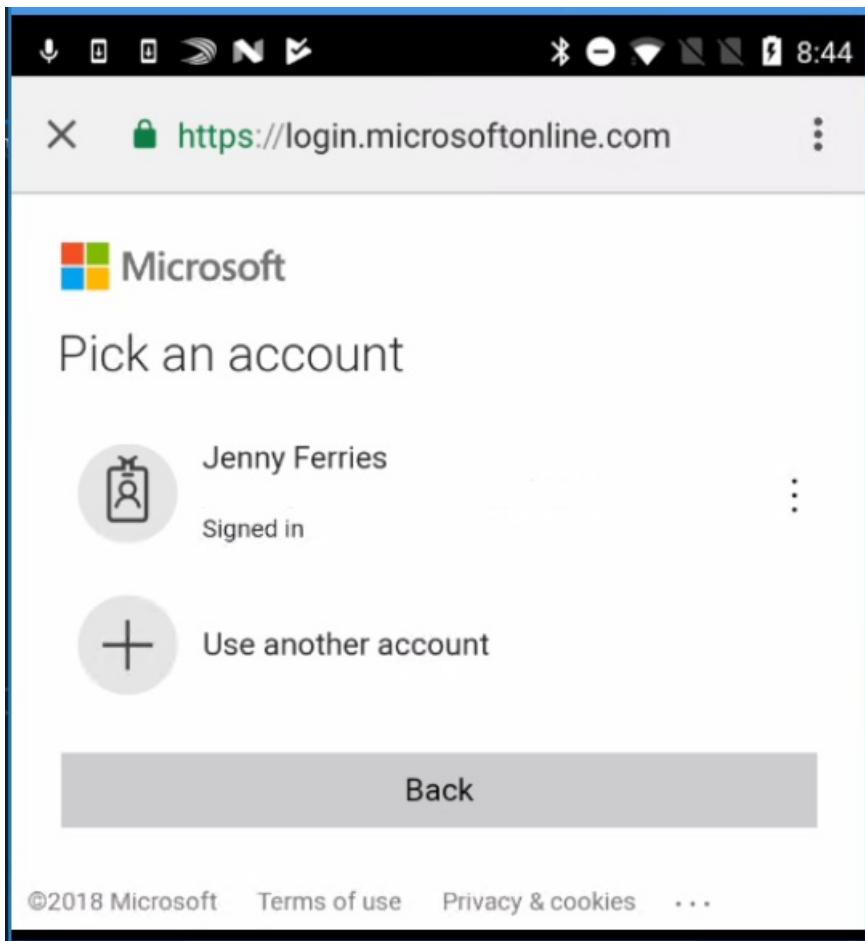
Differences between embedded webview and system browser

There are some visual differences between embedded webview and system browser in MSAL.NET.

Interactive sign-in with MSAL.NET using the Embedded Webview:



Interactive sign-in with MSAL.NET using the System Browser:



Developer Options

As a developer using MSAL.NET, you have several options for displaying the interactive dialog from STS:

- **System browser.** The system browser is set by default in the library. If using Android, read [system browsers](#) for specific information about which browsers are supported for authentication. When using the system browser in Android, we recommend the device has a browser that supports Chrome custom tabs. Otherwise, authentication may fail.
- **Embedded webview.** To use only embedded webview in MSAL.NET, the `AcquireTokenInteractively` parameters builder contains a `WithUseEmbeddedWebView()` method.

iOS

```
AuthenticationResult authResult;
authResult = app.AcquireTokenInteractively(scopes)
    .WithUseEmbeddedWebView(useEmbeddedWebview)
    .ExecuteAsync();
```

Android:

```
authResult = app.AcquireTokenInteractively(scopes)
    .WithParentActivityOrWindow(activity)
    .WithUseEmbeddedWebView(useEmbeddedWebview)
    .ExecuteAsync();
```

Choosing between embedded web browser or system browser on Xamarin.iOS

In your iOS app, in `AppDelegate.cs` you can initialize the `ParentWindow` to `null`. It's not used in iOS

```
App.ParentWindow = null; // no UI parent on iOS
```

Choosing between embedded web browser or system browser on Xamarin.Android

In your Android app, in `MainActivity.cs` you can set the parent activity, so that the authentication result gets back to it:

```
App.ParentWindow = this;
```

Then in the `MainPage.xaml.cs`:

```
authResult = await App.PCA.AcquireTokenInteractive(App.Scopes)
    .WithParentActivityOrWindow(App.ParentWindow)
    .WithUseEmbeddedWebView(true)
    .ExecuteAsync();
```

Detecting the presence of custom tabs on Xamarin.Android

If you want to use the system web browser to enable SSO with the apps running in the browser, but are worried about the user experience for Android devices not having a browser with custom tab support, you have the option to decide by calling the `IsSystemWebViewAvailable()` method in `IPublicClientApplication`. This method returns `true` if the PackageManager detects custom tabs and `false` if they aren't detected on the device.

Based on the value returned by this method, and your requirements, you can make a decision:

- You can return a custom error message to the user. For example: "Please install Chrome to continue with authentication" -OR-
- You can fall back to the embedded webview option and launch the UI as an embedded webview.

The code below shows the embedded webview option:

```
bool useSystemBrowser = app.IsSystemWebViewAvailable();

authResult = await App.PCA.AcquireTokenInteractive(App.Scopes)
    .WithParentActivityOrWindow(App.ParentWindow)
    .WithUseEmbeddedWebView(!useSystemBrowser)
    .ExecuteAsync();
```

.NET Core doesn't support interactive authentication with an embedded browser

For .NET Core, acquisition of tokens interactively is only available through the system web browser, not with embedded web views. Indeed, .NET Core doesn't provide UI yet. If you want to customize the browsing experience with the system web browser, you can implement the `IWithCustomUI` interface and even provide your own browser.

Use Internet Explorer and Microsoft Edge browsers with MSAL.js

10/30/2019 • 2 minutes to read • [Edit Online](#)

Microsoft Authentication Library for JavaScript (MSAL.js) is generated for [JavaScript ES5](#) so that it can run in Internet Explorer. There are, however, a few things to know.

Run an app in Internet Explorer

If you intend to use MSAL.js in applications that can run in Internet Explorer, you will need to add a reference to a promise polyfill before referencing the MSAL.js script.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/bluebird/3.3.4/bluebird.min.js" class="pre"></script>
```

This is because Internet Explorer does not support JavaScript promises natively.

Debugging an application running in Internet Explorer

Running in production

Deploying your application to production (for instance in Azure Web apps) normally works fine, provided the end user has accepted popups. We tested it with Internet Explorer 11.

Running locally

If you want to run and debug locally your application running in Internet Explorer, you need to be aware of the following considerations (assume that you want to run your application as `http://localhost:1234`):

- Internet Explorer has a security mechanism named "protected mode", which prevents MSAL.js from working correctly. Among the symptoms, after you sign in, the page can be redirected to `http://localhost:1234/null`.
- To run and debug your application locally, you'll need to disable this "protected mode". For this:
 1. Click Internet Explorer **Tools** (the gear icon).
 2. Select **Internet Options** and then the **Security** tab.
 3. Click on the **Internet** zone, and uncheck **Enable Protected Mode (requires restarting Internet Explorer)**. Internet Explorer warns that your computer is no longer protected. Click **OK**.
 4. Restart Internet Explorer.
 5. Run and debug your application.

When you are done, restore the Internet Explorer security settings. Select **Settings -> Internet Options -> Security -> Reset all zones to default level**.

Next steps

Learn more about [Known issues when using MSAL.js in Internet Explorer](#).

Known issues on Internet Explorer and Microsoft Edge browsers with MSAL.js

10/27/2019 • 4 minutes to read • [Edit Online](#)

Issues due to security zones

We had multiple reports of issues with authentication in IE and Microsoft Edge (since the update of the *Microsoft Edge browser version to 40.15063.0.0*). We are tracking these and have informed the Microsoft Edge team. While Microsoft Edge works on a resolution, here is a description of the frequently occurring issues and the possible workarounds that can be implemented.

Cause

The cause for most of these issues is as follows. The session storage and local storage are partitioned by security zones in the Microsoft Edge browser. In this particular version of Microsoft Edge, when the application is redirected across zones, the session storage and local storage are cleared. Specifically, the session storage is cleared in the regular browser navigation, and both the session and local storage are cleared in the InPrivate mode of the browser. MSAL.js saves certain state in the session storage and relies on checking this state during the authentication flows. When the session storage is cleared, this state is lost and hence results in broken experiences.

Issues

- **Infinite redirect loops and page reloads during authentication.** When users sign in to the application on Microsoft Edge, they are redirected back from the AAD login page and are stuck in an infinite redirect loop resulting in repeated page reloads. This is usually accompanied by an `invalid_state` error in the session storage.
- **Infinite acquire token loops and AADSTS50058 error.** When an application running on Microsoft Edge tries to acquire a token for a resource, the application may get stuck in an infinite loop of the acquire token call along with the following error from AAD in your network trace:

```
Error :login_required; Error description:AADSTS50058: A silent sign-in request was sent but no user is signed in. The cookies used to represent the user's session were not sent in the request to Azure AD. This can happen if the user is using Internet Explorer or Edge, and the web app sending the silent sign-in request is in different IE security zone than the Azure AD endpoint (login.microsoftonline.com)
```

- **Popup window doesn't close or is stuck when using login through Popup to authenticate.** When authenticating through popup window in Microsoft Edge or IE(InPrivate), after entering credentials and signing in, if multiple domains across security zones are involved in the navigation, the popup window doesn't close because MSAL.js loses the handle to the popup window.

Here are links to these issues in the Microsoft Edge issue tracker:

- [Bug 13861050](#)
- [Bug 13861663](#)

Update: Fix available in MSAL.js 0.2.3

Fixes for the authentication redirect loop issues have been released in [MSAL.js 0.2.3](#). Enable the flag `storeAuthStateInCookie` in the MSAL.js config to take advantage of this fix. By default this flag is set to false.

When the `storeAuthStateInCookie` flag is enabled, MSAL.js will use the browser cookies to store the request state required for validation of the auth flows.

NOTE

This fix is not yet available for the msal-angular and msal-angularjs wrappers. This fix does not address the issue with Popup windows.

Use workarounds below.

Other workarounds

Make sure to test that your issue is occurring only on the specific version of Microsoft Edge browser and works on the other browsers before adopting these workarounds.

1. As a first step to get around these issues, ensure that the application domain, , and any other sites involved in the redirects of the authentication flow are added as trusted sites in the security settings of the browser, so that they belong to the same security zone. To do so, follow these steps:
 - Open **Internet Explorer** and click on the **settings** (gear icon) in the top-right corner
 - Select **Internet Options**
 - Select the **Security** tab
 - Under the **Trusted Sites** option, click on the **sites** button and add the URLs in the dialog box that opens.
2. As mentioned before, since only the session storage is cleared during the regular navigation, you may configure MSAL.js to use the local storage instead. This can be set as the `cacheLocation` config parameter while initializing MSAL.

Note, this will not solve the issue for InPrivate browsing since both session and local storage are cleared.

Issues due to popup blockers

There are cases when popups are blocked in IE or Microsoft Edge, for example when a second popup occurs during multi-factor authentication. You will get an alert in the browser to allow for the popup once or always. If you choose to allow, the browser opens the popup window automatically and returns a `null` handle for it. As a result, the library does not have a handle for the window and there is no way to close the popup window. The same issue does not happen in Chrome when it prompts you to allow popups because it does not automatically open a popup window.

As a **workaround**, developers will need to allow popups in IE and Microsoft Edge before they start using their app to avoid this issue.

Next steps

Learn more about [Using MSAL.js in Internet Explorer](#).

Known issues on Safari browser with MSAL.js

10/27/2019 • 2 minutes to read • [Edit Online](#)

Silent token renewal on Safari 12 and ITP 2.0

Apple iOS 12 and MacOS 10.14 operating systems included a release of the [Safari 12 browser](#). For purposes of security and privacy, Safari 12 includes the [Intelligent Tracking Prevention 2.0](#). This essentially causes the browser to drop third-party cookies being set. ITP 2.0 also treats the cookies set by identity providers as third-party cookies.

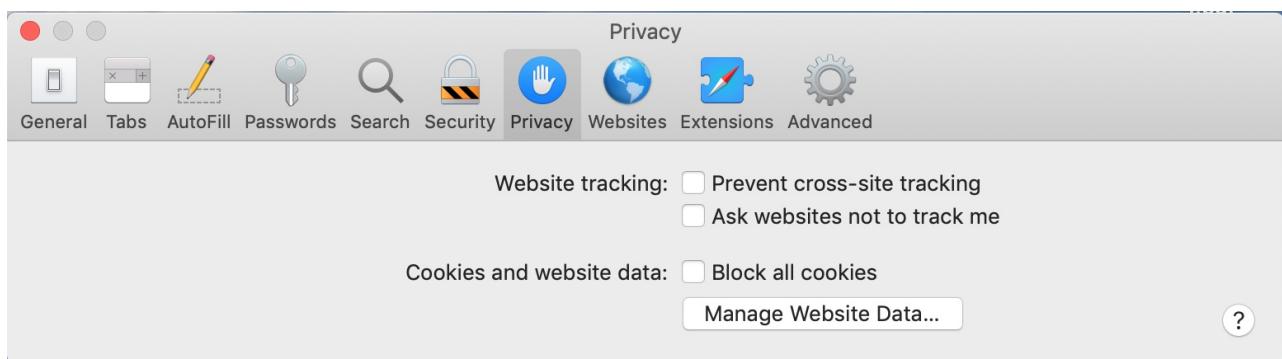
Impact on MSAL.js

MSAL.js uses a hidden Iframe to perform silent token acquisition and renewal as part of the `acquireTokenSilent` calls. The silent token requests rely on the Iframe having access to the authenticated user session represented by the cookies set by Azure AD. With ITP 2.0 preventing access to these cookies, MSAL.js fails to silently acquire and renew tokens and this results in `acquireTokenSilent` failures.

There is no solution for this issue at this point and we are evaluating options with the standards community.

Work around

By default the ITP setting is enabled on Safari browser. You can disable this setting by navigating to **Preferences -> Privacy** and unchecking the **Prevent cross-site tracking** option.



You will need to handle the `acquireTokenSilent` failures with an interactive acquire token call, which prompts the user to sign in. To avoid repeated sign-ins, an approach you can implement is to handle the `acquireTokenSilent` failure and provide the user an option to disable the ITP setting in Safari before proceeding with the interactive call. Once the setting is disabled, subsequent silent token renewals should succeed.

How to: Troubleshoot MSAL for iOS and macOS SSL issues

10/23/2019 • 2 minutes to read • [Edit Online](#)

This article provides information to help you troubleshoot issues that you may come across while using the [Microsoft Authentication Library \(MSAL\) for iOS and macOS](#)

Network issues

Error -1200: "An SSL error has occurred and a secure connection to the server can't be made."

This error means that the connection isn't secure. It occurs when a certificate is invalid. For more information, including which server is failing the SSL check, refer to `NSURLErrorFailingURLErrorKey` in the `userInfo` dictionary of the error object.

This error is from Apple's networking library. A full list of NSURLConnection error codes is in `NSURLConnection.h` in the macOS and iOS SDKs. For more details about this error, see [URL Loading System Error Codes](#).

Certificate issues

If the URL providing an invalid certificate connects to the server that you intend to use as part of the authentication flow, a good start to diagnosing the problem is to test the URL with a SSL validation service such as [Qualys SSL Labs Analyzer](#). It tests the server against a wide array of scenarios and browsers and checks for many known vulnerabilities.

By default, Apple's new [App Transport Security \(ATS\)](#) feature applies more stringent security policies to apps that use SSL certificates. Some operating systems and web browsers have started enforcing some of these policies by default. For security reasons, we recommend you not disable ATS.

Certificates using SHA-1 hashes have known vulnerabilities. Most modern web browsers don't allow certificates with SHA-1 hashes.

Captive portals

A captive portal presents a web page to a user when they first access a Wi-Fi network and haven't yet been granted access to that network. It intercepts their internet traffic until the user satisfies the requirements of the portal. Network errors because the user can't connect to network resources are expected until the user connects through the portal.

Next steps

Learn about [captive portals](#) and Apple's new [App Transport Security \(ATS\)](#) feature.

Application types for Microsoft identity platform

11/12/2019 • 5 minutes to read • [Edit Online](#)

The Microsoft identity platform (v2.0) endpoint supports authentication for a variety of modern app architectures, all of them based on industry-standard protocols [OAuth 2.0 or OpenID Connect](#). This article describes the types of apps that you can build by using Microsoft identity platform, regardless of your preferred language or platform. The information is designed to help you understand high-level scenarios before you [start working with the code](#).

NOTE

The Microsoft identity platform endpoint doesn't support all Azure Active Directory (Azure AD) scenarios and features. To determine whether you should use the Microsoft identity platform endpoint, read about [Microsoft identity platform limitations](#).

The basics

You must register each app that uses the Microsoft identity platform endpoint in the new [App registrations portal](#). The app registration process collects and assigns these values for your app:

- An **Application (client) ID** that uniquely identifies your app
- A **Redirect URI** that you can use to direct responses back to your app
- A few other scenario-specific values such as supported account types

For details, learn how to [register an app](#).

After the app is registered, the app communicates with Microsoft identity platform by sending requests to the endpoint. We provide open-source frameworks and libraries that handle the details of these requests. You also have the option to implement the authentication logic yourself by creating requests to these endpoints:

```
https://login.microsoftonline.com/common/oauth2/v2.0/authorize  
https://login.microsoftonline.com/common/oauth2/v2.0/token
```

Single-page apps (JavaScript)

Many modern apps have a single-page app front end that primarily is written in JavaScript. Often, it's written by using a framework like Angular, React, or Vue. The Microsoft identity platform endpoint supports these apps by using the [OAuth 2.0 implicit flow](#).

In this flow, the app receives tokens directly from the Microsoft identity platform authorize endpoint, without any server-to-server exchanges. All authentication logic and session handling takes place entirely in the JavaScript client, without extra page redirects.

To see this scenario in action, try one of the single-page app code samples in the [Microsoft identity platform getting started](#) section.

Web apps

For web apps (.NET, PHP, Java, Ruby, Python, Node) that the user accesses through a browser, you can use

OpenID Connect for user sign-in. In OpenID Connect, the web app receives an ID token. An ID token is a security token that verifies the user's identity and provides information about the user in the form of claims:

```
// Partial raw ID token  
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6ImtyaU1QZG1Cd...  
  
// Partial content of a decoded ID token  
{  
    "name": "John Smith",  
    "email": "john.smith@gmail.com",  
    "oid": "d9674823-dff-4e3f-a6eb-62fe4bd48a58"  
    ...  
}
```

Further details of different types of tokens used in the Microsoft identity platform endpoint are available in the [access token reference](#) and [id_token reference](#)

In web server apps, the sign-in authentication flow takes these high-level steps:

You can ensure the user's identity by validating the ID token with a public signing key that is received from the Microsoft identity platform endpoint. A session cookie is set, which can be used to identify the user on subsequent page requests.

To see this scenario in action, try one of the web app sign-in code samples in the [Microsoft identity platform getting started](#) section.

In addition to simple sign-in, a web server app might need to access another web service, such as a REST API. In this case, the web server app engages in a combined OpenID Connect and OAuth 2.0 flow, by using the [OAuth 2.0 authorization code flow](#). For more information about this scenario, read about [getting started with web apps and Web APIs](#).

Web APIs

You can use the Microsoft identity platform endpoint to secure web services, such as your app's RESTful Web API. Web APIs can be implemented in numerous platforms and languages. They can also be implemented using HTTP Triggers in Azure Functions. Instead of ID tokens and session cookies, a Web API uses an OAuth 2.0 access token to secure its data and to authenticate incoming requests. The caller of a Web API appends an access token in the authorization header of an HTTP request, like this:

```
GET /api/items HTTP/1.1  
Host: www.mywebapi.com  
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6...  
Accept: application/json  
...
```

The Web API uses the access token to verify the API caller's identity and to extract information about the caller from claims that are encoded in the access token. Further details of different types of tokens used in the Microsoft identity platform endpoint are available in the [access token reference](#) and [id_token reference](#)

A Web API can give users the power to opt in or opt out of specific functionality or data by exposing permissions, also known as [scopes](#). For a calling app to acquire permission to a scope, the user must consent to the scope during a flow. The Microsoft identity platform endpoint asks the user for permission, and then records permissions in all access tokens that the Web API receives. The Web API validates the access tokens it receives on each call and performs authorization checks.

A Web API can receive access tokens from all types of apps, including web server apps, desktop and mobile apps,

single-page apps, server-side daemons, and even other Web APIs. The high-level flow for a Web API looks like this:

To learn how to secure a Web API by using OAuth2 access tokens, check out the Web API code samples in the [Microsoft identity platform getting started](#) section.

In many cases, web APIs also need to make outbound requests to other downstream web APIs secured by Microsoft identity platform. To do so, web APIs can take advantage of the **On-Behalf-Of** flow, which allows the web API to exchange an incoming access token for another access token to be used in outbound requests. For more info, see [Microsoft identity platform and OAuth 2.0 On-Behalf-Of flow](#).

Mobile and native apps

Device-installed apps, such as mobile and desktop apps, often need to access back-end services or Web APIs that store data and perform functions on behalf of a user. These apps can add sign-in and authorization to back-end services by using the [OAuth 2.0 authorization code flow](#).

In this flow, the app receives an authorization code from the Microsoft identity platform endpoint when the user signs in. The authorization code represents the app's permission to call back-end services on behalf of the user who is signed in. The app can exchange the authorization code in the background for an OAuth 2.0 access token and a refresh token. The app can use the access token to authenticate to Web APIs in HTTP requests, and use the refresh token to get new access tokens when older access tokens expire.

Daemons and server-side apps

Apps that have long-running processes or that operate without interaction with a user also need a way to access secured resources, such as Web APIs. These apps can authenticate and get tokens by using the app's identity, rather than a user's delegated identity, with the OAuth 2.0 client credentials flow. You can prove the app's identity using a client secret or certificate. For more info, see [Authenticating to Microsoft identity platform in daemon apps with certificates](#).

In this flow, the app interacts directly with the `/token` endpoint to obtain access:

To build a daemon app, see the [client credentials documentation](#), or try a [.NET sample app](#).

Microsoft identity platform protocols

8/6/2019 • 4 minutes to read • [Edit Online](#)

The Microsoft identity platform endpoint for identity-as-a-service with industry standard protocols, OpenID Connect and OAuth 2.0. While the service is standards-compliant, there can be subtle differences between any two implementations of these protocols. The information here will be useful if you choose to write your code by directly sending and handling HTTP requests or use a third party open-source library, rather than using one of our [open-source libraries](#).

NOTE

Not all Azure AD scenarios and features are supported by the Microsoft identity platform endpoint. To determine if you should use the Microsoft identity platform endpoint, read about [Microsoft identity platform limitations](#).

The basics

In nearly all OAuth 2.0 and OpenID Connect flows, there are four parties involved in the exchange:

- The **Authorization Server** is the Microsoft identity platform endpoint and responsible for ensuring the user's identity, granting and revoking access to resources, and issuing tokens. The authorization server also known as the identity provider - it securely handles anything to do with the user's information, their access, and the trust relationships between parties in a flow.
- The **Resource Owner** is typically the end user. It's the party that owns the data and has the power to allow third parties to access that data or resource.
- The **OAuth Client** is your app, identified by its application ID. The OAuth client is usually the party that the end user interacts with, and it requests tokens from the authorization server. The client must be granted permission to access the resource by the resource owner.
- The **Resource Server** is where the resource or data resides. It trusts the Authorization Server to securely authenticate and authorize the OAuth Client, and uses Bearer access tokens to ensure that access to a resource can be granted.

App Registration

Every app that wants to accept both personal and work or school accounts must be registered through the **App registrations** experience in the [Azure portal](#) before it can sign these users in using OAuth 2.0 or OpenID Connect. The app registration process will collect and assign a few values to your app:

- An **Application ID** that uniquely identifies your app
- A **Redirect URI** (optional) that can be used to direct responses back to your app
- A few other scenario-specific values.

For more details, learn how to [register an app](#).

Endpoints

Once registered, the app communicates with Microsoft identity platform by sending requests to the endpoint:

<https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize>
<https://login.microsoftonline.com/{tenant}/oauth2/v2.0/token>

Where the `{tenant}` can take one of four different values:

| VALUE | DESCRIPTION |
|---|---|
| <code>common</code> | Allows users with both personal Microsoft accounts and work/school accounts from Azure AD to sign into the application. |
| <code>organizations</code> | Allows only users with work/school accounts from Azure AD to sign into the application. |
| <code>consumers</code> | Allows only users with personal Microsoft accounts (MSA) to sign into the application. |
| <code>8eaeef023-2b34-4da1-9baa-8bc8c9d6a490</code> or
<code>contoso.onmicrosoft.com</code> | Allows only users with work/school accounts from a particular Azure AD tenant to sign into the application. Either the friendly domain name of the Azure AD tenant or the tenant's GUID identifier can be used. |

To learn how to interact with these endpoints, choose a particular app type in the [Protocols](#) section and follow the links for more info.

TIP

Any app registered in Azure AD can use the Microsoft identity platform endpoint, even if they don't sign in personal accounts. This way, you can migrate existing applications to Microsoft identity platform and [MSAL](#) without re-creating your application.

Tokens

The Microsoft identity platform implementation of OAuth 2.0 and OpenID Connect make extensive use of bearer tokens, including bearer tokens represented as JWTs. A bearer token is a lightweight security token that grants the "bearer" access to a protected resource. In this sense, the "bearer" is any party that can present the token. Though a party must first authenticate with Microsoft identity platform to receive the bearer token, if the required steps are not taken to secure the token in transmission and storage, it can be intercepted and used by an unintended party. While some security tokens have a built-in mechanism for preventing unauthorized parties from using them, bearer tokens do not have this mechanism and must be transported in a secure channel such as transport layer security (HTTPS). If a bearer token is transmitted in the clear, a malicious party can use a man-in-the-middle attack to acquire the token and use it for unauthorized access to a protected resource. The same security principles apply when storing or caching bearer tokens for later use. Always ensure that your app transmits and stores bearer tokens in a secure manner. For more security considerations on bearer tokens, see [RFC 6750 Section 5](#).

Further details of different types of tokens used in the Microsoft identity platform endpoint is available in the [Microsoft identity platform endpoint token reference](#).

Protocols

If you're ready to see some example requests, get started with one of the below tutorials. Each one corresponds to a particular authentication scenario. If you need help determining which is the right flow for you, check out the [types of apps you can build with Microsoft identity platform](#).

- Build mobile and native application with OAuth 2.0
- Build web apps with OpenID Connect
- Build single-page apps with the OAuth 2.0 Implicit Flow
- Build daemons or server-side processes with the OAuth 2.0 client credentials flow
- Get tokens in a web API with the OAuth 2.0 on-behalf-of Flow

Microsoft identity platform and OpenID Connect protocol

8/7/2019 • 14 minutes to read • [Edit Online](#)

OpenID Connect is an authentication protocol built on OAuth 2.0 that you can use to securely sign in a user to a web application. When you use the Microsoft identity platform endpoint's implementation of OpenID Connect, you can add sign-in and API access to your web-based apps. This article shows how to do this independent of language and describes how to send and receive HTTP messages without using any Microsoft open-source libraries.

NOTE

The Microsoft identity platform endpoint does not support all Azure Active Directory (Azure AD) scenarios and features. To determine whether you should use the Microsoft identity platform endpoint, read about [Microsoft identity platform limitations](#).

[OpenID Connect](#) extends the OAuth 2.0 *authorization* protocol to use as an *authentication* protocol, so that you can do single sign-on using OAuth. OpenID Connect introduces the concept of an *ID token*, which is a security token that allows the client to verify the identity of the user. The ID token also gets basic profile information about the user. Because OpenID Connect extends OAuth 2.0, apps can securely acquire *access tokens*, which can be used to access resources that are secured by an [authorization server](#). The Microsoft identity platform endpoint also allows third-party apps that are registered with Azure AD to issue access tokens for secured resources such as Web APIs. For more information about how to set up an application to issue access tokens, see [How to register an app with the Microsoft identity platform endpoint](#). We recommend that you use OpenID Connect if you are building a [web application](#) that is hosted on a server and accessed via a browser.

Protocol diagram: Sign-in

The most basic sign-in flow has the steps shown in the next diagram. Each step is described in detail in this article.

Fetch the OpenID Connect metadata document

OpenID Connect describes a metadata document that contains most of the information required for an app to do sign-in. This includes information such as the URLs to use and the location of the service's public signing keys. For the Microsoft identity platform endpoint, this is the OpenID Connect metadata document you should use:

```
https://login.microsoftonline.com/{tenant}/v2.0/.well-known/openid-configuration
```

TIP

Try it! Click <https://login.microsoftonline.com/common/v2.0/.well-known/openid-configuration> to see the `common` tenants configuration.

The `{tenant}` can take one of four values:

| VALUE | DESCRIPTION |
|--|---|
| common | Users with both a personal Microsoft account and a work or school account from Azure AD can sign in to the application. |
| organizations | Only users with work or school accounts from Azure AD can sign in to the application. |
| consumers | Only users with a personal Microsoft account can sign in to the application. |
| 8eaef023-2b34-4da1-9baa-8bc8c9d6a490 or
contoso.onmicrosoft.com | Only users from a specific Azure AD tenant (whether they are members in the directory with a work or school account, or they are guests in the directory with a personal Microsoft account) can sign in to the application. Either the friendly domain name of the Azure AD tenant or the tenant's GUID identifier can be used. You can also use the consumer tenant, 9188040d-6c67-4c5b-b112-36a304b66dad, in place of the consumers tenant. |

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](#).

```
{
  "authorization_endpoint": "https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize",
  "token_endpoint": "https://login.microsoftonline.com/{tenant}/oauth2/v2.0/token",
  "token_endpoint_auth_methods_supported": [
    "client_secret_post",
    "private_key_jwt"
  ],
  "jwks_uri": "https://login.microsoftonline.com/{tenant}/discovery/v2.0/keys",
  ...
}
```

If your app has custom signing keys as a result of using the [claims-mapping](#) feature, you must append an `appid` query parameter containing the app ID in order to get a `jwks_uri` pointing to your app's signing key information. For example:

`https://login.microsoftonline.com/{tenant}/.well-known/v2.0/openid-configuration?appid=6731de76-14a6-49ae-97bc-6eba6914391e`

contains a `jwks_uri` of

`https://login.microsoftonline.com/{tenant}/discovery/v2.0/keys?appid=6731de76-14a6-49ae-97bc-6eba6914391e`.

Typically, you would use this metadata document to configure an OpenID Connect library or SDK; the library would use the metadata to do its work. However, if you're not using a pre-built OpenID Connect library, you can follow the steps in the remainder of this article to do sign-in in a web app by using the Microsoft identity platform endpoint.

Send the sign-in request

When your web app needs to authenticate the user, it can direct the user to the `/authorize` endpoint. This request is similar to the first leg of the [OAuth 2.0 authorization code flow](#), with these important distinctions:

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

IMPORTANT

In order to successfully request an ID token from the /authorization endpoint, the app registration in the [registration portal](#) must have the implicit grant of id_tokens enabled in the Authentication tab (which sets the `oauth2AllowIdTokenImplicitFlow` flag in the [application manifest](#) to `true`). If it isn't enabled, an `unsupported_response` error will be returned: "The provided value for the input parameter 'response_type' isn't allowed for this client. Expected value is 'code'"

For example:

```
// Line breaks are for legibility only.

GET https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&response_type=id_token
&redirect_uri=http%3A%2Flocalhost%2Fmyapp%2F
&response_mode=form_post
&scope=openid
&state=12345
&nonce=678910
```

TIP

Click the following link to execute this request. After you sign in, your browser will be redirected to

<https://localhost/myapp/>, with an ID token in the address bar. Note that this request uses `response_mode=fragment` (for demonstration purposes only). We recommend that you use `response_mode=form_post`.

<https://login.microsoftonline.com/common/oauth2/v2.0/authorize...>

| PARAMETER | CONDITION | DESCRIPTION |
|----------------------------|-------------|--|
| <code>tenant</code> | Required | You can use the <code>{tenant}</code> value in the path of the request to control who can sign in to the application. The allowed values are <code>common</code> , <code>organizations</code> , <code>consumers</code> , and tenant identifiers. For more information, see protocol basics . |
| <code>client_id</code> | Required | The Application (client) ID that the Azure portal – App registrations experience assigned to your app. |
| <code>response_type</code> | Required | Must include <code>id_token</code> for OpenID Connect sign-in. It might also include other <code>response_type</code> values, such as <code>code</code> . |
| <code>redirect_uri</code> | 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 URLs you registered in the portal, except that it must be URL encoded. If not present, the endpoint will pick one registered redirect_uri at random to send the user back to. |

| PARAMETER | CONDITION | DESCRIPTION |
|----------------------------|-------------|--|
| <code>scope</code> | Required | A space-separated list of scopes. For OpenID Connect, it must include the scope <code>openid</code> , which translates to the "Sign you in" permission in the consent UI. You might also include other scopes in this request for requesting consent. |
| <code>nonce</code> | Required | A value included in the request, generated by the app, that will be included in the resulting <code>id_token</code> value as a claim. The app can verify this value to mitigate token replay attacks. The value typically is a randomized, unique string that can be used to identify the origin of the request. |
| <code>response_mode</code> | Recommended | Specifies the method that should be used to send the resulting authorization code back to your app. Can be <code>form_post</code> or <code>fragment</code> . For web applications, we recommend using <code>response_mode=form_post</code> , to ensure the most secure transfer of tokens to your application. |
| <code>state</code> | Recommended | A value included in the request that also will be returned in the token response. It can be a string of any content you want. A randomly generated unique value typically is used to prevent cross-site request forgery attacks . The state also is used to encode information about the user's state in the app before the authentication request occurred, such as the page or view the user was on. |

| PARAMETER | CONDITION | DESCRIPTION |
|--------------------------|-----------|---|
| <code>prompt</code> | Optional | Indicates the type of user interaction that is required. The only valid values at this time are <code>login</code> , <code>none</code> , and <code>consent</code> . The <code>prompt=login</code> claim forces the user to enter their credentials on that request, which negates single sign-on. The <code>prompt=None</code> claim is the opposite. This claim ensures that the user isn't presented with any interactive prompt at. If the request can't be completed silently via single sign-on, the Microsoft identity platform endpoint returns an error. The <code>prompt=consent</code> claim triggers the OAuth consent dialog after the user signs in. The dialog asks the user to grant permissions to the app. |
| <code>login_hint</code> | Optional | You can use this parameter to pre-fill the username and email address field of the sign-in page for the user, if you know the username ahead of time. Often, apps use this parameter during reauthentication, after already extracting the username from an earlier sign-in by using the <code>preferred_username</code> claim. |
| <code>domain_hint</code> | Optional | The realm of the user in a federated directory. This skips the email-based discovery process that the user goes through on the sign-in page, for a slightly more streamlined user experience. For tenants that are federated through an on-premises directory like AD FS, this often results in a seamless sign-in because of the existing login session. |

At this point, the user is prompted to enter their credentials and complete the authentication. The Microsoft identity platform endpoint verifies that the user has consented to the permissions indicated in the `scope` query parameter. If the user hasn't consented to any of those permissions, the Microsoft identity platform endpoint prompts the user to consent to the required permissions. You can read more about [permissions, consent, and multi-tenant apps](#).

After the user authenticates and grants consent, the Microsoft identity platform endpoint returns a response to your app at the indicated redirect URI by using the method specified in the `response_mode` parameter.

Successful response

A successful response when you use `response_mode=form_post` looks like this:

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

id_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik1uQ19WVmNB...&state=12345
```

| PARAMETER | DESCRIPTION |
|-----------------------|--|
| <code>id_token</code> | The ID token that the app requested. You can use the <code>id_token</code> parameter to verify the user's identity and begin a session with the user. For more information about ID tokens and their contents, see the id_tokens reference . |
| <code>state</code> | If a <code>state</code> 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 might also be sent to the redirect URI so that the app can handle them. An error response looks like this:

```
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 |
|--------------------------------|---|
| <code>error</code> | An error code string that you can use to classify types of errors that occur, and to react to errors. |
| <code>error_description</code> | A specific error message that can help you identify the root cause of an authentication error. |

Error codes for authorization endpoint errors

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

| ERROR CODE | DESCRIPTION | CLIENT ACTION |
|--|---|---|
| <code>invalid_request</code> | Protocol error, such as a missing, required parameter. | Fix and resubmit the request. This is a development error that typically is caught during initial testing. |
| <code>unauthorized_client</code> | The client application can't request an authorization code. | This usually occurs when the client application isn't registered in Azure AD or isn't added to the user's Azure AD tenant. The application can prompt the user with instructions to install the application and add it to Azure AD. |
| <code>access_denied</code> | The resource owner denied consent. | The client application can notify the user that it can't proceed unless the user consents. |
| <code>unsupported_response_type</code> | The authorization server does not support the response type in the request. | Fix and resubmit the request. This is a development error that typically is caught during initial testing. |

| ERROR CODE | DESCRIPTION | CLIENT ACTION |
|-------------------------|--|---|
| 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 because of 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 because of a temporary condition. |
| invalid_resource | The target resource is invalid because either it does not exist, Azure AD can't find it, or it isn't correctly configured. | This indicates that the resource, if it exists, hasn't been configured in the tenant. The application can prompt the user with instructions for installing the application and adding it to Azure AD. |

Validate the ID token

Just receiving an `id_token` isn't sufficient to authenticate the user; you must validate the `id_token`'s signature and verify the claims in the token per your app's requirements. The Microsoft identity platform endpoint uses [JSON Web Tokens \(JWTs\)](#) and public key cryptography to sign tokens and verify that they're 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 do the validation there. Once you've validated the signature of the `id_token`, there are a few claims you'll be required to verify. See the [id_token reference](#) for more information, including [Validating Tokens](#) and [Important Information About Signing Key Rollover](#). We recommend making use of a library for parsing and validating tokens - there is at least one available for most languages and platforms.

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 want to sign out the user from your app, it isn't sufficient to clear your app's cookies or otherwise end the user's session. You must also redirect the user to the Microsoft identity platform endpoint to sign out. If you don't do this, the user reauthenticates to your app without entering their credentials again, because they will have a valid single sign-in session with the Microsoft identity platform endpoint.

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

```
GET https://login.microsoftonline.com/common/oauth2/v2.0/logout?
post_logout_redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
```

| Parameter | Condition | Description |
|---------------------------------------|-------------|--|
| <code>post_logout_redirect_uri</code> | Recommended | The URL that the user is redirected to after successfully signing out. If the parameter isn't included, the user is shown a generic message that's generated by the Microsoft identity platform endpoint. This URL must match one of the redirect URLs registered for your application in the app registration portal. |

Single sign-out

When you redirect the user to the `end_session_endpoint`, the Microsoft identity platform endpoint clears the user's session from the browser. However, the user may still be signed in to other applications that use Microsoft accounts for authentication. To enable those applications to sign the user out simultaneously, the Microsoft identity platform endpoint 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 app registration portal.

Protocol diagram: Access token acquisition

Many web apps need to not only sign the user in, but also to access a web service on behalf of the user by using OAuth. This scenario combines OpenID Connect for user authentication while simultaneously getting an authorization code that you can use to get access tokens if you are using the OAuth authorization code flow.

The full OpenID Connect sign-in and token acquisition flow looks similar to the next diagram. We describe each step in detail in the next sections of the article.

Get access tokens

To acquire access tokens, modify the sign-in request:

TIP

Click the following link to execute this request. After you sign in, your browser is redirected to <https://localhost/myapp/>, with an ID token and a code in the address bar. Note that this request uses `response_mode=fragment` for demonstration purposes only. We recommend that you use `response_mode=form_post`.

<https://login.microsoftonline.com/common/oauth2/v2.0/authorize...>

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

Successful response

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

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

id_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik1uQ19WVmNB...&code=AwABAAAAvPM1KaPlrEqdFSBzjqfTGBCmLdg
fSTLEMPGYuNHSUYBrq...&state=12345
```

| PARAMETER | DESCRIPTION |
|-----------------------|---|
| <code>id_token</code> | 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. You'll find more details about ID tokens and their contents in the id_tokens reference. |
| <code>code</code> | The authorization code that the app requested. The app can use the authorization code to request an access token for the target resource. An authorization code is short-lived. Typically, an authorization code expires in about 10 minutes. |
| <code>state</code> | 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 might also be sent to the redirect URI so that the app can handle them appropriately. An error response looks like this:

```
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 |
|--------------------------------|---|
| <code>error</code> | An error code string that you can use to classify types of errors that occur, and to react to errors. |
| <code>error_description</code> | A specific error message that can help you identify the root cause of an authentication error. |

For a description of possible error codes and recommended client responses, see [Error codes for authorization endpoint errors](#).

When you have 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](#). To get access tokens, follow the steps described in [OAuth code flow documentation](#).

Microsoft identity platform and Implicit grant flow

11/4/2019 • 11 minutes to read • [Edit Online](#)

Applies to:

- Microsoft identity platform endpoint

With the Microsoft identity platform endpoint, you can sign users into your single-page apps with both personal and work or school accounts from Microsoft. Single-page and other JavaScript apps that run primarily in a browser face a few interesting challenges when it comes to authentication:

- The security characteristics of these apps are significantly different from traditional server-based web applications.
- Many authorization servers and identity providers do not support CORS requests.
- Full page browser redirects away from the app become particularly invasive to the user experience.

For these applications (AngularJS, Ember.js, React.js, and so on), Microsoft identity platform supports the OAuth 2.0 Implicit Grant flow. The implicit flow is described in the [OAuth 2.0 Specification](#). Its primary benefit is that it allows the app to get tokens from Microsoft identity platform without performing a backend server credential exchange. This allows the app to sign in the user, maintain session, and get tokens to other web APIs all within the client JavaScript code. There are a few important security considerations to take into account when using the implicit flow specifically around [client](#) and [user impersonation](#).

If you want to use the implicit flow and Microsoft identity platform to add authentication to your JavaScript app, we recommend you use the open-source JavaScript library, [msal.js](#).

However, if you prefer not to use a library in your single-page app and send protocol messages yourself, follow the general steps below.

NOTE

Not all Azure Active Directory (Azure AD) scenarios and features are supported by the Microsoft identity platform endpoint. To determine if you should use the Microsoft identity platform endpoint, read about [Microsoft identity platform limitations](#).

Protocol diagram

The following diagram shows what the entire implicit sign-in flow looks like and the sections that follow describe each step in more detail.

Send the sign-in request

To initially sign the user into your app, you can send an [OpenID Connect](#) authentication request and get an `id_token` from the Microsoft identity platform endpoint.

IMPORTANT

To successfully request an ID token and/or an access token, the app registration in the [Azure portal - App registrations](#) page must have the corresponding implicit grant flow enabled, by selecting **ID tokens** and/or **access tokens** under the **Implicit grant** section. If it's not enabled, an `unsupported_response` error will be returned: **The provided value for the input parameter 'response_type' is not allowed for this client. Expected value is 'code'**

```
// Line breaks for legibility only
https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&response_type=id_token
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
&scope=openid
&response_mode=fragment
&state=12345
&nonce=678910
```

TIP

To test signing in using the implicit flow, click <https://login.microsoftonline.com/common/oauth2/v2.0/authorize...>. After signing in, your browser should be redirected to <https://localhost/myapp/> with an `id_token` in the address bar.

| PARAMETER | | DESCRIPTION |
|----------------------------|----------|---|
| <code>tenant</code> | required | The <code>{tenant}</code> value in the path of the request can be used to control who can sign into the application. The allowed values are <code>common</code> , <code>organizations</code> , <code>consumers</code> , and tenant identifiers. For more detail, see protocol basics . |
| <code>client_id</code> | required | The Application (client) ID that the Azure portal - App registrations page assigned to your app. |
| <code>response_type</code> | required | Must include <code>id_token</code> for OpenID Connect sign-in. It may also include the <code>response_type token</code> . Using <code>token</code> here will allow your app to receive an access token immediately from the authorize endpoint without having to make a second request to the authorize endpoint. If you use the <code>token</code> response_type, the <code>scope</code> parameter must contain a scope indicating which resource to issue the token for (for example, <code>user.read</code> on Microsoft Graph). |

| PARAMETER | | DESCRIPTION |
|----------------------------|-------------|---|
| <code>redirect_uri</code> | 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. |
| <code>scope</code> | required | A space-separated list of scopes . For OpenID Connect (id_tokens), it must include the scope <code>openid</code> , which translates to the "Sign you in" permission in the consent UI. Optionally you may also want to include the <code>email</code> and <code>profile</code> scopes for gaining access to additional user data. You may also include other scopes in this request for requesting consent to various resources, if an access token is requested. |
| <code>response_mode</code> | optional | Specifies the method that should be used to send the resulting token back to your app. Defaults to query for just an access token, but fragment if the request includes an id_token. |
| <code>state</code> | recommended | A value included in the request that will also be 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. |
| <code>nonce</code> | required | A value included in the request, generated by the app, that will be 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 that can be used to identify the origin of the request. Only required when an id_token is requested. |
| <code>prompt</code> | optional | Indicates the type of user interaction that is required. The only valid values at this time are 'login', 'none', 'select_account', and 'consent'. <code>prompt=login</code> will force the user to enter their credentials on that request, negating single-sign on. <code>prompt=none</code> is the opposite - it will ensure that the user isn't presented with any interactive prompt whatsoever. If the request can't be completed silently via single-sign on, the Microsoft identity platform endpoint will return error. <code>prompt=select_account</code> sends the user to an account picker where all of the accounts remembered in the session will appear. <code>prompt=consent</code> will trigger the OAuth consent dialog after the user signs in, asking the user to grant permissions to the app. |
| <code>login_hint</code> | 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 will use this parameter during re-authentication, having already extracted the username from a previous sign-in using the <code>preferred_username</code> claim. |
| <code>domain_hint</code> | optional | If included, it will skip the email-based discovery process that user goes through on the sign in page, leading to a slightly more streamlined user experience. This is commonly used for Line of Business apps that operate in a single tenant, where they will provide a domain name within a given tenant. This will forward the user to the federation provider for that tenant. Note that this will prevent guests from signing into this application. |

At this point, the user will be asked to enter their credentials and complete the authentication. The Microsoft identity platform endpoint will also ensure that the user has consented to the permissions indicated in the `scope` query parameter. If the user has consented to **none** of those permissions, it will ask the user to consent to the required permissions. For more info, see [permissions, consent, and multi-tenant apps](#).

Once the user authenticated and grants consent, the Microsoft identity platform endpoint will return a response to your app at the indicated `redirect_uri`, using the method specified in the `response_mode` parameter.

Successful response

A successful response using `response_mode=fragment` and `response_type=id_token+token` looks like the following (with line breaks for legibility):

```
GET https://localhost/myapp/#&token_type=Bearer&expires_in=3599&id_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZn10aEV1Q...&state=12345
```

| PARAMETER | DESCRIPTION |
|---------------------------|---|
| <code>access_token</code> | Included if <code>response_type</code> includes <code>token</code> . The access token that the app requested. The access token shouldn't be decoded or otherwise inspected, it should be treated as an opaque string. |
| <code>token_type</code> | Included if <code>response_type</code> includes <code>token</code> . Will always be <code>Bearer</code> . |
| <code>expires_in</code> | Included if <code>response_type</code> includes <code>token</code> . Indicates the number of seconds the token is valid, for caching purposes. |

| PARAMETER | DESCRIPTION |
|-----------------------|---|
| <code>scope</code> | Included if <code>response_type</code> includes <code>token</code> . Indicates the scope(s) for which the access_token will be valid. May not include all of the scopes requested, if they were not applicable to the user (in the case of Azure AD-only scopes being requested when a personal account is used to log in). |
| <code>id_token</code> | A signed JSON Web Token (JWT). The app can 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 shouldn't rely on them for any authorization or security boundaries. For more information about id_tokens, see the id_token reference . |
| <code>state</code> | 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:

```
GET https://localhost/myapp/#error=access_denied&error_description=the+user+canceled+the+authentication
```

| PARAMETER | DESCRIPTION |
|--------------------------------|---|
| <code>error</code> | An error code string that can be used to classify types of errors that occur, and can be used to react to errors. |
| <code>error_description</code> | A specific error message that can help a developer identify the root cause of an authentication error. |

Getting access tokens silently in the background

Now that you've signed the user into your single-page app, you can silently get access tokens for calling web APIs secured by Microsoft identity platform, such as the [Microsoft Graph](#). Even if you already received a token using the `token` `response_type`, you can use this method to acquire tokens to additional resources without having to redirect the user to sign in again.

In the normal OpenID Connect/OAuth flow, you would do this by making a request to the Microsoft identity platform `/token` endpoint. However, the Microsoft identity platform endpoint does not support CORS requests, so making AJAX calls to get and refresh tokens is out of the question. Instead, you can use the implicit flow in a hidden iframe to get new tokens for other web APIs:

```
// Line breaks for legibility only

https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&response_type=token
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
&scope=https%3A%2F%2Fgraph.microsoft.com%2Fuser.read
&response_mode=fragment
&state=12345
&nonce=678910
&prompt=none
&login_hint=myuser@mycompany.com
```

For details on the query parameters in the URL, see [send the sign in request](#).

TIP

Try copy & pasting the below request into a browser tab! (Don't forget to replace the `login_hint` values with the correct value for your user)

```
https://login.microsoftonline.com/common/oauth2/v2.0/authorize?client_id=6731de76-14a6-49ae-97bc-6eba6914391e&response_type=token&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F&scope=https%3A%2F%2Fgraph.microsoft.com%2Fuser.read&response_mode=fragment&state=12345&nonce=678910&prompt=(your-username)
```

Thanks to the `prompt=None` parameter, this request will either succeed or fail immediately and return to your application. A successful response will be sent to your app at the indicated `redirect_uri`, using the method specified in the `response_mode` parameter.

Successful response

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

```
GET https://localhost/myapp/
access_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVZ2ZEstZnl0aEV1Q...
&state=12345
&token_type=Bearer
&expires_in=3599
&scope=https%3A%2F%2Fgraph.windows.net%2Fdirectory.read
```

| PARAMETER | DESCRIPTION |
|---------------------------|---|
| <code>access_token</code> | Included if <code>response_type</code> includes <code>token</code> . The access token that the app requested, in this case for the Microsoft Graph. The access token shouldn't be decoded or otherwise inspected, it should be treated as an opaque string. |
| <code>token_type</code> | Will always be <code>Bearer</code> . |
| <code>expires_in</code> | Indicates the number of seconds the token is valid, for caching purposes. |
| <code>scope</code> | Indicates the scope(s) for which the access_token will be valid. May not include all of the scopes requested, if they were not applicable to the user (in the case of Azure AD-only scopes being requested when a personal account is used to log in). |

| PARAMETER | DESCRIPTION |
|-----------------------|---|
| <code>id_token</code> | A signed JSON Web Token (JWT). Included if <code>response_type</code> includes <code>id_token</code> . The app can 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 shouldn't rely on them for any authorization or security boundaries. For more information about id_tokens, see the id_token reference .
Note: Only provided if <code>openid</code> scope was requested. |
| <code>state</code> | 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. In the case of `prompt=none`, an expected error will be:

```
GET https://localhost/myapp/
error=user_authentication_required
&error_description=the+request+could+not+be+completed+silently
```

| PARAMETER | DESCRIPTION |
|--------------------------------|---|
| <code>error</code> | An error code string that can be used to classify types of errors that occur, and can be used to react to errors. |
| <code>error_description</code> | A specific error message that can help a developer identify the root cause of an authentication error. |

If you receive this error in the iframe request, the user must interactively sign in again to retrieve a new token. You can choose to handle this case in whatever way makes sense for your application.

Refreshing tokens

The implicit grant does not provide refresh tokens. Both `id_token`s and `access_token`s will expire after a short period of time, so your app must be prepared to refresh these tokens periodically. To refresh either type of token, you can perform the same hidden iframe request from above using the `prompt=none` parameter to control the identity platform's behavior. If you want to receive a new `id_token`, be sure to use `id_token` in the `response_type` and `scope=openid`, as well as a `nonce` parameter.

Send a sign out request

The OpenID Connect `end_session_endpoint` allows your app to send a request to the Microsoft identity platform endpoint to end a user's session and clear cookies set by the Microsoft identity platform endpoint. To fully sign a user out of a web application, your app should end its own session with the user (usually by clearing a token cache or dropping cookies), and then redirect the browser to:

```
https://login.microsoftonline.com/{tenant}/oauth2/v2.0/logout?post_logout_redirect_uri=https://localhost/myapp/
```

| PARAMETER | | DESCRIPTION |
|---------------------------------------|-------------|--|
| <code>tenant</code> | required | The <code>{tenant}</code> value in the path of the request can be used to control who can sign into the application. The allowed values are <code>common</code> , <code>organizations</code> , <code>consumers</code> , and tenant identifiers. For more detail, see protocol basics . |
| <code>post_logout_redirect_uri</code> | recommended | The URL that the user should be returned to after logout completes. This value must match one of the redirect URLs registered for the application. If not included, the user will be shown a generic message by the Microsoft identity platform endpoint. |

Next steps

- Go over the [MSAL JS samples](#) to get started coding.

Microsoft identity platform and OAuth 2.0 authorization code flow

8/7/2019 • 16 minutes to read • [Edit Online](#)

Applies to:

- Microsoft identity platform endpoint

The OAuth 2.0 authorization code grant can be used in apps that are installed on a device to gain access to protected resources, such as web APIs. Using the Microsoft identity platform implementation of OAuth 2.0, you can add sign in and API access to your mobile and desktop apps. This guide is language-independent, and describes how to send and receive HTTP messages without using any of the [Azure open-source authentication libraries](#).

NOTE

Not all Azure Active Directory scenarios & features are supported by the Microsoft identity platform endpoint. To determine if you should use the Microsoft identity platform endpoint, read about [Microsoft identity platform limitations](#).

The OAuth 2.0 authorization code flow is described in [section 4.1 of the OAuth 2.0 specification](#). It's used to perform authentication and authorization in the majority of app types, including [web apps](#) and [natively installed apps](#). The flow enables apps to securely acquire access_tokens that can be used to access resources secured by the Microsoft identity platform endpoint.

Protocol diagram

At a high level, the entire authentication flow for a native/mobile application looks a bit like this:

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:

```
// Line breaks for legibility only  
  
https://login.microsoftonline.com/{tenant}/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=openid%20offline_access%20https%3A%2F%2Fgraph.microsoft.com%2Fuser.read  
&state=12345
```

TIP

Click the link below to execute this request! After signing in, your browser should be redirected to `https://localhost/myapp/` with a `code` in the address bar: <https://login.microsoftonline.com/common/oauth2/v2.0/authorize...>

| PARAMETER | REQUIRED/OPTIONAL | DESCRIPTION |
|---------------------|-------------------|--|
| <code>tenant</code> | required | The <code>{tenant}</code> value in the path of the request can be used to control who can sign into the application. The allowed values are <code>common</code> , <code>organizations</code> , <code>consumers</code> , and tenant identifiers. For more detail, see protocol basics . |

| PARAMETER | REQUIRED/OPTIONAL | DESCRIPTION |
|----------------------------|-------------------|--|
| <code>client_id</code> | required | The Application (client) ID that the Azure portal – App registrations experience assigned to your app. |
| <code>response_type</code> | required | Must include <code>code</code> for the authorization code flow. |
| <code>redirect_uri</code> | required | 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
<code>https://login.microsoftonline.com/common/oauth2/</code>
. |
| <code>scope</code> | required | A space-separated list of scopes that you want the user to consent to. For the <code>/authorize</code> leg of the request, this can cover multiple resources, allowing your app to get consent for multiple web APIs you want to call. |
| <code>response_mode</code> | recommended | Specifies the method that should be used to send the resulting token back to your app. Can be one of the following: <ul style="list-style-type: none"> - <code>query</code> - <code>fragment</code> - <code>form_post</code>
<code>query</code> provides the code as a query string parameter on your redirect URI. If you're requesting an ID token using the implicit flow, you can't use <code>query</code> as specified in the OpenID spec . If you're requesting just the code, you can use <code>query</code> , <code>fragment</code> , or <code>form_post</code> . <code>form_post</code> executes a POST containing the code to your redirect URI. For more info, see OpenID Connect protocol . |
| <code>state</code> | recommended | A value included in the request that will also be 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 value can also encode information about the user's state in the app before the authentication request occurred, such as the page or view they were on. |
| <code>prompt</code> | optional | Indicates the type of user interaction that is required. The only valid values at this time are <code>login</code> , <code>none</code> , and <code>consent</code> . <ul style="list-style-type: none"> - <code>prompt=login</code> will force the user to enter their credentials on that request, negating single-sign on. - <code>prompt=none</code> is the opposite - it will ensure that the user isn't presented with any interactive prompt whatsoever. If the request can't be completed silently via single-sign on, the Microsoft identity platform endpoint will return an <code>interaction_required</code> error. - <code>prompt=consent</code> will trigger the OAuth consent dialog after the user signs in, asking the user to grant permissions to the app. |

| PARAMETER | REQUIRED/OPTIONAL | DESCRIPTION |
|------------------------------------|-------------------|--|
| <code>login_hint</code> | 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 will use this parameter during re-authentication, having already extracted the username from a previous sign-in using the <code>preferred_username</code> claim. |
| <code>domain_hint</code> | optional | Can be one of <code>consumers</code> or <code>organizations</code> . If included, it will skip the email-based discovery process that user goes through on the sign-in page, leading to a slightly more streamlined user experience. Often apps will use this parameter during re-authentication, by extracting the <code>tid</code> from a previous sign-in. If the <code>tid</code> claim value is <code>9188040d-6c67-4c5b-b112-36a304b66dad</code> , you should use <code>domain_hint=consumers</code> . Otherwise, use <code>domain_hint=organizations</code> . |
| <code>code_challenge_method</code> | optional | The method used to encode the <code>code_verifier</code> for the <code>code_challenge</code> parameter. Can be one of the following values:
- <code>plain</code>
- <code>S256</code>

If excluded, <code>code_challenge</code> is assumed to be plaintext if <code>code_challenge</code> is included. Microsoft identity platform supports both <code>plain</code> and <code>s256</code> . For more information, see the PKCE RFC . |
| <code>code_challenge</code> | optional | Used to secure authorization code grants via Proof Key for Code Exchange (PKCE) from a native client. Required if <code>code_challenge_method</code> is included. For more information, see the PKCE RFC . |

At this point, the user will be asked to enter their credentials and complete the authentication. The Microsoft identity platform endpoint will also ensure that the user has consented to the permissions indicated in the `scope` query parameter. If the user has not consented to any of those permissions, it will ask the user to consent to the required permissions. Details of [permissions, consent, and multi-tenant apps are provided here](#).

Once the user authenticates and grants consent, the Microsoft identity platform endpoint will return a response to your app at the indicated `redirect_uri`, using the method specified in the `response_mode` parameter.

Successful response

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

```
GET https://login.microsoftonline.com/common/oauth2/nativeclient?
code=AwABAAAAvPM1KaPlrEqdFSBzjqfTGBcmLdgfSTLEMPGYuNHSUYBrq...
&state=12345
```

| PARAMETER | DESCRIPTION |
|-------------------|--|
| <code>code</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, typically they expire after about 10 minutes. |

| PARAMETER | DESCRIPTION |
|--------------------|---|
| <code>state</code> | 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:

```
GET https://login.microsoftonline.com/common/oauth2/nativeclient?
error=access_denied
&error_description=the+user+canceled+the+authentication
```

| PARAMETER | DESCRIPTION |
|--------------------------------|---|
| <code>error</code> | An error code string that can be used to classify types of errors that occur, and can be used to react to errors. |
| <code>error_description</code> | 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 |
|--|---|---|
| <code>invalid_request</code> | Protocol error, such as a missing required parameter. | Fix and resubmit the request. This is a development error typically caught during initial testing. |
| <code>unauthorized_client</code> | The client application isn't permitted to request an authorization code. | This error usually occurs when the client application isn't registered in Azure AD or isn't 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. |
| <code>access_denied</code> | Resource owner denied consent | The client application can notify the user that it can't proceed unless the user consents. |
| <code>unsupported_response_type</code> | The authorization server does not support the response type in the request. | Fix and resubmit the request. This is a development error typically caught during initial testing. |
| <code>server_error</code> | 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 to a temporary error. |
| <code>temporarily_unavailable</code> | 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 because of a temporary condition. |
| <code>invalid_resource</code> | The target resource is invalid because it does not exist, Azure AD can't find it, or it's not correctly configured. | This error 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. |

| ERROR CODE | DESCRIPTION | CLIENT ACTION |
|-----------------------------------|--|--|
| <code>login_required</code> | Too many or no users found | The client requested silent authentication (<code>prompt=none</code>), but a single user could not be found. This may mean there are multiple users active in the session, or no users. This takes into account the tenant chosen (for example, if there are two Azure AD accounts active and one Microsoft account, and <code>consumers</code> is chosen, silent authentication will work). |
| <code>interaction_required</code> | The request requires user interaction. | An additional authentication step or consent is required. Retry the request without <code>prompt=none</code> . |

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. Do this by sending a `POST` request to the `/token` endpoint:

```
// Line breaks for legibility only

POST /{tenant}/oauth2/v2.0/token HTTP/1.1
Host: https://login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&scope=https%3A%2F%2Fgraph.microsoft.com%2Fuser.read
&code=OAAABAAAiL9KnZz27UubvWPbm0gLWQJVzCTE9UkP3p5x1aXxUjq3n8b2JRLk40xVXr...
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
&grant_type=authorization_code
&client_secret=JqQX2PNo9bpM0uEihUPzyrh    // NOTE: Only required for web apps
```

TIP

Try executing this request in Postman! (Don't forget to replace the `code`) [▶ Run in Postman](#)

| PARAMETER | REQUIRED/OPTIONAL | DESCRIPTION |
|-------------------------|-------------------|--|
| <code>tenant</code> | required | The <code>{tenant}</code> value in the path of the request can be used to control who can sign into the application. The allowed values are <code>common</code> , <code>organizations</code> , <code>consumers</code> , and tenant identifiers. For more detail, see protocol basics . |
| <code>client_id</code> | required | The Application (client) ID that the Azure portal – App registrations page assigned to your app. |
| <code>grant_type</code> | required | Must be <code>authorization_code</code> for the authorization code flow. |
| <code>scope</code> | required | A space-separated list of scopes. The scopes requested in this leg must be equivalent to or a subset of the scopes requested in the first leg. The scopes must all be from a single resource, along with OIDC scopes (<code>profile</code> , <code>openid</code> , <code>email</code>). For a more detailed explanation of scopes, refer to permissions, consent, and scopes . |
| <code>code</code> | required | The <code>authorization_code</code> that you acquired in the first leg of the flow. |

| PARAMETER | REQUIRED/OPTIONAL | DESCRIPTION |
|----------------------------|-----------------------|--|
| <code>redirect_uri</code> | required | The same redirect_uri value that was used to acquire the authorization_code. |
| <code>client_secret</code> | required for web apps | The application secret that you created in the app registration portal for your app. You shouldn't use the application secret in a native app because client_secrets can't be reliably stored on devices. It's required for web apps and web APIs, which have the ability to store the client_secret securely on the server side. The client secret must be URL-encoded before being sent. |
| <code>code_verifier</code> | optional | The same code_verifier that was used to obtain the authorization_code. Required if PKCE was used in the authorization code grant request. For more information, see the PKCE RFC . |

Successful response

A successful token response will look like:

```
{
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZn10aEV1Q...",
  "token_type": "Bearer",
  "expires_in": 3599,
  "scope": "https%3A%2F%2Fgraph.microsoft.com%2Fuser.read",
  "refresh_token": "AwABAAAavPM1KaPlrEqdFSBzjqfTGAMxZGUTdM0t4B4...",
  "id_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhdWQiOiIyZDRkMTFhMi1mODE0LTQ2YTctOD...",
}
```

| PARAMETER | DESCRIPTION |
|----------------------------|--|
| <code>access_token</code> | The requested access token. The app can use this token to authenticate to the secured resource, such as a web API. |
| <code>token_type</code> | Indicates the token type value. The only type that Azure AD supports is Bearer |
| <code>expires_in</code> | How long the access token is valid (in seconds). |
| <code>scope</code> | The scopes that the access_token is valid for. |
| <code>refresh_token</code> | An OAuth 2.0 refresh token. The app can use this token 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. For more detail on refreshing an access token, refer to the section below .
Note: Only provided if <code>offline_access</code> scope was requested. |
| <code>id_token</code> | A JSON Web Token (JWT). The app can 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. For more information about id_tokens, see the id_token reference .
Note: Only provided if <code>openid</code> scope was requested. |

Error response

Error responses will look like:

```
{
  "error": "invalid_scope",
  "error_description": "AADSTS70011: The provided value for the input parameter 'scope' is not valid. The scope https://foo.microsoft.com/mail.read is not valid.\r\nTrace ID: 255d1aef-8c98-452f-ac51-23d051240864\r\nCorrelation ID: fb3d2015-bc17-4bb9-bb85-30c5cf1aaaa7\r\nTimestamp: 2016-01-09 02:02:12Z",
  "error_codes": [
    70011
  ],
  "timestamp": "2016-01-09 02:02:12Z",
  "trace_id": "255d1aef-8c98-452f-ac51-23d051240864",
  "correlation_id": "fb3d2015-bc17-4bb9-bb85-30c5cf1aaaa7"
}
```

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

Error codes for token endpoint errors

| ERROR CODE | DESCRIPTION | CLIENT ACTION |
|-------------------------------------|---|---|
| <code>invalid_request</code> | Protocol error, such as a missing required parameter. | Fix and resubmit the request |
| <code>invalid_grant</code> | The authorization code or PKCE code verifier is invalid or has expired. | Try a new request to the <code>/authorize</code> endpoint and verify that the <code>code_verifier</code> parameter was correct. |
| <code>unauthorized_client</code> | The authenticated client isn't authorized to use this authorization grant type. | This usually occurs when the client application isn't registered in Azure AD or isn't 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. |
| <code>invalid_client</code> | Client authentication failed. | The client credentials aren't valid. To fix, the application administrator updates the credentials. |
| <code>unsupported_grant_type</code> | 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. |
| <code>invalid_resource</code> | The target resource is invalid because it does not exist, Azure AD can't find it, or it's 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. |
| <code>interaction_required</code> | The request requires user interaction. For example, an additional authentication step is required. | Retry the request with the same resource. |

| ERROR CODE | DESCRIPTION | CLIENT ACTION |
|-------------------------|---|--|
| 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 because of a temporary condition. |

Use the access token

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:

TIP

Execute this request in Postman! (Replace the `Authorization` header first) [▶ Run in Postman](#)

```
GET /v1.0/me/messages
Host: https://graph.microsoft.com
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZnl0aEV1Q...
```

Refresh the access token

Access_tokens are short lived, and you must refresh them after they expire to continue accessing resources. You can do so by submitting another `POST` request to the `/token` endpoint, this time providing the `refresh_token` instead of the `code`. Refresh tokens are valid for all permissions that your client has already received consent for - thus, a refresh token issued on a request for `scope=mail.read` can be used to request a new access token for `scope=api://contoso.com/api/UseResource`.

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.

Although refresh tokens aren't revoked when used to acquire new access tokens, you are expected to discard the old refresh token. The [OAuth 2.0 spec](#) says: "The authorization server MAY issue a new refresh token, in which case the client MUST discard the old refresh token and replace it with the new refresh token. The authorization server MAY revoke the old refresh token after issuing a new refresh token to the client."

```
// Line breaks for legibility only

POST /{tenant}/oauth2/v2.0/token HTTP/1.1
Host: https://login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&scope=https%3A%2Fgraph.microsoft.com%2Fuser.read
&refresh_token=0AAABAAAiL9Kn2Z27UubvWFPbm0gLWQJVzCTE9UkP3pSx1aXxUjq...
&grant_type=refresh_token
&client_secret=JqQX2PNo9bpM0uEihUPzyrh // NOTE: Only required for web apps
```

TIP

Try executing this request in Postman! (Don't forget to replace the `refresh_token`) [▶ Run in Postman](#)

| PARAMETER | DESCRIPTION |
|---------------------|--|
| <code>tenant</code> | required
The <code>{tenant}</code> value in the path of the request can be used to control who can sign into the application. The allowed values are <code>common</code> , <code>organizations</code> , <code>consumers</code> , and tenant identifiers. For more detail, see protocol basics . |

| PARAMETER | | DESCRIPTION |
|----------------------------|-----------------------|---|
| <code>client_id</code> | required | The Application (client) ID that the Azure portal – App registrations experience assigned to your app. |
| <code>grant_type</code> | required | Must be <code>refresh_token</code> for this leg of the authorization code flow. |
| <code>scope</code> | required | A space-separated list of scopes. The scopes requested in this leg must be equivalent to or a subset of the scopes requested in the original authorization_code request leg. If the scopes specified in this request span multiple resource server, then the Microsoft identity platform endpoint will return a token for the resource specified in the first scope. For a more detailed explanation of scopes, refer to permissions, consent, and scopes . |
| <code>refresh_token</code> | required | The refresh_token that you acquired in the second leg of the flow. |
| <code>client_secret</code> | required for web apps | The application secret that you created in the app registration portal for your app. It should not be used in a native app, because client_secrets can't be reliably stored on devices. It's required for web apps and web APIs, which have the ability to store the client_secret securely on the server side. |

Successful response

A successful token response will look like:

```
{
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZZEstZn10aEV1Q...",
  "token_type": "Bearer",
  "expires_in": 3599,
  "scope": "https%3A%2F%2Fgraph.microsoft.com%2Fuser.read",
  "refresh_token": "AwABAAAavPM1KaP1rEqdFSBzjqFTGAMxZGUTdM0t4B4...",
  "id_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhdWQiOiIyZDRkMTFhMi1mODE0LTQ2YTctOD....",
}
```

| PARAMETER | DESCRIPTION |
|----------------------------|---|
| <code>access_token</code> | The requested access token. The app can use this token to authenticate to the secured resource, such as a web API. |
| <code>token_type</code> | Indicates the token type value. The only type that Azure AD supports is Bearer |
| <code>expires_in</code> | How long the access token is valid (in seconds). |
| <code>scope</code> | The scopes that the access_token is valid for. |
| <code>refresh_token</code> | A new OAuth 2.0 refresh token. You should replace the old refresh token with this newly acquired refresh token to ensure your refresh tokens remain valid for as long as possible.
Note: Only provided if <code>offline_access</code> scope was requested. |
| <code>id_token</code> | An unsigned JSON Web Token (JWT). The app can 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. For more information about id_tokens, see the id_token reference .
Note: Only provided if <code>openid</code> scope was requested. |

Error response

```
{  
    "error": "invalid_scope",  
    "error_description": "AADSTS70011: The provided value for the input parameter 'scope' is not valid. The scope  
https://foo.microsoft.com/mail.read is not valid.\r\nTrace ID: 255d1aef-8c98-452f-ac51-23d051240864\r\nCorrelation ID:  
fb3d2015-bc17-4bb9-bb85-30c5cf1aaaa7\r\nTimestamp: 2016-01-09 02:02:12Z",  
    "error_codes": [  
        70011  
    ],  
    "timestamp": "2016-01-09 02:02:12Z",  
    "trace_id": "255d1aef-8c98-452f-ac51-23d051240864",  
    "correlation_id": "fb3d2015-bc17-4bb9-bb85-30c5cf1aaaa7"  
}
```

| PARAMETER | DESCRIPTION |
|--------------------------------|---|
| <code>error</code> | An error code string that can be used to classify types of errors that occur, and can be used to react to errors. |
| <code>error_description</code> | A specific error message that can help a developer identify the root cause of an authentication error. |
| <code>error_codes</code> | A list of STS-specific error codes that can help in diagnostics. |
| <code>timestamp</code> | The time at which the error occurred. |
| <code>trace_id</code> | A unique identifier for the request that can help in diagnostics. |
| <code>correlation_id</code> | 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](#).

Microsoft identity platform and OAuth 2.0 On-Behalf-Of flow

7/1/2019 • 9 minutes to read • [Edit Online](#)

Applies to:

- Microsoft identity platform endpoint

The OAuth 2.0 On-Behalf-Of flow (OBO) serves the use case where an application invokes a service/web API, which in turn needs to call another service/web API. The idea is to propagate the delegated user identity and permissions through the request chain. For the middle-tier service to make authenticated requests to the downstream service, it needs to secure an access token from the Microsoft identity platform, on behalf of the user.

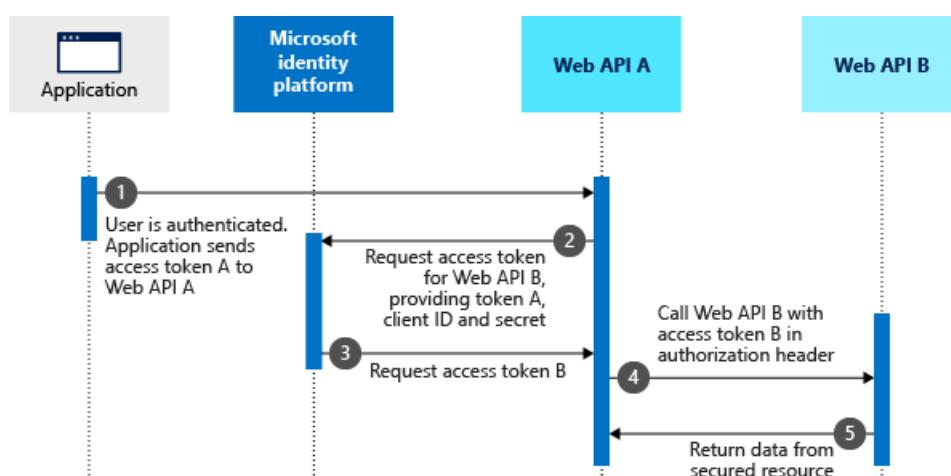
NOTE

- The Microsoft identity platform endpoint doesn't support all scenarios and features. To determine whether you should use the Microsoft identity platform endpoint, read about [Microsoft identity platform limitations](#). Specifically, known client applications aren't supported for apps with Microsoft account (MSA) and Azure AD audiences. Thus, a common consent pattern for OBO will not work for clients that sign in both personal and work or school accounts. To learn more about how to handle this step of the flow, see [Gaining consent for the middle-tier application](#).
- As of May 2018, some implicit-flow derived `id_token` can't be used for OBO flow. Single-page apps (SPAs) should pass an **access** token to a middle-tier confidential client to perform OBO flows instead. For more info about which clients can perform OBO calls, see [limitations](#).

Protocol diagram

Assume that the user has been authenticated on an application using the [OAuth 2.0 authorization code grant flow](#). At this point, the application has an access token for API A (token A) with the user's claims and consent to access the middle-tier web API (API A). Now, API A needs to make an authenticated request to the downstream web API (API B).

The steps that follow constitute the OBO flow and are explained with the help of the following diagram.



1. The client application makes a request to API A with token A (with an `aud` claim of API A).
2. API A authenticates to the Microsoft identity platform token issuance endpoint and requests a token to access API B.
3. The Microsoft identity platform token issuance endpoint validates API A's credentials with token A and issues the access token for API B (token B).
4. Token B is set in the authorization header of the request to API B.
5. Data from the secured resource is returned by API B.

NOTE

In this scenario, the middle-tier service has no user interaction to obtain the user's consent to access the downstream API. Therefore, the option to grant access to the downstream API is presented upfront as a part of the consent step during authentication. To learn how to set this up for your app, see [Gaining consent for the middle-tier application](#).

Service-to-service access token request

To request an access token, make an HTTP POST to the tenant-specific Microsoft identity platform token endpoint with the following parameters.

```
https://login.microsoftonline.com/<tenant>/oauth2/v2.0/token
```

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 |
|----------------------------------|----------|--|
| <code>grant_type</code> | Required | The type of token request. For a request using a JWT, the value must be <code>urn:ietf:params:oauth:grant-type:jwt-bearer</code> |
| <code>client_id</code> | Required | The application (client) ID that the Azure portal - App registrations page has assigned to your app. |
| <code>client_secret</code> | Required | The client secret that you generated for your app in the Azure portal - App registrations page. |
| <code>assertion</code> | Required | The value of the token used in the request. |
| <code>scope</code> | Required | A space separated list of scopes for the token request. For more information, see scopes . |
| <code>requested_token_use</code> | Required | Specifies how the request should be processed. In the OBO flow, the value must be set to <code>on_behalf_of</code> . |

Example

The following HTTP POST requests an access token and refresh token with `user.read` scope for the <https://graph.microsoft.com> web API.

```
//line breaks for legibility only

POST /oauth2/v2.0/token HTTP/1.1
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn:ietf:params:oauth:grant-type:jwt-bearer
&client_id=2846f71b-a7a4-4987-bab3-760035b2f389
&client_secret=BYyVnAt56JpLwUcyo47XODd
&assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6InowMzl6ZHNGdWl6cEJmQ1ZLMVRuMjVRSF1PMCJ9.eyJhdWQiOiIy0DQ2ZjcxYi1hN2E0LTQ50DctYmFiMy03NjAwMzViMmYzODkilCJpc3Mi0iJodHRwczovL2xvZ2luLm1pY3Jvc29mdG9ubGluZS5jb20vNzJmOTg4YmYtODZmMS00MWFMlTkxYWItMmQ3Y2QwMTFKYjQ3L3YyLjAiLCJpYXQi0jE00TM5Mja5MTYsIm5iZii6MTQ5MzkyMDkxNiwiZXhwIjoxNDKzOTI0ODE2LCJhaW8iOiJBU1FBM184REFBQUFnZm8vNk9CR0NaaFV2NjJ6MFYSEZKR0VVYUIwRULIV3NhcgdundMMnVrPSIsIm5hbWUi0iJOYXZ5YSBDYW51bWFsbGEiLCJvaWQiOiJkNWU5NzljNy0zzDjkLTQyYWYtOGYZMC03MjdkZDRjMmQzODMlCJwcmVmZXJyZWRfdXNlc5hbWUi0iJuYWNNhbNvtYUBtaWNyb3NvZnQuY29tIiwic3ViIjoiz1Q5a1FMN2hXRUpUUGg10WJlx115dVZNRFOTEdiREJFWFRhbEqzU3FZYyIsInRpZC16IjcyZjk40GJmlTg2ZjEtNDfhZi05MWFiltJkN2NkMDEzZGI0NyIsInV0aSI6IjN5U3F4UHJweUPd0zsTWFFMU1PQEiLCJ2ZXIi0iIyLjAifQ.TPPJSvpNCSCyUeIiKQoLMixN1-M-Y5U0QxtxVkepepyoWNG0i49YFAJC6AddCs5nJxr6f-ozIRuaiPzy29yRU0dSz_8KqG42luCyC1c951HyeDgqUJSz91Ku150D9kP5B9-2R-jgCerD_VVuxXUdkuPFE13VEADC_1qkGBiIg0AyLLbz7DTMp5DvmbC09DhrQqiouHQGFSk2TPmksgHm3-b3RgeNM1rJmpLThis2ZWBEIPx662pxL6NJdmV08cPVicGX4KkFo54Z3rfwiYg4YssiUc4w-w3NJUBQhnzfT14_Mtq2d7cvlul9uDzras091vFy32tWkrpa970UvdVfQ
&scope=https://graph.microsoft.com/user.read+offline_access
&requested_token_use=on_behalf_of
```

Second case: Access token request with a certificate

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

| PARAMETER | DESCRIPTION |
|------------------------------------|--|
| <code>grant_type</code> | Required
The type of the token request. For a request using a JWT, the value must be <code>urn:ietf:params:oauth:grant-type:jwt-bearer</code> |
| <code>client_id</code> | Required
The application (client) ID that the Azure portal - App registrations page has assigned to your app. |
| <code>client_assertion_type</code> | Required
The value must be <code>urn:ietf:params:oauth:client-assertion-type:jwt-bearer</code> |
| <code>client_assertion</code> | Required
An assertion (a JSON web token) that you need to create and sign with the certificate you registered as credentials for your application. To learn how to register your certificate and the format of the assertion, see certificate credentials . |
| <code>assertion</code> | Required
The value of the token used in the request. |

| PARAMETER | | DESCRIPTION |
|----------------------------------|----------|--|
| <code>requested_token_use</code> | Required | Specifies how the request should be processed. In the OBO flow, the value must be set to <code>on_behalf_of</code> . |
| <code>scope</code> | Required | A space-separated list of scopes for the token request. For more information, see scopes . |

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 with `user.read` scope for the <https://graph.microsoft.com> web API with a certificate.

```
// line breaks for legibility only

POST /oauth2/v2.0/token HTTP/1.1
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Ajwt-bearer
&client_id=625391af-c675-43e5-8e44-edd3e30ceb15
&client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJSUzIiNiIsIng1dCI6Imd4OHRHeXN5amNScUtqR1BuZDdSRnd2d1pJMCJ9.eyJ{a lot of
characters here}M8U3bSUKKJDEg
&assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzIiNiIsIng1dCI6InowMz16ZHNGdWl6cEJmQ1ZLMVRuMjVRSF1PMCIisImtpZCI6InowMz
16ZHNGdWl6cEJmQ1ZLMVRuMjVRSF1PMCI9.eyJhdWQiOiJodHRwczovL2Rkb2JhbGlhb91dGxvb2sub25taWNyb3NvZnQuY29tLzE5MjNmOD
YyLWU2ZGMtNDHfMy04MRhLTgwMmJhZTAwYWY2ZCIsImlzcyI6Imh0dBz0i8vc3RzLndpbmRvd3MubmV0LzI2MDM5Y2N1LTQ40WQtNDAwMi0
4MjkzLTViMGM1MTM0ZWfjYi8iLCJpYXQiOjE0OTM0MjMxNTIsIm5iZiI6MTQ5MzQyMzE1MiwiZXhwIjoxNDkzNDY2NjUyLCJhY3IiOiIxIiwi
YwlvIjoiWTJaZ1lCRFF2aT1VZEc0LzM0L3dpQndqbjhYeVp4YmR1TFhmVE1QeG8yY1N2elgreHBVQSIisImFtcIi6WyJwd2QiXSwiYXBwaWQio
iJiMzE1MDA3OS03YmViLTQxN2YtYTA2Ys0zZmrjNzhjMzI1NDUiLCJhcHBpZGFjciI6IjAiLCJ1X2V4cCI6MzAyNDAwLCJmYW1pbHfbmFtZS
I6I1Rlc3QiLCJnaXZ1b19uYw1lIjoiTmF2eWEiLCJpcGFkZHii0iIxNjcuMjIwljEuMTc3IiwibmFtZSI6Ik5hdnlhIFRlc3QiLCJvaWQioII
xY2Q0YmNhYy1i0DA4LTQyM2EtOWUyZi04MjdmYmIxYmI3Mzk1LCJwbGF0ZiI6IjMiLCJzY3Ai0iJ1c2Vx2ltcGVyc29uYXRpb24iLCJzdWIi
OjJEVxPbkdKMDJIUk0zRW5pbDFxdjZCakxTNU1lQy0tQ2ZpbzRxS1MzNEc4IiwidGlkIjoiMjYwMz1jY2UtNDg5ZC00MDAyLTgyOTMtNWlwy
zUxmzR1YWNiIiwidw5pcXV1X25hbWUi0iJuYXZ5YUBkZG9iYwpxpYw5vdXrsb29rLm9ubWljqcm9zb2Z0LmNvbSISInVwbiI6Im5hdnlhQRkb2
JhbGlhb91dGxvb2sub25taWNyb3NvZnQuY29tIiwidmVyIjoiMS4wIn0.R-Ke-
X071K0r5uLwx8g5CrcPAwRln5SccJCFjeJu6IUqpqcjWcDzeDdNOySiVPDU_ZU5knJmzRCF8fcjFtPsaA4R7vdIEbDuOur15FXsvE8FvVSjP-
490H6hBYqoSAslN3Mfb06Z8YfcIY4tSOB2I6ahQ_x4ZWFwg1C3w5mK-
_4iX81bqi95eV4RUKeFuHhQDXTwWhrSgIEC0YiluMvA4TnaJdLq_tWXIc4_Tq_KfpkvI0040NkgU7EAMEr1wZ4aDcJV2yf22gQ1sCSig6EGST
mmzDuEPsYiyd4NhidRZJP4HiiQh-hePBQsgcSgYGvz9wC6n57ufYKh2wm_Ti3Q
&requested_token_use=on_behalf_of
&scope=https://graph.microsoft.com/user.read+offline_access
```

Service to service access token response

A success response is a JSON OAuth 2.0 response with the following parameters.

| PARAMETER | DESCRIPTION |
|-------------------------|--|
| <code>token_type</code> | Indicates the token type value. The only type that Microsoft identity platform supports is <code>Bearer</code> . For more info about bearer tokens, see the OAuth 2.0 Authorization Framework: Bearer Token Usage (RFC 6750) . |
| <code>scope</code> | The scope of access granted in the token. |

| PARAMETER | DESCRIPTION |
|----------------------------|---|
| <code>expires_in</code> | The length of time, in seconds, that the access token is valid. |
| <code>access_token</code> | The requested access token. The calling service can use this token to authenticate to the receiving service. |
| <code>refresh_token</code> | The refresh token for the requested access token. The calling service can use this token to request another access token after the current access token expires. The refresh token is only provided if the <code>offline_access</code> scope was requested. |

Success response example

The following example shows a success response to a request for an access token for the

<https://graph.microsoft.com> web API.

```
{
  "token_type": "Bearer",
  "scope": "https://graph.microsoft.com/user.read",
  "expires_in": 3269,
  "ext_expires_in": 0,
  "access_token": "eyJ0eXAiOiJKV1QiLCJub25jZSI6IkFRQUJBQUFBQmZPrY1tQTZOVGFlN0NkV1c3UWZkQ0NDYy0tY0hGa18wZE50MVEtc2loVzRMd2RwQVZISGpnTVdQz0tQeVJIAg1DbUN2NkdyMEpmYmRfY1RmUFxU21TcFJkVXVydVJqX3Nqd0JoN211eH1BQSIsImFsZyI6IlJTmjuIiwiDV0IjoiejAzOxpkc0Z1aXpwQmZCVksxVG4yNVFIWU8wIiwi2lkIjoejAzOxpkc0Z1aXpwQmZCVksxVG4yNVFIWU8wIn0.eyJhdWQiOiJodHRwczovL2dyYXB0Lm1pY3Jvc29mdC5jb20iLCJpc3MiOjJodHRwczovL3N0cy53aw5kb3dzLm51dC83MmY50DhiZ1o4NmYxLTQxYwYtOTfhiYi0yZDdjZDAxMWRiNDcvIiwiWF0IjoxNDkzOTMwMzA1LCJuYmYi0jE00TM5MzAzMDUsImV4cCI6MTQ5MzkzMzg3NSwiYWNyIjoiMCIsImFpbvI6IkFTUUEyLzhEQUFBQ9KYnFFW1RNTnEyZFcxxYxpKN1RZMD1YeDd0T29EMkJEU1RWXJ3b2Zrc1k9IiwiYW1yIjpbInB3ZCJdLCJhcHBfZGlzcGxew5hbWu0iJUb2RvRG90bmV0T2JvIiwiYXBwaWQoIiY0DQ2ZjcxYi1hN2E0LTQ50DctYmFiMy03NjAwMzViMmYzODkiLCJhcHBpZGFjciI6IjEiLCJmYW1pbHfbmFtZSI6IkNhbnVtYwxsYSIsImdpdmVuX25hbWUi0iJOYXZ5YSiSImlwYWRkciI6IjE2Ny4yMjAuMC4xOTkiLCJuYWl1IjoiTmF2eWeqQ2FudW1hbGxhIiwi2lkIjoiZDV10Tc5YzctM2QyZC00MmFmLThmMzAtNzI3ZGQ0YzJkMzg2iwiw25wcmVtX3NpZCI6I1MtMS01LTixLTixMjc1MjExODQtMTYwNDAxMjkycMC0xODg30T13NTI3LT12MTE4NDg0IiwickGxhdGYiOiIxNCIsInB1aWQoIiIxMDAzm0ZGrkEwNkQxN0M5IiwiC2NwIjoiVXNlcis5ZWFKiIiwiC3ViIjoiwbWtMMHbiLXlpMXQ1ckRGd2J7Z1jTwrxZE52b3UzSjNWNm84UFE3a1VCRSIsInRpZC16IjcyZjk40GJmLTg2ZjEtNDfhZi05MWfLTJkN2NkMDExZG10NyIsInVuaXF1ZV9uYw1IjoiBfmFjYw51bwFABwljcm9zb2Z0LmNvbSIsInWbii6Im5hY2FudW1hQG1pY3Jvc29mdC5jb20iLCJ1dGkiOjJWR1ItdmtEZ1BFQ2M1dwFdAEnRSkFBiwidmVyIjoiMS4wIn0.cubh1L2VtruiwF8ut1m9uNBmnUJeYx4x0G30F7CqSpzHj1Sv5DCgNZxyUz3pEiz77G81f0F0_U5A_02k-xzwdYvtJUYGH3bfISzdqymiEGmdFcIRK19KMeo211Gv0ScCniIhr2U1yxTIkip092xcdadT-2_2q_q1Ha_HtvTV1f9XR3t7_Id9bR5BqwVX5zP07JMYDvhUzRx08eqZcc-F3wi0xd_5ND_mavMux2wrpF-EZvi03yg0QVRr59tE3AoW181SGpVc97vvRCnp4WVrk26jJhYXFpsdk4yWqOKZqr3IFGyD08WizD_vPSrXcCPbZP3XWaoTUKZSNJg",
  "refresh_token": "OAQABAAAAAAABnfig-"
}
```

NOTE

The above access token is a v1.0-formatted token. This is because the token is provided based on the resource being accessed. The Microsoft Graph requests v1.0 tokens, so Microsoft identity platform produces v1.0 access tokens when a client requests tokens for Microsoft Graph. Only applications should look at access tokens. Clients should not need to inspect them.

Error response example

An error response is returned by the token endpoint when trying to acquire an access token for the downstream

API, if the downstream API has a Conditional Access policy (such as multi-factor authentication) set on it. The middle-tier service should surface this error to the client application so that the client application can provide the user interaction to satisfy the Conditional Access policy.

```
{  
    "error": "interaction_required",  
    "error_description": "AADSTS50079: Due to a configuration change made by your administrator, or because  
you moved to a new location, you must enroll in multi-factor authentication to access 'bf8d80f9-9098-4972-  
b203-500f535113b1'.\r\nTrace ID: b72a68c3-0926-4b8e-bc35-3150069c2800\r\nCorrelation ID: 73d656cf-54b1-4eb2-  
b429-26d8165a52d7\r\nTimestamp: 2017-05-01 22:43:20Z",  
    "error_codes": [50079],  
    "timestamp": "2017-05-01 22:43:20Z",  
    "trace_id": "b72a68c3-0926-4b8e-bc35-3150069c2800",  
    "correlation_id": "73d656cf-54b1-4eb2-b429-26d8165a52d7",  
    "claims": "{\"access_token\": {\"polids\": {\"essential\": true, \"values\": [\"9ab03e19-ed42-4168-b6b7-  
7001fb3e933a\"]}}}"  
}
```

Use the access token to access the secured resource

Now the middle-tier service can use the token acquired above to make authenticated requests to the downstream web API, by setting the token in the `Authorization` header.

Example

```
GET /v1.0/me HTTP/1.1  
Host: graph.microsoft.com  
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJub25jZSI6IkFRQUJBQUFBCbmZpRy1tQTZ0VGFlN0NKV1c3UWZkSzdnN0RyNXlVuUDLNmFc19vdDF3cEQyZjNqR  
kxiNlVrcm9PcXA2cXBjclAxZVV0QktzMHEza29HN3RzXzJpSkYtQjY1UV8zVGgzSnktUHZsMjkxaFNBQStsImFsZyI6IlJTMjU2IiwieDV0Ij  
oiejAzOXpkc0Z1aXpwQmZCVksxVG4yNVFIWU8wIiwi21kIjoiejAzOXpkc0Z1aXpwQmZCVksxVG4yNVFIWU8wIn0.eyJhdWQiOjodHRwczo  
vL2dyYXBoLm1pY3Jvc29mdC5jb20iLCJpc3MiOjodHRwczoL3N0cy53aW5kb3dzLm51dC83MmY50DhiZi04NmYxLTQxYWYtOTFhYi0yZDdj  
ZDAxMWRiNDcvIiwiawF0IjoxNDkzOTMwMDEx2LCJuYmYi0jE0OTM5MzAwMTYsImV4cCI6MTQ5MzkzMzg3NSwiYWNyIjoiMCIsImFpbvI6IkFTU  
UEyLzhEQUFBQu1zQjN5ZU1jnKz1aEhkdlYxckoxS1d1bzJPckZOUQwN2FENTVjUVRtems9IiwiYw1yIjpblnB3ZCJdLCJhcHBfZG1zcGxheW  
5hbWUiOjUb2RvRG90bmV0T2JvIiwiYXBwAWQiOjIyODQ2ZjcxYi1hN2E0LTQ50DctYmFiMy03NjAwMzViMmYzODkiLCJhcHBpZGFjciI6IjE  
iLCJmY1pbHfbmFtZSI6IkNhbnVtYWxsYSIsImdpdmVuX25hbWUiOjJOYXZ5YSIsIm1lwYWRkciI6IjE2Ny4yMjAuMC4xOTkiLCJuYw1Ijoi  
TmF2eWEgQ2FudW1hbGxhIiwb21kIjoiZDV10Tc5YzctM2QyZC00MmFmLThmMzAtNzI3ZGQ0YzJkMzgZIiwb25wcmVtX3NpZCI6I1MtMS01L  
TIxLTIxMjc1MjExODQtMTYwNDAxMjkyMC0xODg30TI3NTI3LTi2MTE4NDg0IiwigxhdGYiOiiXNCIsInB1aWQioiIxMDAzm0ZGRkEwNkQxN0  
M5IiwiC2NwIjoiVXNlcis5ZWfkIiwiic3ViIjoiBwtMMHbiLXlpMXQ1ckRGd2JTz1JvTwxrZE52b3UzSjNWNm84UFE3alVCRSiisInRpZCI6Ijc  
yZjk40GJmlTg2ZjEtNDFhZi05MWFILTJkN2NkMDExZGI0NyIsInVuaXF1Zv9uYW11IjoiibmFjYW51bWFAbWljqcm9zb2Z0LmNvbSiisInVwbiI6  
Im5hY2FudW1hQG1pY3Jvc29mdC5jb20iLCJ1dGkiOjIjzUv1VekYxdUVVS0NQS0dRTVFVrkFB1iwidmVtIjoiMS4wIn0.Hrn__RGi-  
HMAzYRyCqX3kBGB60S7z7y49XPVPpwK_7rJ6nik9E4s6PNY4XkIamJYn7tphpmsHdfM91Q1gqeeFvFGhweIACsNBWhJ9Nx4dvQnGRkqZ17KnF  
_wf_QLcyOr0WpUxdSD_oPKcPS-Qr5AFkjw0t7GOKLY-  
Xw3QLJhzeKmYuuOkmMDJDA10eNDbH0HiCh3g189a176Bfyar0MgK8wrXI_6MTnFSVfBePqk1QeLhcr50YTBFwg3Svgl6MuK_g1h0ua0-  
XpjUxpdv5dZ0svI47fAuVDdpCE48igCX5Vmj4KUVytDif6T78aIXMKYHGgW3-xAmuSyYH_Fr0yVAQ
```

Gaining consent for the middle-tier application

Depending on the audience for your application, you may consider different strategies for ensuring that the OBO flow is successful. In all cases, the ultimate goal is to ensure proper consent is given. How that occurs, however, depends on which users your application supports.

Consent for Azure AD-only applications

`/.default` and combined consent

For applications that only need to sign in work or school accounts, the traditional "Known Client Applications" approach is sufficient. The middle tier application adds the client to the known client applications list in its manifest, and then the client can trigger a combined consent flow for both itself and the middle tier application. On the Microsoft identity platform endpoint, this is done using the `/.default` scope. When triggering a consent screen using known client applications and `/.default`, the consent screen will show permissions for both the

client to the middle tier API, and also request whatever permissions are required by the middle-tier API. The user provides consent for both applications, and then the OBO flow works.

At this time, the personal Microsoft account system does not support combined consent and so this approach does not work for apps that want to specifically sign in personal accounts. Personal Microsoft accounts being used as guest accounts in a tenant are handled using the Azure AD system, and can go through combined consent.

Pre-authorized applications

A feature of the application portal is "pre-authorized applications". In this way, a resource can indicate that a given application always has permission to receive certain scopes. This is primarily useful to make connections between a front-end client and a back-end resource more seamless. A resource can declare multiple pre-authorized applications - any such application can request these permissions in an OBO flow and receive them without the user providing consent.

Admin consent

A tenant admin can guarantee that applications have permission to call their required APIs by providing admin consent for the middle tier application. To do this, the admin can find the middle tier application in their tenant, open the required permissions page, and choose to give permission for the app. To learn more about admin consent, see the [consent and permissions documentation](#).

Consent for Azure AD + Microsoft account applications

Because of restrictions in the permissions model for personal accounts and the lack of a governing tenant, the consent requirements for personal accounts are a bit different from Azure AD. There is no tenant to provide tenant-wide consent for, nor is there the ability to do combined consent. Thus, other strategies present themselves - note that these work for applications that only need to support Azure AD accounts as well.

Use of a single application

In some scenarios, you may only have a single pairing of middle-tier and front-end client. In this scenario, you may find it easier to make this a single application, negating the need for a middle-tier application altogether. To authenticate between the front-end and the web API, you can use cookies, an id_token, or an access token requested for the application itself. Then, request consent from this single application to the back-end resource.

Client limitations

If a client uses the implicit flow to get an id_token, and that client also has wildcards in a reply URL, the id_token can't be used for an OBO flow. However, access tokens acquired through the implicit grant flow can still be redeemed by a confidential client even if the initiating client has a wildcard reply URL registered.

Next steps

Learn more about the OAuth 2.0 protocol and another way to perform service to service auth using client credentials.

- [OAuth 2.0 client credentials grant in Microsoft identity platform](#)
- [OAuth 2.0 code flow in Microsoft identity platform](#)
- [Using the `/default` scope](#)

Microsoft identity platform and the OAuth 2.0 client credentials flow

8/30/2019 • 11 minutes to read • [Edit Online](#)

Applies to:

- Microsoft identity platform endpoint

You can use the [OAuth 2.0 client credentials grant](#) specified in RFC 6749, sometimes called *two-legged OAuth*, to access web-hosted resources by using the identity of an application. This type of grant is commonly used for server-to-server interactions that must run in the background, without immediate interaction with a user. These types of applications are often referred to as *daemons* or *service accounts*.

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 a web site. For a higher level of assurance, the Microsoft identity platform also allows the calling service to use a certificate (instead of a shared secret) as a credential.

NOTE

The Microsoft identity platform endpoint doesn't support all Azure AD scenarios and features. To determine whether you should use the Microsoft identity platform endpoint, read about [Microsoft identity platform limitations](#).

In the more typical *three-legged OAuth*, a client application is granted permission to access a resource on behalf of a specific user. The permission is delegated from the user to the application, usually during the [consent](#) process. However, in the client credentials (*two-legged OAuth*) flow, permissions are granted directly to the application itself. When the app presents a token to a resource, the resource enforces that the app itself has authorization to perform an action and not the user.

Protocol diagram

The entire client credentials flow looks similar to the following diagram. We describe each of the steps later in this article.

Get direct authorization

An app typically receives direct authorization to access a resource in one of two ways:

- [Through an access control list \(ACL\) at the resource](#)
- [Through application permission assignment in Azure AD](#)

These two methods are the most common in Azure AD and we recommend them for clients and resources that perform the client credentials flow. A resource can also choose to authorize its clients in other ways. Each resource server can choose the method that makes the most sense for its application.

Access control lists

A resource provider might enforce an authorization check based on a list of application (client) IDs that it knows and grants a specific level of access to. When the resource receives a token from the Microsoft identity platform endpoint, it can decode the token and extract the client's application ID from the `appid` and `iss` claims. Then it compares the application against an access control list (ACL) that it maintains. The ACL's granularity and method might vary substantially between resources.

A common use case is to use an ACL to run tests for a web application or for a web API. The web API might grant only a subset of full permissions to a specific client. To run end-to-end tests on the API, create a test client that acquires tokens from the Microsoft identity platform endpoint and then sends them to the API. The API then checks the ACL for the test client's application ID for full access to the API's entire functionality. If you use this kind of ACL, be sure to validate not only the caller's `appid` value but also validate that the `iss` value of the token is trusted.

This type of authorization is common for daemons and service accounts that need to access data owned by consumer users who have personal Microsoft accounts. For data owned by organizations, we recommend that you get the necessary authorization through application permissions.

Application permissions

Instead of using ACLs, you can use APIs to expose a set of application permissions. An application permission is granted to an application by an organization's administrator, and can be used only to access data owned by that organization and its employees. For example, Microsoft Graph exposes several application permissions to do the following:

- Read mail in all mailboxes
- Read and write mail in all mailboxes
- Send mail as any user
- Read directory data

For more information about application permissions, go to [Microsoft Graph](#).

To use application permissions in your app, follow the steps discussed in the next sections.

Request the permissions in the app registration portal

1. Register and create an app through the new [App registrations \(Preview\)](#) experience.
2. Go to your application in the App registrations (Preview) experience. Navigate to the **Certificates & secrets** section, and add a **new client secret**, because you'll need at least one client secret to request a token.
3. Locate the **API permissions** section, and then add the **application permissions** that your app requires.
4. **Save** the app registration.

Recommended: Sign the user into your app

Typically, when you build an application that uses application permissions, the app requires a page or view on which the admin approves the app's permissions. This page can be part of the app's sign-in flow, part of the app's settings, or it can be a dedicated "connect" flow. In many cases, it makes sense for the app to show this "connect" view only after a user has signed in with a work or school Microsoft account.

If you sign the user into your app, you can identify the organization to which the user belongs to before you ask the user to approve the application permissions. Although not strictly necessary, it can help you create a more intuitive experience for your users. To sign the user in, follow our [Microsoft identity platform protocol tutorials](#).

Request the permissions from a directory admin

When you're ready to request permissions from the organization's admin, you can redirect the user to the Microsoft identity platform *admin consent endpoint*.

TIP

Try executing this request in Postman! (Use your own app ID for best results - the tutorial application won't request useful permissions.)

▶ [Run in Postman](#)

```
// Line breaks are for legibility only.

GET https://login.microsoftonline.com/{tenant}/adminconsent?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&state=12345
&redirect_uri=http://localhost/myapp/permissions
```

```
// Pro tip: Try pasting the following request in a browser.
```

```
https://login.microsoftonline.com/common/adminconsent?client_id=6731de76-14a6-49ae-97bc-
6eba6914391e&state=12345&redirect_uri=http://localhost/myapp/permissions
```

| PARAMETER | CONDITION | DESCRIPTION |
|--------------|-------------|--|
| tenant | Required | The directory tenant that you want to request permission from. This can be in GUID or friendly name format. If you don't know which tenant the user belongs to and you want to let them sign in with any tenant, use common . |
| client_id | Required | The Application (client) ID that the Azure portal – App registrations experience assigned to your app. |
| redirect_uri | Required | The redirect URI where you want the response to be sent for your app to handle. It must exactly match one of the redirect URIs that you registered in the portal, except that it must be URL encoded, and it can have additional path segments. |
| state | Recommended | A value that's included in the request that's also returned in the token response. It can be a string of any content that you want. The state is 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. |

At this point, Azure AD enforces that only a tenant administrator can sign into complete the request. The administrator will be asked to approve all the direct application permissions that you have requested for your app in the app registration portal.

Successful response

If the admin approves the permissions for your application, the successful response looks like this:

```
GET http://localhost/myapp/permissions?tenant=a8990e1f-ff32-408a-9f8e-  
78d3b9139b95&state=state=12345&admin_consent=True
```

| PARAMETER | DESCRIPTION |
|---------------|--|
| tenant | The directory tenant that granted your application the permissions that it requested, in GUID format. |
| state | A value that is included in the request that also is returned in the token response. It can be a string of any content that you want. The state is 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. |
| admin_consent | Set to True . |

Error response

If the admin does not approve the permissions for your application, the failed response looks like this:

```
GET http://localhost/myapp/permissions?  
error=permission_denied&error_description=The+admin+canceled+the+request
```

| PARAMETER | DESCRIPTION |
|-------------------|--|
| error | An error code string that you can use to classify types of errors, and which you can use to react to errors. |
| error_description | A specific error message that can help you identify the root cause of an error. |

After you've received a successful response from the app provisioning endpoint, your app has gained the direct application permissions that it requested. Now you can request a token for the resource that you want.

Get a token

After you've acquired the necessary authorization for your application, proceed with acquiring access tokens for APIs. To get a token by using the client credentials grant, send a POST request to the `/token` Microsoft identity platform endpoint:

TIP

Try executing this request in Postman! (Use your own app ID for best results - the tutorial application won't request useful permissions.)

[Run in Postman](#)

First case: Access token request with a shared secret

```

POST /{tenant}/oauth2/v2.0/token HTTP/1.1          //Line breaks for clarity
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

client_id=535fb089-9ff3-47b6-9bfb-4f1264799865
&scope=https%3A%2F%2Fgraph.microsoft.com%2F.default
&client_secret=qWgdYAmab0YSkuL1qKv5bPX
&grant_type=client_credentials

```

```

// Replace {tenant} with your tenant!
curl -X POST -H "Content-Type: application/x-www-form-urlencoded" -d 'client_id=535fb089-9ff3-47b6-9bfb-
4f1264799865&scope=https%3A%2F%2Fgraph.microsoft.com%2F.default&client_secret=qWgdYAmab0YSkuL1qKv5bPX&grant
_type=client_credentials' 'https://login.microsoftonline.com/{tenant}/oauth2/v2.0/token'

```

| PARAMETER | CONDITION | DESCRIPTION |
|----------------------------|-----------|--|
| <code>tenant</code> | Required | The directory tenant the application plans to operate against, in GUID or domain-name format. |
| <code>client_id</code> | Required | The application ID that's assigned to your app. You can find this information in the portal where you registered your app. |
| <code>scope</code> | Required | <p>The value passed for the <code>scope</code> parameter in this request should be the resource identifier (application ID URI) of the resource you want, affixed with the <code>.default</code> suffix. For the Microsoft Graph example, the value is https://graph.microsoft.com/.default.</p> <p>This value tells the Microsoft identity platform endpoint that of all the direct application permissions you have configured for your app, the endpoint should issue a token for the ones associated with the resource you want to use. To learn more about the <code>.default</code> scope, see the consent documentation.</p> |
| <code>client_secret</code> | Required | The client secret that you generated for your app in the app registration portal. The client secret must be URL-encoded before being sent. |
| <code>grant_type</code> | Required | Must be set to <code>client_credentials</code> . |

Second case: Access token request with a certificate

```

POST /{tenant}/oauth2/v2.0/token HTTP/1.1
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

// Line breaks for clarity

scope=https%3A%2F%2Fgraph.microsoft.com%2F.default
&client_id=97e0a5b7-d745-40b6-94fe-5f77d35c6e05
&client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJSUzI1NiIsIng1dCI6Imd40HRHeXN5amNScUtqR1BuZDdSRnd2d1pJMCJ9.eyJ{a lot of
characters here}M8U3bSUKKJDEg
&grant_type=client_credentials

```

| PARAMETER | CONDITION | DESCRIPTION |
|------------------------------------|-----------|---|
| <code>tenant</code> | Required | The directory tenant the application plans to operate against, in GUID or domain-name format. |
| <code>client_id</code> | Required | The application (client) ID that's assigned to your app. |
| <code>scope</code> | Required | <p>The value passed for the <code>scope</code> parameter in this request should be the resource identifier (application ID URI) of the resource you want, affixed with the <code>.default</code> suffix. For the Microsoft Graph example, the value is https://graph.microsoft.com/.default.</p> <p>This value informs the Microsoft identity platform endpoint that of all the direct application permissions you have configured for your app, it should issue a token for the ones associated with the resource you want to use. To learn more about the <code>/.default</code> scope, see the consent documentation.</p> |
| <code>client_assertion_type</code> | Required | <p>The value must be set to <code>urn:ietf:params:oauth:client-assertion-type:jwt-bearer</code>.</p> |
| <code>client_assertion</code> | 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 to learn how to register your certificate and the format of the assertion. |
| <code>grant_type</code> | Required | Must be set to <code>client_credentials</code> . |

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`.

Successful response

A successful response looks like this:

```
{
  "token_type": "Bearer",
  "expires_in": 3599,
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik1uQ19WVmNBVGZNNXBP... "
}
```

| PARAMETER | DESCRIPTION |
|---------------------------|---|
| <code>access_token</code> | The requested access token. The app can use this token to authenticate to the secured resource, such as to a Web API. |
| <code>token_type</code> | Indicates the token type value. The only type that Microsoft identity platform supports is <code>bearer</code> . |
| <code>expires_in</code> | The amount of time that an access token is valid (in seconds). |

Error response

An error response looks like this:

```
{
  "error": "invalid_scope",
  "error_description": "AADSTS70011: The provided value for the input parameter 'scope' is not valid. The scope https://foo.microsoft.com/.default is not valid.\r\nTrace ID: 255d1aef-8c98-452f-ac51-23d051240864\r\nCorrelation ID: fb3d2015-bc17-4bb9-bb85-30c5cf1aaaa7\r\nTimestamp: 2016-01-09 02:02:12Z",
  "error_codes": [
    70011
  ],
  "timestamp": "2016-01-09 02:02:12Z",
  "trace_id": "255d1aef-8c98-452f-ac51-23d051240864",
  "correlation_id": "fb3d2015-bc17-4bb9-bb85-30c5cf1aaaa7"
}
```

| PARAMETER | DESCRIPTION |
|--------------------------------|---|
| <code>error</code> | An error code string that you can use to classify types of errors that occur, and to react to errors. |
| <code>error_description</code> | A specific error message that might help you identify the root cause of an authentication error. |
| <code>error_codes</code> | A list of STS-specific error codes that might help with diagnostics. |
| <code>timestamp</code> | The time when the error occurred. |
| <code>trace_id</code> | A unique identifier for the request to help with diagnostics. |
| <code>correlation_id</code> | A unique identifier for the request to help with diagnostics across components. |

Use a token

Now that you've acquired a token, use the token to make requests to the resource. When the token expires, repeat the request to the `/token` endpoint to acquire a fresh access token.

```
GET /v1.0/me/messages
Host: https://graph.microsoft.com
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZn10aEV1Q...
```

```
// Pro tip: Try the following command! (Replace the token with your own.)
```

```
curl -X GET -H "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbG...)"
'https://graph.microsoft.com/v1.0/me/messages'
```

Code samples and other documentation

Read the [client credentials overview documentation](#) from the Microsoft Authentication Library

| SAMPLE | PLATFORM | DESCRIPTION |
|---|-----------------------|---|
| active-directory-dotnetcore-daemon-v2 | .NET Core 2.1 Console | A simple .NET Core application that displays the users of a tenant querying the Microsoft Graph using the identity of the application, instead of on behalf of a user. The sample also illustrates the variation using certificates for authentication. |
| active-directory-dotnet-daemon-v2 | ASP.NET MVC | A web application that syncs data from the Microsoft Graph using the identity of the application, instead of on behalf of a user. |

Microsoft identity platform and the OAuth 2.0 device authorization grant flow

10/24/2019 • 4 minutes to read • [Edit Online](#)

Applies to:

- Microsoft identity platform endpoint

The Microsoft identity platform supports the [device authorization grant](#), which allows users to sign in to input-constrained devices such as a smart TV, IoT device, or printer. To enable this flow, the device has the user visit a webpage in their browser on another device to sign in. Once the user signs in, the device is able to get access tokens and refresh tokens as needed.

NOTE

The Microsoft identity platform endpoint doesn't support all Azure Active Directory scenarios and features. To determine whether you should use the Microsoft identity platform endpoint, read about [Microsoft identity platform limitations](#).

Protocol diagram

The entire device code flow looks similar to the next diagram. We describe each of the steps later in this article.

Device authorization request

The client must first check with the authentication server for a device and user code that's used to initiate authentication. The client collects this request from the `/devicecode` endpoint. In this request, the client should also include the permissions it needs to acquire from the user. From the moment this request is sent, the user has only 15 minutes to sign in (the usual value for `expires_in`), so only make this request when the user has indicated they're ready to sign in.

TIP

Try executing this request in Postman! [▶ Run in Postman](#)

```
// Line breaks are for legibility only.
```

```
POST https://login.microsoftonline.com/{tenant}/oauth2/v2.0/devicecode
Content-Type: application/x-www-form-urlencoded
```

```
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
scope=user.read%20openid%20profile
```

| PARAMETER | CONDITION | DESCRIPTION |
|------------------------|-------------|--|
| <code>tenant</code> | Required | Can be /common, /consumers, or /organizations. It can also be the directory tenant that you want to request permission from in GUID or friendly name format. |
| <code>client_id</code> | Required | The Application (client) ID that the Azure portal – App registrations experience assigned to your app. |
| <code>scope</code> | Recommended | A space-separated list of scopes that you want the user to consent to. |

Device authorization response

A successful response will be a JSON object containing the required information to allow the user to sign in.

| PARAMETER | FORMAT | DESCRIPTION |
|-------------------------------|--------|--|
| <code>device_code</code> | String | A long string used to verify the session between the client and the authorization server. The client uses this parameter to request the access token from the authorization server. |
| <code>user_code</code> | String | A short string shown to the user that's used to identify the session on a secondary device. |
| <code>verification_uri</code> | URI | The URI the user should go to with the <code>user_code</code> in order to sign in. |
| <code>expires_in</code> | int | The number of seconds before the <code>device_code</code> and <code>user_code</code> expire. |
| <code>interval</code> | int | The number of seconds the client should wait between polling requests. |
| <code>message</code> | String | A human-readable string with instructions for the user. This can be localized by including a query parameter in the request of the form <code>?mkt=xx-XX</code> , filling in the appropriate language culture code. |

NOTE

The `verification_uri_complete` response field is not included or supported at this time. We mention this because if you read the [standard](#) you see that `verification_uri_complete` is listed as an optional part of the device code flow standard.

Authenticating the user

After receiving the `user_code` and `verification_uri`, the client displays these to the user, instructing them to sign in using their mobile phone or PC browser.

If the user authenticates with a personal account (on /common or /consumers), they will be asked to sign in again in order to transfer authentication state to the device. They will also be asked to provide consent, to ensure they are aware of the permissions being granted. This does not apply to work or school accounts used to authenticate.

While the user is authenticating at the `verification_uri`, the client should be polling the `/token` endpoint for the requested token using the `device_code`.

```
POST https://login.microsoftonline.com/{tenant}/oauth2/v2.0/token
Content-Type: application/x-www-form-urlencoded

grant_type: urn:ietf:params:oauth:grant-type:device_code
client_id: 6731de76-14a6-49ae-97bc-6eba6914391e
device_code: GMMhmHCXhWEzkobqIHGG_EnNYYsAkukHspeYUk9E8...
```

| PARAMETER | REQUIRED | DESCRIPTION |
|--------------------------|----------|--|
| <code>tenant</code> | Required | The same tenant or tenant alias used in the initial request. |
| <code>grant_type</code> | Required | Must be
<code>urn:ietf:params:oauth:grant-type:device_code</code> |
| <code>client_id</code> | Required | Must match the <code>client_id</code> used in the initial request. |
| <code>device_code</code> | Required | The <code>device_code</code> returned in the device authorization request. |

Expected errors

The device code flow is a polling protocol so your client must expect to receive errors before the user has finished authenticating.

| ERROR | DESCRIPTION | CLIENT ACTION |
|-------------------------------------|---|--|
| <code>authorization_pending</code> | The user hasn't finished authenticating, but hasn't canceled the flow. | Repeat the request after at least <code>interval</code> seconds. |
| <code>authorization_declined</code> | The end user denied the authorization request. | Stop polling, and revert to an unauthenticated state. |
| <code>bad_verification_code</code> | The <code>device_code</code> sent to the <code>/token</code> endpoint wasn't recognized. | Verify that the client is sending the correct <code>device_code</code> in the request. |
| <code>expired_token</code> | At least <code>expires_in</code> seconds have passed, and authentication is no longer possible with this <code>device_code</code> . | Stop polling and revert to an unauthenticated state. |

Successful authentication response

A successful token response will look like:

```
{
  "token_type": "Bearer",
  "scope": "User.Read profile openid email",
  "expires_in": 3599,
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZnl0aEV1Q...",
  "refresh_token": "AwABAAAAvPM1KaPlrEqdFSBzjqfTGAMxZGUTdM0t4B4...",
  "id_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhdWQiOiIyZDRkMTFhMi1mODE0LTQ2YTctOD..."
}
```

| PARAMETER | FORMAT | DESCRIPTION |
|----------------------------|-------------------------|---|
| <code>token_type</code> | String | Always "Bearer." |
| <code>scope</code> | Space separated strings | If an access token was returned, this lists the scopes the access token is valid for. |
| <code>expires_in</code> | int | Number of seconds before the included access token is valid for. |
| <code>access_token</code> | Opaque string | Issued for the scopes that were requested. |
| <code>id_token</code> | JWT | Issued if the original <code>scope</code> parameter included the <code>openid</code> scope. |
| <code>refresh_token</code> | Opaque string | Issued if the original <code>scope</code> parameter included <code>offline_access</code> . |

You can use the refresh token to acquire new access tokens and refresh tokens using the same flow documented in the [OAuth Code flow documentation](#).

Microsoft identity platform and the OAuth 2.0 resource owner password credential

10/13/2019 • 3 minutes to read • [Edit Online](#)

Microsoft identity platform supports the [resource owner password credential \(ROPC\) grant](#), which allows an application to sign in the user by directly handling their password. The ROPC flow requires a high degree of trust and user exposure and you should only use this flow when other, more secure, flows can't be used.

IMPORTANT

- The Microsoft identity platform endpoint only supports ROPC for Azure AD tenants, not personal accounts. This means that you must use a tenant-specific endpoint (https://login.microsoftonline.com/{TenantId_or_Name}) or the [organizations](#) endpoint.
- Personal accounts that are invited to an Azure AD tenant can't use ROPC.
- Accounts that don't have passwords can't sign in through ROPC. For this scenario, we recommend that you use a different flow for your app instead.
- If users need to use multi-factor authentication (MFA) to log in to the application, they will be blocked instead.
- ROPC is not supported in [hybrid identity federation](#) scenarios (for example, Azure AD and ADFS used to authenticate on-premise accounts). If users are full-page redirected to an on-premises identity providers, Azure AD is not able to test the username and password against that identity provider. [Pass-through authentication](#) is supported with ROPC, however.

Protocol diagram

The following diagram shows the ROPC flow.

Authorization request

The ROPC flow is a single request: it sends the client identification and user's credentials to the IDP, and then receives tokens in return. The client must request the user's email address (UPN) and password before doing so. Immediately after a successful request, the client should securely release the user's credentials from memory. It must never save them.

TIP

Try executing this request in Postman!

[▶ Run in Postman](#)

```
// Line breaks and spaces are for legibility only. This is a public client, so no secret is required.
```

```
POST {tenant}/oauth2/v2.0/token
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&scope=user.read%20openid%20profile%20offline_access
&username=MyUsername@myTenant.com
&password=SuperS3cret
&grant_type=password
```

| PARAMETER | CONDITION | DESCRIPTION |
|-------------------------------|--------------------|---|
| <code>tenant</code> | Required | The directory tenant that you want to log the user into. This can be in GUID or friendly name format. This parameter can't be set to <code>common</code> or <code>consumers</code> , but may be set to <code>organizations</code> . |
| <code>client_id</code> | Required | The Application (client) ID that the Azure portal - App registrations page assigned to your app. |
| <code>grant_type</code> | Required | Must be set to <code>password</code> . |
| <code>username</code> | Required | The user's email address. |
| <code>password</code> | Required | The user's password. |
| <code>scope</code> | Recommended | A space-separated list of scopes , or permissions, that the app requires. In an interactive flow, the admin or the user must consent to these scopes ahead of time. |
| <code>client_secret</code> | Sometimes required | If your app is a public client, then the <code>client_secret</code> or <code>client_assertion</code> cannot be included. If the app is a confidential client, then it must be included. |
| <code>client_assertion</code> | Sometimes required | A different form of <code>client_secret</code> , generated using a certificate. See certificate credentials for more details. |

Successful authentication response

The following example shows a successful token response:

```
{
  "token_type": "Bearer",
  "scope": "User.Read profile openid email",
  "expires_in": 3599,
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZnl0aEV1Q...",
  "refresh_token": "AwABAAAAvPM1KaPlrEqdFSBzjqfTGAMxZGUTdM0t4B4...",
  "id_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhdWQiOiIyZDRkMTFhMi1mODE0LTQ2YTctOD..."}
```

| PARAMETER | FORMAT | DESCRIPTION |
|----------------------------|-------------------------|---|
| <code>token_type</code> | String | Always set to <code>Bearer</code> . |
| <code>scope</code> | Space separated strings | If an access token was returned, this parameter lists the scopes the access token is valid for. |
| <code>expires_in</code> | int | Number of seconds that the included access token is valid for. |
| <code>access_token</code> | Opaque string | Issued for the scopes that were requested. |
| <code>id_token</code> | JWT | Issued if the original <code>scope</code> parameter included the <code>openid</code> scope. |
| <code>refresh_token</code> | Opaque string | Issued if the original <code>scope</code> parameter included <code>offline_access</code> . |

You can use the refresh token to acquire new access tokens and refresh tokens using the same flow described in the [OAuth Code flow documentation](#).

Error response

If the user hasn't provided the correct username or password, or the client hasn't received the requested consent, authentication will fail.

| ERROR | DESCRIPTION | CLIENT ACTION |
|------------------------------|--|--|
| <code>invalid_grant</code> | The authentication failed | The credentials were incorrect or the client doesn't have consent for the requested scopes. If the scopes aren't granted, a <code>consent_required</code> error will be returned. If this occurs, the client should send the user to an interactive prompt using a webview or browser. |
| <code>invalid_request</code> | The request was improperly constructed | The grant type isn't supported on the <code>/common</code> or <code>/consumers</code> authentication contexts. Use <code>/organizations</code> or a tenant ID instead. |

Learn more

- Try out ROPC for yourself using the [sample console application](#).
- To determine whether you should use the v2.0 endpoint, read about [Microsoft identity platform limitations](#).

Microsoft identity platform and OAuth 2.0 SAML bearer assertion flow

8/13/2019 • 3 minutes to read • [Edit Online](#)

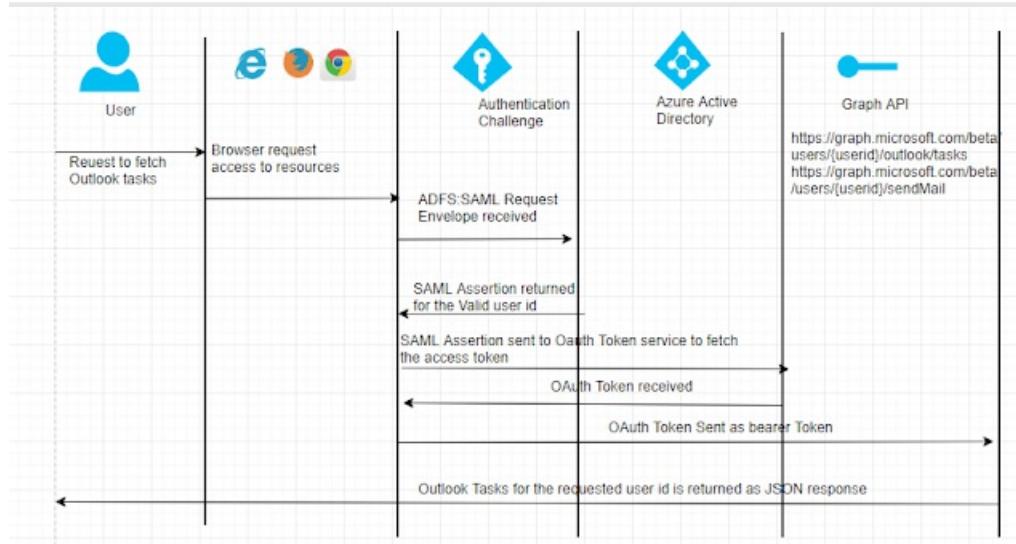
The OAuth 2.0 SAML bearer assertion flow allows you to request an OAuth access token using a SAML assertion when a client needs to use an existing trust relationship. The signature applied to the SAML assertion provides authentication of the authorized app. A SAML assertion is an XML security token issued by an identity provider and consumed by a service provider. The service provider relies on its content to identify the assertion's subject for security-related purposes.

The SAML assertion is posted to the OAuth token endpoint. The endpoint processes the assertion and issues an access token based on prior approval of the app. The client isn't required to have or store a refresh token, nor is the client secret required to be passed to the token endpoint.

SAML Bearer Assertion flow is useful when fetching data from Microsoft Graph APIs (which only support delegated permissions) without prompting the user for credentials. In this scenario the client credentials grant, which is preferred for background processes, does not work.

For applications that do interactive browser-based sign-in to get a SAML assertion and then want to add access to an OAuth protected API (such as Microsoft Graph), you can make an OAuth request to get an access token for the API. When the browser is redirected to Azure AD to authenticate the user, the browser will pick up the session from the SAML sign-in and the user doesn't need to enter their credentials.

The OAuth SAML Bearer Assertion flow is also supported for users authenticating with identity providers such as Active Directory Federation Services (ADFS) federated to Azure Active Directory. The SAML assertion obtained from ADFS can be used in an OAuth flow to authenticate the user.



Call Graph using SAML bearer assertion

Now let us understand on how we can actually fetch SAML assertion programatically. This approach is tested with ADFS. However, this works with any identity provider that supports the return of SAML assertion programatically. The basic process is: get a SAML assertion, get an access token, and access Microsoft Graph.

Prerequisites

Establish a trust relationship between the authorization server/environment (Microsoft 365) and the identity provider, or issuer of the SAML 2.0 bearer assertion (ADFS). To configure ADFS for single sign-on and as an identity provider you may refer to [this article](#).

Register the application in the [portal](#):

1. Sign in to the [app registration blade of the portal](#) (Please note that we are using the v2.0 endpoints for Graph API and hence need to register the application in this portal. Otherwise we could have used the registrations in Azure active directory).
2. Select **New registration**.
3. When the **Register an application** page appears, enter your application's registration information:
 - a. **Name** - Enter a meaningful application name that will be displayed to users of the app.
 - b. **Supported account types** - Select which accounts you would like your application to support.
 - c. **Redirect URI (optional)** - Select the type of app you're building, Web, or Public client (mobile & desktop), and then enter the redirect URI (or reply URL) for your application.
 - d. When finished, select **Register**.
4. Make a note of the application (client) ID.
5. In the left pane, select **Certificates & secrets**. Click **New client secret** in the **Client secrets** section. Copy the new client secret, you won't be able to retrieve when you leave the blade.
6. In the left pane, select **API permissions** and then **Add a permission**. Select **Microsoft Graph**, then **delegated permissions**, and then select **Tasks.read** since we intend to use the Outlook Graph API.

Install [Postman](#), a tool required to test the sample requests. Later, you can convert the requests to code.

Get the SAML assertion from ADFS

Create a POST request to the ADFS endpoint using SOAP envelope to fetch the SAML assertion:

The screenshot shows the Postman interface with the following details:

- Method: POST
- URL: https://ADFSFQDN/adfs/services/trust/2005/usernamemixed
- Headers tab is selected.
- Body tab is selected.
- Params tab is selected.
- Authorization tab is selected.
- Headers table:

| KEY | VALUE | DESCRIPTION | ... | Bulk Edit |
|-------------------|-----------|-------------|-----|-----------|
| client-request-id | CLIENT_ID | | | |
| Key | Value | Description | | |

Header values:

The screenshot shows the Postman interface with the following details:

- Method: POST
- URL: https://ADFSFQDN/adfs/services/trust/2005/usernamemixed
- Headers tab is selected.
- Body tab is selected.
- Params tab is selected.
- Authorization tab is selected.
- Headers table:

| KEY | VALUE | DESCRIPTION | ... | Bulk Edit | Presets |
|--------------------------|--|-------------|-----|-----------|---------|
| SOAPAction | https://schemas.xmlsoap.org/ws/2005/02/trust/RST/Issue | | | | |
| Content-Type | application/soap+xml | | | | |
| client-request-id | CLIENT_ID | | | | |
| return-client-request-id | true | | | | |
| Accept | application/json | | | | |
| Key | Value | Description | | | |

ADFS request body:

```

1 <s:Envelope xmlns:s='http://www.w3.org/2003/05/soap-envelope'
2   xmlns:a='http://www.w3.org/2005/08/addressing'
3   xmlns:wss='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd'
4     <s:Header><a:Action s:mustUnderstand='1'>http://schemas.xmlsoap.org/ws/2005/02/trust/RST/Issue</a:Action>
5     <a:MessageID>urn:uuid:90F33D03-1F9E-466C-993B-C9A8B222557</a:MessageID>
6     <a:ReplyTo><a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address></a:ReplyTo>
7     <a:To s:mustUnderstand='1'>https://ADFSQDN/adfs/services/trust/2005/usernamemixed</a:To>
8     <o:Security s:mustUnderstand='1' xmlns:o='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'>
9       <o:UsernameToken id='uid-25258SEF-6@A-4420-83BA-e0E026F40009'>
10         <o:Username>USERNAME</o:Username>
11         <o>Password>PASSWORD</o>Password></o:UsernameToken>
12       </o:Security>
13     </s:Header>
14   <s:Body>
15     <trust:RequestSecurityToken xmlns:trust='http://schemas.xmlsoap.org/ws/2005/02/trust'>
16       <wsp:AppliesTo xmlns:wsp='http://schemas.xmlsoap.org/ws/2004/09/policy'>
17         <a:EndpointReference>
18           <a:Address>urn:federation:MicrosoftOnline</a:Address>
19         </a:EndpointReference>
20       </wsp:AppliesTo>
21       <trust:KeyType>http://schemas.xmlsoap.org/ws/2005/05/identity/NoProofKey</trust:KeyType>
22       <trust:RequestType>http://schemas.xmlsoap.org/ws/2005/02/trust/Issue</trust:RequestType>
23       <trust:RequestSecurityToken>
24     </s:Body>
25   </s:Envelope>

```

Once this request is posted successfully, you should receive a SAML assertion from ADFS. Only the **SAML:Assertion** tag data is required, convert it to base64 encoding to use in further requests.

Get the OAuth2 token using the SAML assertion

In this step, fetch an OAuth2 token using the ADFS assertion response.

1. Create a POST request as shown below with the header values:

| KEY | VALUE | DESCRIPTION |
|--|-----------------------------------|-------------|
| <input checked="" type="checkbox"/> Host | login.microsoftonline.com | |
| <input checked="" type="checkbox"/> Content-Type | application/x-www-form-urlencoded | |
| Key | Value | Description |

2. In the body of the request, replace **client_id**, **client_secret**, and **assertion** (the base64 encoded SAML assertion obtained the previous step):

| KEY | VALUE | DESCRIPTION |
|---|---|-------------|
| <input checked="" type="checkbox"/> grant_type | urn:ietf:params:oauth:grant-type:saml1_1-bearer | |
| <input checked="" type="checkbox"/> client_id | CLIENTID | X |
| <input checked="" type="checkbox"/> client_secret | CLIENTSECRET | |
| <input checked="" type="checkbox"/> assertion | ASSERTION | |
| <input checked="" type="checkbox"/> scope | openid https://graph.microsoft.com/.default | |
| Key | Value | Description |

3. Upon successful request, you will receive an access token from Azure active directory.

Get the data with the Oauth token

After receiving the access token, call the Graph APIs (Outlook tasks in this example).

1. Create a GET request with the access token fetched in the previous step:

Graph API Outlook Tasks

Examples (0) ▾

GET https://graph.microsoft.com/beta/users/<USEREMAIL>/outlook/tasks

Send Save

Params Authorization Headers (2) Body Pre-request Script Tests Cookies Code

| KEY | VALUE | DESCRIPTION | *** | Bulk Edit | Presets ▾ |
|---|-----------------------------------|-------------|-----|-----------|-----------|
| <input type="checkbox"/> Content-Type | application/x-www-form-urlencoded | | | | |
| <input checked="" type="checkbox"/> Authorization | ACCESS TOKEN | | | | |
| Key | Value | Description | | | |

curl -X GET "https://graph.microsoft.com/beta/users/<USEREMAIL>/outlook/tasks" -H "Content-Type: application/x-www-form-urlencoded" -H "Authorization: ACCESS TOKEN"

- Upon successful request, you will receive a JSON response.

Next steps

Learn about the different [authentication flows and application scenarios](#).

Signing key rollover in Azure Active Directory

8/6/2019 • 13 minutes to read • [Edit Online](#)

This article discusses what you need to know about the public keys that are used in Azure Active Directory (Azure AD) to sign security tokens. It is important to note that these keys roll over on a periodic basis and, in an emergency, could be rolled over immediately. All applications that use Azure AD should be able to programmatically handle the key rollover process or establish a periodic manual rollover process. Continue reading to understand how the keys work, how to assess the impact of the rollover to your application and how to update your application or establish a periodic manual rollover process to handle key rollover if necessary.

Overview of signing keys in Azure AD

Azure AD uses public-key cryptography built on industry standards to establish trust between itself and the applications that use it. In practical terms, this works in the following way: Azure AD uses a signing key that consists of a public and private key pair. When a user signs in to an application that uses Azure AD for authentication, Azure AD creates a security token that contains information about the user. This token is signed by Azure AD using its private key before it is sent back to the application. To verify that the token is valid and originated from Azure AD, the application must validate the token's signature using the public key exposed by Azure AD that is contained in the tenant's [OpenID Connect discovery document](#) or SAML/WS-Fed [federation metadata document](#).

For security purposes, Azure AD's signing key rolls on a periodic basis and, in the case of an emergency, could be rolled over immediately. Any application that integrates with Azure AD should be prepared to handle a key rollover event no matter how frequently it may occur. If it doesn't, and your application attempts to use an expired key to verify the signature on a token, the sign-in request will fail.

There is always more than one valid key available in the OpenID Connect discovery document and the federation metadata document. Your application should be prepared to use any of the keys specified in the document, since one key may be rolled soon, another may be its replacement, and so forth.

How to assess if your application will be affected and what to do about it

How your application handles key rollover depends on variables such as the type of application or what identity protocol and library was used. The sections below assess whether the most common types of applications are impacted by the key rollover and provide guidance on how to update the application to support automatic rollover or manually update the key.

- [Native client applications accessing resources](#)
- [Web applications / APIs accessing resources](#)
- [Web applications / APIs protecting resources and built using Azure App Services](#)
- [Web applications / APIs protecting resources using .NET OWIN OpenID Connect, WS-Fed or WindowsAzureActiveDirectoryBearerAuthentication middleware](#)
- [Web applications / APIs protecting resources using .NET Core OpenID Connect or JwtBearerAuthentication middleware](#)
- [Web applications / APIs protecting resources using Node.js passport-azure-ad module](#)
- [Web applications / APIs protecting resources and created with Visual Studio 2015 or later](#)
- [Web applications protecting resources and created with Visual Studio 2013](#)
- [Web APIs protecting resources and created with Visual Studio 2013](#)
- [Web applications protecting resources and created with Visual Studio 2012](#)

- Web applications protecting resources and created with Visual Studio 2010, 2008 or using Windows Identity Foundation
- Web applications / APIs protecting resources using any other libraries or manually implementing any of the supported protocols

This guidance is **not** applicable for:

- Applications added from Azure AD Application Gallery (including Custom) have separate guidance with regards to signing keys. [More information.](#)
- On-premises applications published via application proxy don't have to worry about signing keys.

Native client applications accessing resources

Applications that are only accessing resources (i.e Microsoft Graph, KeyVault, Outlook API, and other Microsoft APIs) generally only obtain a token and pass it along to the resource owner. Given that they are not protecting any resources, they do not inspect the token and therefore do not need to ensure it is properly signed.

Native client applications, whether desktop or mobile, fall into this category and are thus not impacted by the rollover.

Web applications / APIs accessing resources

Applications that are only accessing resources (i.e Microsoft Graph, KeyVault, Outlook API, and other Microsoft APIs) generally only obtain a token and pass it along to the resource owner. Given that they are not protecting any resources, they do not inspect the token and therefore do not need to ensure it is properly signed.

Web applications and web APIs that are using the app-only flow (client credentials / client certificate), fall into this category and are thus not impacted by the rollover.

Web applications / APIs protecting resources and built using Azure App Services

Azure App Services' Authentication / Authorization (EasyAuth) functionality already has the necessary logic to handle key rollover automatically.

Web applications / APIs protecting resources using .NET OWIN OpenID Connect, WS-Fed or WindowsAzureActiveDirectoryBearerAuthentication middleware

If your application is using the .NET OWIN OpenID Connect, WS-Fed or WindowsAzureActiveDirectoryBearerAuthentication middleware, it already has the necessary logic to handle key rollover automatically.

You can confirm that your application is using any of these by looking for any of the following snippets in your application's Startup.cs or Startup.Auth.cs

```
app.UseOpenIdConnectAuthentication(
    new OpenIdConnectAuthenticationOptions
    {
        // ...
    });

```

```
app.UseWsFederationAuthentication(
    new WsFederationAuthenticationOptions
    {
        // ...
    });

```

```
app.UseWindowsAzureActiveDirectoryBearerAuthentication(
    new WindowsAzureActiveDirectoryBearerAuthenticationOptions
    {
        // ...
    });

```

Web applications / APIs protecting resources using .NET Core OpenID Connect or JwtBearerAuthentication middleware

If your application is using the .NET Core OWIN OpenID Connect or JwtBearerAuthentication middleware, it already has the necessary logic to handle key rollover automatically.

You can confirm that your application is using any of these by looking for any of the following snippets in your application's Startup.cs or Startup.Auth.cs

```
app.UseOpenIdConnectAuthentication(
    new OpenIdConnectAuthenticationOptions
    {
        // ...
    });

```

```
app.UseJwtBearerAuthentication(
    new JwtBearerAuthenticationOptions
    {
        // ...
    });

```

Web applications / APIs protecting resources using Node.js passport-azure-ad module

If your application is using the Node.js passport-ad module, it already has the necessary logic to handle key rollover automatically.

You can confirm that your application passport-ad by searching for the following snippet in your application's app.js

```
var OIDCStrategy = require('passport-azure-ad').OIDCStrategy;

passport.use(new OIDCStrategy({
    //...
}));

```

Web applications / APIs protecting resources and created with Visual Studio 2015 or later

If your application was built using a web application template in Visual Studio 2015 or later and you selected **Work Or School Accounts** from the **Change Authentication** menu, it already has the necessary logic to handle key rollover automatically. This logic, embedded in the OWIN OpenID Connect middleware, retrieves and caches the keys from the OpenID Connect discovery document and periodically refreshes them.

If you added authentication to your solution manually, your application might not have the necessary key rollover logic. You will need to write it yourself, or follow the steps in [Web applications / APIs using any other libraries or manually implementing any of the supported protocols](#).

Web applications protecting resources and created with Visual Studio 2013

If your application was built using a web application template in Visual Studio 2013 and you selected **Organizational Accounts** from the **Change Authentication** menu, it already has the necessary logic to handle key rollover automatically. This logic stores your organization's unique identifier and the signing key information in two database tables associated with the project. You can find the connection string for the database in the project's Web.config file.

If you added authentication to your solution manually, your application might not have the necessary key rollover logic. You will need to write it yourself, or follow the steps in [Web applications / APIs using any other libraries or manually implementing any of the supported protocols..](#)

The following steps will help you verify that the logic is working properly in your application.

1. In Visual Studio 2013, open the solution, and then click on the **Server Explorer** tab on the right window.
2. Expand **Data Connections**, **DefaultConnection**, and then **Tables**. Locate the **IssuingAuthorityKeys** table, right-click it, and then click **Show Table Data**.
3. In the **IssuingAuthorityKeys** table, there will be at least one row, which corresponds to the thumbprint value for the key. Delete any rows in the table.
4. Right-click the **Tenants** table, and then click **Show Table Data**.
5. In the **Tenants** table, there will be at least one row, which corresponds to a unique directory tenant identifier. Delete any rows in the table. If you don't delete the rows in both the **Tenants** table and **IssuingAuthorityKeys** table, you will get an error at runtime.
6. Build and run the application. After you have logged in to your account, you can stop the application.
7. Return to the **Server Explorer** and look at the values in the **IssuingAuthorityKeys** and **Tenants** table. You'll notice that they have been automatically repopulated with the appropriate information from the federation metadata document.

Web APIs protecting resources and created with Visual Studio 2013

If you created a web API application in Visual Studio 2013 using the Web API template, and then selected **Organizational Accounts** from the **Change Authentication** menu, you already have the necessary logic in your application.

If you manually configured authentication, follow the instructions below to learn how to configure your Web API to automatically update its key information.

The following code snippet demonstrates how to get the latest keys from the federation metadata document, and then use the [JWT Token Handler](#) to validate the token. The code snippet assumes that you will use your own caching mechanism for persisting the key to validate future tokens from Azure AD, whether it be in a database, configuration file, or elsewhere.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IdentityModel.Tokens;
using System.Configuration;
using System.Security.Cryptography.X509Certificates;
using System.Xml;
using System.IdentityModel.Metadata;
using System.ServiceModel.Security;
using System.Threading;

namespace JWTValidation
{
    public class JWTValidator
    {
        private string MetadataAddress = "[Your Federation Metadata document address goes here]";

        // Validates the JWT Token that's part of the Authorization header in an HTTP request.
        public void ValidateJwtToken(string token)
        {
            JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler()
            {
                // Do not disable for production code
                CertificateValidator = X509CertificateValidator.None
            };
        }
    }
}
```

```

        ,
        TokenValidationParameters validationParams = new TokenValidationParameters()
        {
            AllowedAudience = "[Your App ID URI goes here, as registered in the Azure Portal]",
            ValidIssuer = "[The issuer for the token goes here, such as https://sts.windows.net/68b98905-
130e-4d7c-b6e1-a158a9ed8449/]",
            SigningTokens = GetSigningCertificates(MetadataAddress)

            // Cache the signing tokens by your desired mechanism
        };

        Thread.CurrentPrincipal = tokenHandler.ValidateToken(token, validationParams);
    }

    // Returns a list of certificates from the specified metadata document.
    public List<X509SecurityToken> GetSigningCertificates(string metadataAddress)
    {
        List<X509SecurityToken> tokens = new List<X509SecurityToken>();

        if (metadataAddress == null)
        {
            throw new ArgumentNullException(metadataAddress);
        }

        using (XmlReader metadataReader = XmlReader.Create(metadataAddress))
        {
            MetadataSerializer serializer = new MetadataSerializer()
            {
                // Do not disable for production code
                CertificateValidationMode = X509CertificateValidationMode.None
            };

            EntityDescriptor metadata = serializer.ReadMetadata(metadataReader) as EntityDescriptor;

            if (metadata != null)
            {
                SecurityTokenServiceDescriptor stsd =
                metadata.RoleDescriptors.OfType<SecurityTokenServiceDescriptor>().First();

                if (stsd != null)
                {
                    IEnumerable<X509RawDataKeyIdentifierClause> x509DataClauses = stsd.Keys.Where(key =>
key.KeyInfo != null && (key.Use == KeyType.Signing || key.Use == KeyType.Unspecified)).
Select(key =>
key.KeyInfo.OfType<X509RawDataKeyIdentifierClause>().First());

                    tokens.AddRange(x509DataClauses.Select(token => new X509SecurityToken(new
X509Certificate2(token.GetX509RawData()))));
                }
                else
                {
                    throw new InvalidOperationException("There is no RoleDescriptor of type
SecurityTokenServiceType in the metadata");
                }
            }
            else
            {
                throw new Exception("Invalid Federation Metadata document");
            }
        }
        return tokens;
    }
}
}

```

If your application was built in Visual Studio 2012, you probably used the Identity and Access Tool to configure your application. It's also likely that you are using the [Validating Issuer Name Registry \(VINR\)](#). The VINR is responsible for maintaining information about trusted identity providers (Azure AD) and the keys used to validate tokens issued by them. The VINR also makes it easy to automatically update the key information stored in a Web.config file by downloading the latest federation metadata document associated with your directory, checking if the configuration is out of date with the latest document, and updating the application to use the new key as necessary.

If you created your application using any of the code samples or walkthrough documentation provided by Microsoft, the key rollover logic is already included in your project. You will notice that the code below already exists in your project. If your application does not already have this logic, follow the steps below to add it and to verify that it's working correctly.

1. In **Solution Explorer**, add a reference to the **System.IdentityModel** assembly for the appropriate project.
2. Open the **Global.asax.cs** file and add the following using directives:

```
using System.Configuration;
using System.IdentityModel.Tokens;
```

3. Add the following method to the **Global.asax.cs** file:

```
protected void RefreshValidationSettings()
{
    string configPath = AppDomain.CurrentDomain.BaseDirectory + "\\Web.config";
    string metadataAddress =
        ConfigurationManager.AppSettings["ida:FederationMetadataLocation"];
    ValidatingIssuerNameRegistry.WriteToConfig(metadataAddress, configPath);
}
```

4. Invoke the **RefreshValidationSettings()** method in the **Application_Start()** method in **Global.asax.cs** as shown:

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    ...
    RefreshValidationSettings();
}
```

Once you have followed these steps, your application's Web.config will be updated with the latest information from the federation metadata document, including the latest keys. This update will occur every time your application pool recycles in IIS; by default IIS is set to recycle applications every 29 hours.

Follow the steps below to verify that the key rollover logic is working.

1. After you have verified that your application is using the code above, open the **Web.config** file and navigate to the **<issuerNameRegistry>** block, specifically looking for the following few lines:

```
<issuerNameRegistry type="System.IdentityModel.Tokens.ValidatingIssuerNameRegistry,
System.IdentityModel.Tokens.ValidatingIssuerNameRegistry">
    <authority name="https://sts.windows.net/ec4187af-07da-4f01-b18f-64c2f5abecea/">
        <keys>
            <add thumbprint="3A38FA984E8560F19AADC9F86FE9594BB6AD049B" />
        </keys>

```

2. In the **<add thumbprint="">** setting, change the thumbprint value by replacing any character with a different

one. Save the **Web.config** file.

3. Build the application, and then run it. If you can complete the sign-in process, your application is successfully updating the key by downloading the required information from your directory's federation metadata document. If you are having issues signing in, ensure the changes in your application are correct by reading the [Adding Sign-On to Your Web Application Using Azure AD](#) article, or downloading and inspecting the following code sample: [Multi-Tenant Cloud Application for Azure Active Directory](#).

Web applications protecting resources and created with Visual Studio 2008 or 2010 and Windows Identity Foundation (WIF) v1.0 for .NET 3.5

If you built an application on WIF v1.0, there is no provided mechanism to automatically refresh your application's configuration to use a new key.

- *Easiest way* Use the FedUtil tooling included in the WIF SDK, which can retrieve the latest metadata document and update your configuration.
- Update your application to .NET 4.5, which includes the newest version of WIF located in the System namespace. You can then use the [Validating Issuer Name Registry \(VINR\)](#) to perform automatic updates of the application's configuration.
- Perform a manual rollover as per the instructions at the end of this guidance document.

Instructions to use the FedUtil to update your configuration:

1. Verify that you have the WIF v1.0 SDK installed on your development machine for Visual Studio 2008 or 2010. You can [download it from here](#) if you have not yet installed it.
2. In Visual Studio, open the solution, and then right-click the applicable project and select **Update federation metadata**. If this option is not available, FedUtil and/or the WIF v1.0 SDK has not been installed.
3. From the prompt, select **Update** to begin updating your federation metadata. If you have access to the server environment where the application is hosted, you can optionally use FedUtil's [automatic metadata update scheduler](#).
4. Click **Finish** to complete the update process.

Web applications / APIs protecting resources using any other libraries or manually implementing any of the supported protocols

If you are using some other library or manually implemented any of the supported protocols, you'll need to review the library or your implementation to ensure that the key is being retrieved from either the OpenID Connect discovery document or the federation metadata document. One way to check for this is to do a search in your code or the library's code for any calls out to either the OpenID discovery document or the federation metadata document.

If the key is being stored somewhere or hardcoded in your application, you can manually retrieve the key and update it accordingly by performing a manual rollover as per the instructions at the end of this guidance document. **It is strongly encouraged that you enhance your application to support automatic rollover** using any of the approaches outline in this article to avoid future disruptions and overhead if Azure AD increases its rollover cadence or has an emergency out-of-band rollover.

How to test your application to determine if it will be affected

You can validate whether your application supports automatic key rollover by downloading the scripts and following the instructions in [this GitHub repository](#).

How to perform a manual rollover if your application does not support automatic rollover

If your application does **not** support automatic rollover, you will need to establish a process that periodically monitors Azure AD's signing keys and performs a manual rollover accordingly. [This GitHub repository](#) contains

scripts and instructions on how to do this.

Microsoft identity platform ID tokens

10/17/2019 • 7 minutes to read • [Edit Online](#)

`id_tokens` are sent to the client application as part of an [OpenID Connect](#) flow. They can be sent along side or instead of an access token, and are used by the client to authenticate the user.

Using the id_token

ID Tokens should be used to validate that a user is who they claim to be and get additional useful information about them - it shouldn't be used for authorization in place of an [access token](#). The claims it provides can be used for UX inside your application, as keys in a database, and providing access to the client application. When creating keys for a database, `idp` should not be used because it messes up guest scenarios. Keying should be done on `sub` alone (which is always unique), with `tid` used for routing if need be. If you need to share data across services, `oid + sub + tid` will work since multiple services all get the same `oid`.

Claims in an id_token

`id_tokens` for a Microsoft identity are [JWTs](#), meaning they consist of a header, payload, and signature portion. You can use the header and signature to verify the authenticity of the token, while the payload contains the information about the user requested by your client. Except where noted, all claims listed here appear in both v1.0 and v2.0 tokens.

v1.0

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6IjdfWnVmMXR2a3dMeF1hSFMcTZsVWpVWUlHdyIsImtpZCI6IjdfWnVmMXR2a3dMeF1hSFMcTZsVWpVWUlHdyJ9.eyJhdWQiOjIjMTRhNzUwNS05NmU5LTQ5MjctOTF1OCwNjAxZDBmYzljYWEiLCJpc3MiOiJodHRwczovL3N0cy53aW5kb3dzLm51dC9mYTE1ZDY5Mi1l0WM3LTQ0NjAtYTC0My0yOWYyOTU2ZmQ0MjkvIiwiawF0IjoxNTM2Mjc1MTI0LCJuYmYiOjE1MzYyNzUxMjQsImV4cCI6MTUzNjI30TAyNCwiYWlvIjoiQVhRQWkvOE1BQUFBcXhzdUIrUjREMNjGUxFPRVRPNF1kWGJMRD1rWjh4Z1hhZGVBTBRMk5rT1Q1axpmZzN1d2JXU1hodVNTajZVVDFoeTJEN1dxQXBCNwpLQTzaZ1o5ay9TVTI3dVY5Y2V0WGZMT3RwTnR0Z2s1RGNCdGsrTExzdHovSmcrZ11SbXY5Y1VVNFhscGhUYzzDODZKbWoxRkN3PT0iLCJhbXIiOlsicnNhIl0sImVtYwlSijoiYWJ1bG1Abwljcm9zb2Z0LmNvbSIImZhbwlseV9uYW11IjoiTGluy29sbIisImdpdmVuX25hbWUiOjBYmUiLCJpzZHAIoijodHRwczovL3N0cy53aW5kb3dzLm51dC83MmY50DhiZi04NmYxLTQxYWyotOTfHVi0yZDdjZDAXMRiNDcvIiwiXBhZGRyIjoiMTMxLjEwNy4yMjIuMjIiLCJuYW11IjoiYWJ1bGkiLCJub25jZSI6IjEyMzUyMyIsIm9pZCI6IjA10DzYjZiLWFhMWQtNDJkNC05ZWmwLTFiMmJiOTE5NDQzOCIsInJoIjoiSSIsInN1YiI6IjVFSjlyU3NzOC1qdnRFswN1NnV1Uk5MOHhYYjhMRjRGc2dfs29vQzJSS1EiLCJ0aWQioiJmYTE1ZDY5Mi1l0WM3LTQ0NjAtYTC0My0yOWYyOTU2ZmQ0MjkiLCJ1bm1xdWVfbmFtZSI6IkFiZUxpQG1pY3Jvc29mdC5jb20iLCJ1dGkiOjJMeGvfNDZhCrVrT3BHU2ZUbG40RUFBIiwidmVyIjoiMS4wIn0=.UJQrCA6qn2bXq57qzGX_-D3HcPHqBMOKDPx4su1yKRLNErVD8xkxJLNLVRdASHqEcphyDctbdHccu6DPpkq5f0ibcaQFhejQNcABidJCTz0Bb2AbdUCTqAzdt9pdgQvMBnVH1xk3SCM6d4BbT4BkLLj10ZLasX7vRknaSjE_C5DI7Fg4WrZPw0hII1dB0HEZ_qpNaYXEiy-o94UJ94zCr07GgrqMsfyQqFR7kn-mn68AjvLcgwSFZvyR_yIK75S_K37vC3QryQ7cNoafDe9upql_6pB2ybMV1gwPs_DmbJ8g0om-sPlwyn74Cc1tw3ze-Xptw_2uVdPgWYqfuWAfq6Q
```

View this v1.0 sample token in [jwt.ms](#).

v2.0

```

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IjFMVE16YWtpaGlSbGFFoHoyQkVKVlhlV01xbyJ9.eyJ2ZXIiOiIyLjAiLCJpc3MiOi
JodHRwcovL2xvZ2luLm1pY3Jvc29mdG9ubGluZS5jb20vOTEyMjA0MGQtNmM2Ny00YzViLWIxMTItMzzhMzA0YjY2ZGFkL3YyLjAiLCJzdWIiO
iJBQUFBQUFBQUFBQUFBQUFBQUFJa3pxR1ZyU2FTYUZIEtC4MmJidGFRIiwiYXVkJoiNmNiMDQwMTgtYTNmNS00NmE3LWI5OTUtOTQwYzc4
ZjVhZWYzIiwiZXhwIjoxNTM2MzYxNDExLCJpYXQiOjE1MzYyNzQ3MTEsIm5iZiI6MTUzNjI3NDcxMSwibmFtZSI6IkFiZSBMaW5jb2xuIiwichJ
1ZmVycmVx3VzZXJuYW1lIjoiQWJ1TG1AbWljcm9zb2Z0LmNvbSIsIm9pZCI6IjAwMDAwMDAwLTAWMDAtMDAwMC02NmYzLTMzMzJ1Y2E3ZWE4MS
IsInRpZCI6IjkxMjIwNDBkLTzjNjctNGM1Yi1iMTEyLTM2YTMwNGI2NmRhZCIsIm5vbmlNlIjoiMTIzNTIzIiwiYwlvIjoiRGYyVVZYTDFpeCFsT
UNXTVNPskJjRmF0emNHZnZGR2hqS3Y4cTVnMHg3MzJkUjVNQjVCaN2R1FPN11XQn1qZDhpUURMcSF1R2JJRGFreXA1bW5PcmNkcUhlWVNubHR1
cFFtUnA2QU1aOGpZIn0.1AFWW-Ck5nROwS1ltm7GzZvDwUkqvhsQpm55TQsmVo9Y59cLhRXpvB8n-
55HCr9Z6G_31_UbeUkoz612I2j_Sm9FFShSDDjoaLQr54CreGIJvjtms3EkK9a7SJBBcpL1MpUt1fygow39tFjY7EVNW9p1WUvRrTgVk71YLprv
fzw-CIqw3gHC-T7IK_m_xkr08INERBtaecwhTeN4chPC4W3jdmw_lIxzc48YoQ0dB1L9-
Im98Egypfr1bm0IBL5spFzL6JDZIRRJ0u8vecJvj1mq-IUhGt0MacxX8jdxYLP-KUu2d9MbNkpkCKJuZ7p8gwTL5B7NlUdh_dmSviPWrw

```

View this v2.0 sample token in jwt.ms.

Header claims

| CLAIM | FORMAT | DESCRIPTION |
|-------|-----------------------|---|
| typ | String - always "JWT" | Indicates that the token is a JWT. |
| alg | String | Indicates the algorithm that was used to sign the token. Example: "RS256" |
| kid | String | Thumbprint for the public key used to sign this token. Emitted in both v1.0 and v2.0 <code>id_tokens</code> . |
| x5t | String | The same (in use and value) as <code>kid</code> . However, this is a legacy claim emitted only in v1.0 <code>id_tokens</code> for compatibility purposes. |

Payload claims

This list shows the claims that are in most id_tokens by default (except where noted). However, your app can use [optional claims](#) to request additional claims in the id_token. These can range from the `groups` claim to information about the user's name.

| CLAIM | FORMAT | DESCRIPTION |
|-------|-----------------------|---|
| aud | String, an App ID URI | Identifies the intended recipient of the token. In <code>id_tokens</code> , the audience is your app's Application ID, assigned to your app in the Azure portal. Your app should validate this value, and reject the token if the value does not match. |

| CLAIM | FORMAT | DESCRIPTION |
|---------------------|----------------------------|---|
| <code>iss</code> | String, an STS URI | <p>Identifies the security token service (STS) that constructs and returns the token, and the Azure AD tenant in which the user was authenticated. If the token was issued by the v2.0 endpoint, the URI will end in <code>/v2.0</code>. The GUID that indicates that the user is a consumer user from a Microsoft account is</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;"> 9188040d-6c67-4c5b-b112-36a304b66dad </div> <p>Your app should use the GUID portion of the claim to restrict the set of tenants that can sign in to the app, if applicable.</p> |
| <code>iat</code> | int, a UNIX timestamp | "Issued At" indicates when the authentication for this token occurred. |
| <code>idp</code> | String, usually an STS URI | <p>Records the identity provider that authenticated the subject of the token. This value is identical to the value of the Issuer claim unless the user account not in the same tenant as the issuer - guests, for instance. If the claim isn't present, it means that the value of <code>iss</code> can be used instead. For personal accounts being used in an organizational context (for instance, a personal account invited to an Azure AD tenant), the <code>idp</code> claim may be 'live.com' or an STS URI containing the Microsoft account tenant</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;"> 9188040d-6c67-4c5b-b112-36a304b66dad </div> |
| <code>nbf</code> | int, a UNIX timestamp | The "nbf" (not before) claim identifies the time before which the JWT MUST NOT be accepted for processing. |
| <code>exp</code> | int, a UNIX timestamp | The "exp" (expiration time) claim identifies the expiration time on or after which the JWT MUST NOT be accepted for processing. It's important to note that a resource may reject the token before this time as well - if, for example, a change in authentication is required or a token revocation has been detected. |
| <code>c_hash</code> | String | The code hash is included in ID tokens only when the ID token is issued with an OAuth 2.0 authorization code. It can be used to validate the authenticity of an authorization code. For details about performing this validation, see the OpenID Connect specification . |

| CLAIM | FORMAT | DESCRIPTION |
|--------------------|---------------|--|
| at_hash | String | The access token hash is included in ID tokens only when the ID token is issued with an OAuth 2.0 access token. It can be used to validate the authenticity of an access token. For details about performing this validation, see the OpenID Connect specification . |
| aio | Opaque String | An internal claim used by Azure AD to record data for token reuse. Should be ignored. |
| preferred_username | String | The primary username that represents the user. It could be an email address, phone number, or a generic username without a specified format. Its value is mutable and might change over time. Since it is mutable, this value must not be used to make authorization decisions. The <code>profile</code> scope is required to receive this claim. |
| email | String | The <code>email</code> claim is present by default for guest accounts that have an email address. Your app can request the <code>email</code> claim for managed users (those from the same tenant as the resource) using the email optional claim . On the v2.0 endpoint, your app can also request the <code>email</code> OpenID Connect scope - you don't need to request both the optional claim and the scope to get the claim. The <code>email</code> claim only supports addressable mail from the user's profile information. |
| name | String | The <code>name</code> claim provides a human-readable value that identifies the subject of the token. The value isn't guaranteed to be unique, it is mutable, and it's designed to be used only for display purposes. The <code>profile</code> scope is required to receive this claim. |
| nonce | String | The nonce matches the parameter included in the original /authorize request to the IDP. If it does not match, your application should reject the token. |

| CLAIM | FORMAT | DESCRIPTION |
|--------------------------|------------------|---|
| <code>oid</code> | String, a GUID | The immutable identifier for an object in the Microsoft identity system, in this case, a user account. This ID uniquely identifies the user across applications - two different applications signing in the same user will receive the same value in the <code>oid</code> claim. The Microsoft Graph will return this ID as the <code>id</code> property for a given user account. Because the <code>oid</code> allows multiple apps to correlate users, the <code>profile</code> scope is required to receive this claim. Note that if a single user exists in multiple tenants, the user will contain a different object ID in each tenant - they're considered different accounts, even though the user logs into each account with the same credentials. The <code>oid</code> claim is a GUID and cannot be reused. |
| <code>roles</code> | Array of strings | The set of roles that were assigned to the user who is logging in. |
| <code>rh</code> | Opaque String | An internal claim used by Azure to revalidate tokens. Should be ignored. |
| <code>sub</code> | String, a GUID | The principal about which the token asserts information, such as the user of an app. This value is immutable and cannot be reassigned or reused. The subject is a pairwise identifier - it is unique to a particular application ID. If a single user signs into two different apps using two different client IDs, those apps will receive two different values for the subject claim. This may or may not be wanted depending on your architecture and privacy requirements. |
| <code>tid</code> | String, a GUID | A GUID that represents the Azure AD tenant that the user is from. For work and school accounts, the GUID is the immutable tenant ID of the organization that the user belongs to. For personal accounts, the value is
<code>9188040d-6c67-4c5b-b112-36a304b66dad</code> . The <code>profile</code> scope is required to receive this claim. |
| <code>unique_name</code> | String | Provides a human readable value that identifies the subject of the token. This value isn't guaranteed to be unique within a tenant and should be used only for display purposes. Only issued in v1.0 <code>id_tokens</code> . |

| CLAIM | FORMAT | DESCRIPTION |
|-------|---------------------------|--|
| uti | Opaque String | An internal claim used by Azure to revalidate tokens. Should be ignored. |
| ver | String, either 1.0 or 2.0 | Indicates the version of the id_token. |

Validating an id_token

Validating an `id_token` is similar to the first step of [validating an access token](#) - your client should validate that the correct issuer has sent back the token and that it hasn't been tampered with. Because `id_tokens` are always a JWT, many libraries exist to validate these tokens - we recommend you use one of these rather than doing it yourself.

To manually validate the token, see the steps details in [validating an access token](#). After validating the signature on the token, the following claims should be validated in the `id_token` (these may also be done by your token validation library):

- Timestamps: the `iat`, `nbf`, and `exp` timestamps should all fall before or after the current time, as appropriate.
- Audience: the `aud` claim should match the app ID for your application.
- Nonce: the `nonce` claim in the payload must match the nonce parameter passed into the /authorize endpoint during the initial request.

Next steps

- Learn about [access tokens](#)
- Customize the claims in your `id_token` using [optional claims](#).

Microsoft identity platform access tokens

11/12/2019 • 20 minutes to read • [Edit Online](#)

Access tokens enable clients to securely call APIs protected by Azure. Microsoft identity platform access tokens are [JWTs](#), Base64 encoded JSON objects signed by Azure. Clients should treat access tokens as opaque strings, as the contents of the token are intended for the resource only. For validation and debugging purposes, developers can decode JWTs using a site like [jwt.ms](#). Your client can get an access token from either the v1.0 endpoint or the v2.0 endpoint using a variety of protocols.

When your client requests an access token, Azure AD also returns some metadata about the access token for your app's consumption. This information includes the expiry time of the access token and the scopes for which it's valid. This data allows your app to do intelligent caching of access tokens without having to parse the access token itself.

If your application is a resource (web API) that clients can request access to, access tokens provide helpful information for use in authentication and authorization, such as the user, client, issuer, permissions, and more.

See the following sections to learn how a resource can validate and use the claims inside an access token.

IMPORTANT

Access tokens are created based on the *audience* of the token, meaning the application that owns the scopes in the token. This is how a resource setting `accessTokenAcceptedVersion` in the [app manifest](#) to `2` allows a client calling the v1.0 endpoint to receive a v2.0 access token. Similarly, this is why changing the access token [optional claims](#) for your client do not change the access token received when a token is requested for `user.read`, which is owned by the MS Graph resource.

For the same reason, while testing your client application with a personal account (such as [hotmail.com](#) or [outlook.com](#)), you may find that the access token received by your client is an opaque string. This is because the resource being accessed has requested legacy MSA (Microsoft account) tickets that are encrypted and can't be understood by the client.

Sample tokens

v1.0 and v2.0 tokens look similar and contain many of the same claims. An example of each is provided here.

v1.0

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Imk2bEdrM0ZaenhsY1ViMkMzbkVRN3N5SEpsWSIisImtpZC16Imk2bEdrM0ZaenhsY1ViMkMzbkVRN3N5SEpsWSJ9.eyJhdWQioiJzjFkYTlknc1mZjc3LTrjM2UtYTAwNS04NDBjM2Y4MzA3NDUiLCJpc3MiOjJodHRwczoVl3N0cy53aW5kb3dzLm5ldC9mYTE1ZD5M11l0Wm3LTQ0NjAtYTc0My0yOWYyOTUyMjIyOS8iLCJpYXQoIjE1MzcymZmxMDYsIm5iZlI6MTUzNzIzMzEwNiwiZkhwIjoxNTM3MjM3MDA2LCJhY3I0iIxIiwiYv1vIjoiQvhRQWkvOE1BQUFRBrm0rRS9RVEcRz0ZuVnhMa1dkdzhLKzxyQudy0091T1UGNmV1YU1qN1hPM0libuQzKdtkc5Rct0dlp5R24yVmFUL2tEs1h3Ne1jAhnR1ZxNkJU0hMdWG9UMUXrSVorRnpRvmrKUFBMUU9NETjwHFTBENNUERTL0RpQ0RnRTIyM1RjbU12V05hRU1hVU9Uc01Hd1RRPT0iLCJhXiolsid2lH10sImFwcG1kIjoiNzVkyMnU3N2YtMTbMy00ZTU5LTg1ZmQt0GMxmjc1NDRmMTdj1iwiYXBwaWRhY3Ii0iIwIiwiZWh1haWw1jByVmVMaUbtaNyb3NvZnQuY29tIiwiZmftawx5X25hbWUoIjMaW5jb2xuIiwiZ212Zw5fbmfTSI6IkFizSAoTVNGVCKiLCJpZHAi0iJodHRwczoVl3N0cy53aW5kb3dzLm5ldC83MmY50DhiZi04NmYxLTQxVWyt0Tfhyi0yZDjdZDAxMjIyNdcviIwiaXbhZGRyIjoiMjIyLjIyMi4yMjIuMjIiLCjzW1lIjoiYwJlbGkiLCJvaWQioiIwMjIyM2I2Yi1hYTFLkTQyZDQt0WVjMC0xYjIyjkx0TQ0MzgilCjyaC16IkklCzY3Ai0iJc2Vx2ltcGVyc29uXRPb24iLCJzdWlIi0IjsM19y01TUvuyMjJiVuTOXlpMmswHxcE9pTx01SDNaQUNvMu1WEEiLCJ0aWQioiJmYTE1ZDy5M11l0Wm3LTQ0NjAtYTc0My0yOWYyOTU2ZmQ0MjkiLCJ1bm1xdWfbumFtZSI6ImFizWxpQG1pY3Jvc29mdC5jb20iLCJ1dGki0iJGvnNHeF1YSTMwLVR1aWt1dVVrKFBiiwidmVijoiMS4wIn0.D3H6pMuTQnoJAGq6AHd
```

View this v1.0 token in [JWT.ms](#).

v2.0

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Imk2bEdrM0ZaenhsY1ViMkMzbkVRN3N5SEpsWSJ9.eyJhdWQioiJzTc0MTcyYi1iZTU2LTQ4NDMt0WzmNC11NjZhMzliYjEyZTMiLCJpc3MiOjJodHRwczoVl2xvZ2luLm1pY3Jvc29mgD9ubGluzS5jb20vNzJmOTg4YmYt0DzmMS00MWFmLTkxYWItdMmQ3Y2QwMTFkYjQ3L3YyLjAiLCJpYXQoIjE1MzcymZmEwNDgsIm5iZlI6MTUzNzIzMTA0OCwiZxhwIjoxNTM3MjM00TQ4LCJhalw8i0iJBWFFBaS84SUFBQUF0QWfaTG8zQ2hNaWY2S09udHRSOjdlqnE0L0RjY1F6amNKR3hQWxkvQzNqRGFOR3hYZDZ3Tk1JVkdSz2h0Um53SjFsT2NBbk5aY2p2a295ckZ4Q3R0djMzMTQwUmlvT0ZKNGJDQ0dWdw9DwCxdU9UDIyMjIyZoh3TFBZUS91ZjcsUVgrMetJawpkcm1wNj1SY3R6bVE9PS1sImF6cIC6Ijz1NzQxNzjilWJ1NTYNDg0My052mY0LwU2NmEz0WjIyMTJ1MyIsImF6cGFjci6IjAiLCJyW1lIjoiQWJlIExpbnvbg4iLCJvaWQioi20TAyMjJiZs1mZjfhLTRKNTYtWJYkMS03ZTRmN2QzOGU0NzQjilCJwcmVmZxJyZwRfdXN1cm5hbWUoIjhYmVsabtaWNYb3NvZnQuY29tIiwiicgi0iJJiwi2NwIjoiYWNjZXNzX2FzX3VzZXiiLCJzdWIioiJIS1pwZmFieVdhZGVPb3VzbG10anJjlUtmZ1RtMjIyWDVyc1YzeRxZktRiwiidG1kIjoiNzJm0Tg4YmYt0DzmMS00MWFmLTkxYWItdMmQ3Y2QwMTFkYjQ3IiwidXRpIjoiZnfpQnFYTFBqMGVRYTgyUy1JwUZBQSiisInZ1ciI6IjIuMCJ9.pj4N-w_3Us9DrBLfpCt
```

View this v2.0 token in [JWT.ms](#).

Claims in access tokens

JWTs are split into three pieces:

- **Header** - Provides information about how to [validate the token](#) including information about the type of token and how it was signed.
- **Payload** - Contains all of the important data about the user or app that is attempting to call your service.
- **Signature** - Is the raw material used to validate the token.

Each piece is separated by a period (.) and separately Base64 encoded.

Claims are present only if a value exists to fill it. So, your app shouldn't take a dependency on a claim being present. Examples include `pwd_exp` (not every tenant requires passwords to expire) or `family_name` ([client credential](#) flows are on behalf of applications, which don't have names). Claims used for access token validation will always be present.

NOTE

Some claims are used to help Azure AD secure tokens in case of reuse. These are marked as not being for public consumption in the description as "Opaque". These claims may or may not appear in a token, and new ones may be added without notice.

Header claims

| CLAIM | FORMAT | DESCRIPTION |
|--------------------|-----------------------|---|
| <code>typ</code> | String - always "JWT" | Indicates that the token is a JWT. |
| <code>nonce</code> | String | A unique identifier used to protect against token replay attacks. Your resource can record this value to protect against replays. |
| <code>alg</code> | String | Indicates the algorithm that was used to sign the token, for example, "RS256" |
| <code>kid</code> | String | Specifies the thumbprint for the public key that's used to sign this token. Emitted in both v1.0 and v2.0 access tokens. |
| <code>x5t</code> | String | Functions the same (in use and value) as <code>kid</code> . <code>x5t</code> is a legacy claim emitted only in v1.0 access tokens for compatibility purposes. |

Payload claims

| CLAIM | FORMAT | DESCRIPTION |
|------------------|----------------------------|--|
| <code>aud</code> | String, an App ID URI | Identifies the intended recipient of the token. In id tokens, the audience is your app's Application ID, assigned to your app in the Azure portal. Your app should validate this value and reject the token if the value does not match. |
| <code>iss</code> | String, an STS URI | Identifies the security token service (STS) that constructs and returns the token, and the Azure AD tenant in which the user was authenticated. If the token issued is a v2.0 token (see the <code>ver</code> claim), the URI will end in <code>/v2.0</code> . The GUID that indicates that the user is a consumer user from a Microsoft account is <code>9188040d-6c67-4c5b-b112-36a304b66dad</code> . Your app should use the GUID portion of the claim to restrict the set of tenants that can sign in to the app, if applicable. |
| <code>idp</code> | String, usually an STS URI | Records the identity provider that authenticated the subject of the token. This value is identical to the value of the Issuer claim unless the user account not in the same tenant as the issuer - guests, for instance. If the claim isn't present, it means that the value of <code>iss</code> can be used instead. For personal accounts being used in an organizational context (for instance, a personal account invited to an Azure AD tenant), the <code>idp</code> claim may be 'live.com' or an STS URI containing the Microsoft account tenant <code>9188040d-6c67-4c5b-b112-36a304b66dad</code> . |
| <code>iat</code> | int, a UNIX timestamp | "Issued At" indicates when the authentication for this token occurred. |

| CLAIM | FORMAT | DESCRIPTION |
|---------------------------------|-----------------------|--|
| <code>nbf</code> | int, a UNIX timestamp | The "nbf" (not before) claim identifies the time before which the JWT must not be accepted for processing. |
| <code>exp</code> | int, a UNIX timestamp | The "exp" (expiration time) claim identifies the expiration time on or after which the JWT must not be accepted for processing. It's important to note that a resource may reject the token before this time as well, such as when a change in authentication is required or a token revocation has been detected. |
| <code>aio</code> | Opaque String | An internal claim used by Azure AD to record data for token reuse. Resources should not use this claim. |
| <code>acr</code> | String, a "0" or "1" | Only present in v1.0 tokens. The "Authentication context class" claim. A value of "0" indicates the end-user authentication did not meet the requirements of ISO/IEC 29115. |
| <code>amr</code> | JSON array of strings | Only present in v1.0 tokens. Identifies how the subject of the token was authenticated. See the amr claim section for more details. |
| <code>appid</code> | String, a GUID | Only present in v1.0 tokens. The application ID of the client using the token. The application can act as itself or on behalf of a user. The application ID typically represents an application object, but it can also represent a service principal object in Azure AD. |
| <code>appidacr</code> | "0", "1", or "2" | Only present in v1.0 tokens. Indicates how the client was authenticated. For a public client, the value is "0". If client ID and client secret are used, the value is "1". If a client certificate was used for authentication, the value is "2". |
| <code>azp</code> | String, a GUID | Only present in v2.0 tokens, a replacement for <code>appid</code> . The application ID of the client using the token. The application can act as itself or on behalf of a user. The application ID typically represents an application object, but it can also represent a service principal object in Azure AD. |
| <code>azpacr</code> | "0", "1", or "2" | Only present in v2.0 tokens, a replacement for <code>appidacr</code> . Indicates how the client was authenticated. For a public client, the value is "0". If client ID and client secret are used, the value is "1". If a client certificate was used for authentication, the value is "2". |
| <code>preferred_username</code> | String | The primary username that represents the user. It could be an email address, phone number, or a generic username without a specified format. Its value is mutable and might change over time. Since it is mutable, this value must not be used to make authorization decisions. It can be used for username hints though. The <code>profile</code> scope is required in order to receive this claim. |
| <code>name</code> | String | Provides a human-readable value that identifies the subject of the token. The value is not guaranteed to be unique, it is mutable, and it's designed to be used only for display purposes. The <code>profile</code> scope is required in order to receive this claim. |

| CLAIM | FORMAT | DESCRIPTION |
|--------------------------|---|--|
| <code>scp</code> | String, a space separated list of scopes | The set of scopes exposed by your application for which the client application has requested (and received) consent. Your app should verify that these scopes are valid ones exposed by your app, and make authorization decisions based on the value of these scopes. Only included for user tokens . |
| <code>roles</code> | Array of strings, a list of permissions | The set of permissions exposed by your application that the requesting application or user has been given permission to call. For application tokens , this is used during the client-credentials flow in place of user scopes. For user tokens this is populated with the roles the user was assigned to on the target application. |
| <code>wids</code> | Array of RoleTemplateID GUIDs | Denotes the tenant-wide roles assigned to this user, from the section of roles present in the admin roles page . This claim is configured on a per-application basis, through the <code>groupMembershipClaims</code> property of the application manifest . Setting it to "All" or "DirectoryRole" is required. May not be present in tokens obtained through the implicit flow due to token length concerns. |
| <code>groups</code> | JSON array of GUIDs | <p>Provides object IDs that represent the subject's group memberships. These values are unique (see Object ID) and can be safely used for managing access, such as enforcing authorization to access a resource. The groups included in the groups claim are configured on a per-application basis, through the <code>groupMembershipClaims</code> property of the application manifest. A value of null will exclude all groups, a value of "SecurityGroup" will include only Active Directory Security Group memberships, and a value of "All" will include both Security Groups and Office 365 Distribution Lists.</p> <p>See the <code>hasgroups</code> claim below for details on using the <code>groups</code> claim with the implicit grant. For other flows, if the number of groups the user is in goes over a limit (150 for SAML, 200 for JWT), then an overage claim will be added to the claim sources pointing at the AAD Graph endpoint containing the list of groups for the user.</p> |
| <code>hasgroups</code> | Boolean | If present, always <code>true</code> , denoting the user is in at least one group. Used in place of the <code>groups</code> claim for JWTs in implicit grant flows if the full groups claim would extend the URI fragment beyond the URL length limits (currently 6 or more groups). Indicates that the client should use the Graph to determine the user's groups (https://graph.windows.net/{tenantID}/users/{userID}/). |
| <code>groups:src1</code> | JSON object | <p>For token requests that are not length limited (see <code>hasgroups</code> above) but still too large for the token, a link to the full groups list for the user will be included. For JWTs as a distributed claim, for SAML as a new claim in place of the <code>groups</code> claim.</p> <p>Example JWT Value:</p> <pre>"groups":"src1" "_claim_sources : "src1" : { "endpoint" : "https://graph.windows.net/{tenantID}/users/{userID} "}</pre> |

| CLAIM | FORMAT | DESCRIPTION |
|--------------------------|---|---|
| <code>sub</code> | String, a GUID | The principal about which the token asserts information, such as the user of an app. This value is immutable and cannot be reassigned or reused. It can be used to perform authorization checks safely, such as when the token is used to access a resource, and can be used as a key in database tables. Because the subject is always present in the tokens that Azure AD issues, we recommend using this value in a general-purpose authorization system. The subject is, however, a pairwise identifier - it is unique to a particular application ID. Therefore, if a single user signs into two different apps using two different client IDs, those apps will receive two different values for the subject claim. This may or may not be desired depending on your architecture and privacy requirements. See also the <code>oid</code> claim (which does remain the same across apps within a tenant). |
| <code>oid</code> | String, a GUID | The immutable identifier for an object in the Microsoft identity platform, in this case, a user account. It can also be used to perform authorization checks safely and as a key in database tables. This ID uniquely identifies the user across applications - two different applications signing in the same user will receive the same value in the <code>oid</code> claim. Thus, <code>oid</code> can be used when making queries to Microsoft online services, such as the Microsoft Graph. The Microsoft Graph will return this ID as the <code>id</code> property for a given user account . Because the <code>oid</code> allows multiple apps to correlate users, the <code>profile</code> scope is required in order to receive this claim. Note that if a single user exists in multiple tenants, the user will contain a different object ID in each tenant - they are considered different accounts, even though the user logs into each account with the same credentials. |
| <code>tid</code> | String, a GUID | Represents the Azure AD tenant that the user is from. For work and school accounts, the GUID is the immutable tenant ID of the organization that the user belongs to. For personal accounts, the value is <code>9188040d-6c67-4c5b-b112-36a304b66dad</code> . The <code>profile</code> scope is required in order to receive this claim. |
| <code>unique_name</code> | String | Only present in v1.0 tokens. Provides a human readable value that identifies the subject of the token. This value is not guaranteed to be unique within a tenant and should be used only for display purposes. |
| <code>uti</code> | Opaque String | An internal claim used by Azure to revalidate tokens. Resources shouldn't use this claim. |
| <code>rh</code> | Opaque String | An internal claim used by Azure to revalidate tokens. Resources should not use this claim. |
| <code>ver</code> | String, either <code>1.0</code> or <code>2.0</code> | Indicates the version of the access token. |

NOTE

Groups coverage claim

To ensure that the token size doesn't exceed HTTP header size limits, Azure AD limits the number of object IDs that it includes in the groups claim. If a user is member of more groups than the coverage limit (150 for SAML tokens, 200 for JWT tokens), then Azure AD does not emit the groups claim in the token. Instead, it includes an coverage claim in the token that indicates to the application to query the Graph API to retrieve the user's group membership.

```
{
  ...
  "_claim_names": {
    "groups": "src1"
  },
  {
    "_claim_sources": {
      "src1": {
        "endpoint": "[Graph Url to get this user's group membership from]"
      }
    }
  }
  ...
}
```

You can use the `BulkCreateGroups.ps1` provided in the [App Creation Scripts](#) folder to help test overage scenarios.

v1.0 basic claims

The following claims will be included in v1.0 tokens if applicable, but aren't included in v2.0 tokens by default. If you're using v2.0 and need one of these claims, request them using [optional claims](#).

| CLAIM | FORMAT | DESCRIPTION |
|--------------------------|-----------------------|---|
| <code>ipaddr</code> | String | The IP address the user authenticated from. |
| <code>onprem_sid</code> | String, in SID format | In cases where the user has an on-premises authentication, this claim provides their SID. You can use <code>onprem_sid</code> for authorization in legacy applications. |
| <code>pwd_exp</code> | int, a UNIX timestamp | Indicates when the user's password expires. |
| <code>pwd_url</code> | String | A URL where users can be sent to reset their password. |
| <code>in_corp</code> | boolean | Signals if the client is logging in from the corporate network. If they aren't, the claim isn't included. |
| <code>nickname</code> | String | An additional name for the user, separate from first or last name. |
| <code>family_name</code> | String | Provides the last name, surname, or family name of the user as defined on the user object. |
| <code>given_name</code> | String | Provides the first or given name of the user, as set on the user object. |
| <code>upn</code> | String | The username of the user. May be a phone number, email address, or unformatted string. Should only be used for display purposes and providing username hints in reauthentication scenarios. |

The `amr` claim

Microsoft identities can authenticate in different ways, which may be relevant to your application. The `amr` claim is an array that can contain multiple items, such as `["mfa", "rsa", "pwd"]`, for an authentication that used both a password and the Authenticator app.

| VALUE | DESCRIPTION |
|------------------|--|
| <code>pwd</code> | Password authentication, either a user's Microsoft password or an app's client secret. |
| <code>rsa</code> | Authentication was based on the proof of an RSA key, for example with the Microsoft Authenticator app . This includes if authentication was done by a self-signed JWT with a service owned X509 certificate. |
| <code>otp</code> | One-time passcode using an email or a text message. |

| VALUE | DESCRIPTION |
|----------|--|
| fed | A federated authentication assertion (such as JWT or SAML) was used. |
| wia | Windows Integrated Authentication |
| mfa | Multi-factor authentication was used. When this is present the other authentication methods will also be included. |
| ngcmfa | Equivalent to <code>mfa</code> , used for provisioning of certain advanced credential types. |
| wiaormfa | The user used Windows or an MFA credential to authenticate. |
| none | No authentication was done. |

Validating tokens

To validate an `id_token` or an `access_token`, your app should validate both the token's signature and the claims. To validate access tokens, your app should also validate the issuer, the audience, and the signing tokens. These need to be validated against the values in the OpenID discovery document. For example, the tenant-independent version of the document is located at <https://login.microsoftonline.com/common/.well-known/openid-configuration>.

The Azure AD middleware has built-in capabilities for validating access tokens, and you can browse through our [samples](#) to find one in the language of your choice. For more information on how to explicitly validate a JWT token, see the [manual JWT validation sample](#).

We provide libraries and code samples that show how to easily handle token validation. The below information is provided for those who wish to understand the underlying process. There are also several third-party open-source libraries available for JWT validation - there is at least one option for almost every platform and language out there. For more information about Azure AD authentication libraries and code samples, see [v1.0 authentication libraries](#) and [v2.0 authentication libraries](#).

Validating the signature

A JWT contains three segments, which are separated by the `.` character. The first segment is known as the **header**, the second as the **body**, and the third as the **signature**. The signature segment can be used to validate the authenticity of the token so that it can be trusted by your app.

Tokens issued by Azure AD are signed using industry standard asymmetric encryption algorithms, such as RSA 256. The header of the JWT contains information about the key and encryption method used to sign the token:

```
{
  "typ": "JWT",
  "alg": "RS256",
  "x5t": "iBjL1Rcqzhiy4fpIXxdZqohM2Yk",
  "kid": "iBjL1Rcqzhiy4fpIXxdZqohM2Yk"
}
```

The `alg` claim indicates the algorithm that was used to sign the token, while the `kid` claim indicates the particular public key that was used to validate the token.

At any given point in time, Azure AD may sign an `id_token` using any one of a certain set of public-private key pairs. Azure AD rotates the possible set of keys on a periodic basis, so your app should be written to handle those key changes automatically. A reasonable frequency to check for updates to the public keys used by Azure AD is every 24 hours.

You can acquire the signing key data necessary to validate the signature by using the [OpenID Connect metadata document](#) located at:

```
https://login.microsoftonline.com/common/v2.0/.well-known/openid-configuration
```

TIP

Try this [URL](#) in a browser!

This metadata document:

- Is a JSON object containing several useful pieces of information, such as the location of the various endpoints required for doing OpenID Connect authentication.
- Includes a `jwks_uri`, which gives the location of the set of public keys used to sign tokens. The JSON Web Key (JWK) located at the

`jwks_uri` contains all of the public key information in use at that particular moment in time. The JWK format is described in [RFC 7517](#). Your app can use the `kid` claim in the JWT header to select which public key in this document has been used to sign a particular token. It can then do signature validation using the correct public key and the indicated algorithm.

NOTE

The v1.0 endpoint returns both the `x5t` and `kid` claims, while the v2.0 endpoint responds with only the `kid` claim. Going forward, we recommend using the `kid` claim to validate your token.

Doing signature validation is outside the scope of this document - there are many open source libraries available for helping you do so if necessary. However, the Microsoft Identity platform has one token signing extension to the standards - custom signing keys.

If your app has custom signing keys as a result of using the [claims-mapping](#) feature, you must append an `appid` query parameter containing the app ID to get a `jwks_uri` pointing to your app's signing key information, which should be used for validation. For example:

`https://login.microsoftonline.com/{tenant}/.well-known/openid-configuration?appid=6731de76-14a6-49ae-97bc-6eba6914391e` contains a `jwks_uri` of `https://login.microsoftonline.com/{tenant}/discovery/keys?appid=6731de76-14a6-49ae-97bc-6eba6914391e`.

Claims based authorization

Your application's business logic will dictate this step, some common authorization methods are laid out below.

- Check the `scp` or `roles` claim to verify that all present scopes match those exposed by your API, and allow the client to do the requested action.
- Ensure the calling client is allowed to call your API using the `appid` claim.
- Validate the authentication status of the calling client using `appidacr` - it shouldn't be 0 if public clients aren't allowed to call your API.
- Check against a list of past `nonce` claims to verify the token isn't being replayed.
- Check that the `tid` matches a tenant that is allowed to call your API.
- Use the `acr` claim to verify the user has performed MFA. This should be enforced using [Conditional Access](#).
- If you've requested the `roles` or `groups` claims in the access token, verify that the user is in the group allowed to do this action.
 - For tokens retrieved using the implicit flow, you'll likely need to query the [Microsoft Graph](#) for this data, as it's often too large to fit in the token.

User and application tokens

Your application may receive tokens on behalf of a user (the usual flow) or directly from an application (through the [client credentials flow](#)). These app-only tokens indicate that this call is coming from an application and does not have a user backing it. These tokens are handled largely the same, with some differences:

- App-only tokens will not have a `scp` claim, and may instead have a `roles` claim. This is where application permission (as opposed to delegated permissions) will be recorded. For more information about delegated and application permissions, see permission and consent in [v1.0](#) and [v2.0](#).
- Many human-specific claims will be missing, such as `name` or `upn`.
- The `sub` and `oid` claims will be the same.

Token revocation

Refresh tokens can be invalidated or revoked at any time, for different reasons. These fall into two main categories: timeouts and revocations.

Token timeouts

- `MaxInactiveTime`: If the refresh token hasn't been used within the time dictated by the `MaxInactiveTime`, the Refresh Token will no longer be valid.
- `MaxSessionAge`: If `MaxAgeSessionMultiFactor` or `MaxAgeSessionSingleFactor` have been set to something other than their default (Until-revoked), then reauthentication will be required after the time set in the `MaxAgeSession*` elapses.
- Examples:
 - The tenant has a `MaxInactiveTime` of five days, and the user went on vacation for a week, and so Azure AD hasn't seen a new token request from the user in 7 days. The next time the user requests a new token, they'll find their Refresh Token has been revoked, and they must enter their credentials again.
 - A sensitive application has a `MaxAgeSessionSingleFactor` of one day. If a user logs in on Monday, and on Tuesday (after 25 hours have elapsed), they'll be required to reauthenticate.

Revocation

| | PASSWORD-BASED COOKIE | PASSWORD-BASED TOKEN | NON-PASSWORD-BASED COOKIE | NON-PASSWORD-BASED TOKEN | CONFIDENTIAL CLIENT TOKEN |
|--|-----------------------|----------------------|---------------------------|--------------------------|---------------------------|
| Password expires | Stays alive | Stays alive | Stays alive | Stays alive | Stays alive |
| Password changed by user | Revoked | Revoked | Stays alive | Stays alive | Stays alive |
| User does SSPR | Revoked | Revoked | Stays alive | Stays alive | Stays alive |
| Admin resets password | Revoked | Revoked | Stays alive | Stays alive | Stays alive |
| User revokes their refresh tokens via PowerShell | Revoked | Revoked | Revoked | Revoked | Revoked |
| Admin revokes all refresh tokens for the tenant via PowerShell | Revoked | Revoked | Revoked | Revoked | Revoked |
| Single sign-out on web | Revoked | Stays alive | Revoked | Stays alive | Stays alive |

NOTE

A "Non-password based" login is one where the user didn't type in a password to get it. For example, using your face with Windows Hello, a FIDO2 key, or a PIN.

Primary Refresh Tokens (PRT) on Windows 10 are segregated based on the credential. For example, Windows Hello and password have their respective PRTs, isolated from one another. When a user signs-in with a Hello credential (PIN or biometrics) and then changes the password, the password based PRT obtained previously will be revoked. Signing back in with a password invalidates the old PRT and requests a new one.

Refresh tokens aren't invalidated or revoked when used to fetch a new access token and refresh token.

Next steps

- Learn about [id_tokens](#) in Azure AD.
- Learn about permission and consent in [v1.0](#) and [v2.0](#).

Certificate credentials for application authentication

11/4/2019 • 3 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) allows an application to use its own credentials for authentication, for example, in the OAuth 2.0 Client Credentials Grant flow ([v1.0](#), [v2.0](#)) and the On-Behalf-Of flow ([v1.0](#), [v2.0](#)).

One form of credential that an application can use for authentication is a JSON Web Token(JWT) assertion signed with a certificate that the application owns.

Assertion format

To compute the assertion, you can use one of the many [JSON Web Token](#) libraries in the language of your choice. The information carried by the token are as follows:

| PARAMETER | REMARK |
|------------------|--|
| <code>alg</code> | Should be RS256 |
| <code>typ</code> | Should be JWT |
| <code>x5t</code> | Should be the X.509 Certificate SHA-1 thumbprint |

Claims (payload)

| PARAMETER | REMARKS |
|------------------|--|
| <code>aud</code> | Audience: Should be
https://login.microsoftonline.com/*tenant_id*/oauth2/token |
| <code>exp</code> | Expiration date: the date when the token expires. The time is represented as the number of seconds from January 1, 1970 (1970-01-01T0:0:0Z) UTC until the time the token validity expires. |
| <code>iss</code> | Issuer: should be the client_id (Application ID of the client service) |
| <code>jti</code> | GUID: the JWT ID |
| <code>nbf</code> | Not Before: the date before which the token cannot be used. The time is represented as the number of seconds from January 1, 1970 (1970-01-01T0:0:0Z) UTC until the time the token was issued. |
| <code>sub</code> | Subject: As for <code>iss</code> , should be the client_id (Application ID of the client service) |

Signature

The signature is computed applying the certificate as described in the [JSON Web Token RFC7519 specification](#)

Example of a decoded JWT assertion

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "x5t": "gx8tGysyjcRqKjFPnd7RFwvwZI0"  
}  
.  
{  
  "aud": "https://login.microsoftonline.com/contoso.onmicrosoft.com/oauth2/token",  
  "exp": 1484593341,  
  "iss": "97e0a5b7-d745-40b6-94fe-5f77d35c6e05",  
  "jti": "22b3bb26-e046-42df-9c96-65dbd72c1c81",  
  "nbf": 1484592741,  
  "sub": "97e0a5b7-d745-40b6-94fe-5f77d35c6e05"  
}  
.  
"Gh95kHCOEGq5E_ArMBbDXhwKR577scxYaoJ1P{a lot of characters here}KKJDEg"
```

Example of an encoded JWT assertion

The following string is an example of encoded assertion. If you look carefully, you notice three sections separated by dots (.):

- The first section encodes the header
- The second section encodes the payload
- The last section is the signature computed with the certificates from the content of the first two sections

```
"eyJhbGciOiJSUzI1NiIsIng1dCI6Imd40HRHeXN5amNScUtqR1BuZDdSRnd2d1pJMCJ9.eyJhdWQiOiJodHRwczpcL1wvbG9naW4ubWljcm9zb2Z0b25saW51LmNvbVwvam1wcm1ldXJob3RtYWlsLm9ubW1jcm9zb2Z0LmNvbVwvb2F1dGgyXC90b2t1biIsImV4cCI6MTQ4NDU5MzM0MSwiaXNzIjoiOTd1MGE1YjctZDc0NS00MGI2LTk0ZmUtNWY3N2QzNWM2ZTA1IiwianRpIjoiMjJiM2JiMjYtZTA0Ni00MmRmLTljOTYtNjVkJmQ3MmMxYzgxiIwibmJmIjoxNDg0NTkyNzQxLCJzdWIiOiI5N2UwYTViNy1kNzQ1LTQwYjYtOTRmZS01Zjc3ZDM1Yz1MDUiifQ.  
Gh95kHCOEGq5E_ArMBbDXhwKR577scxYaoJ1P{a lot of characters here}KKJDEg"
```

Register your certificate with Azure AD

You can associate the certificate credential with the client application in Azure AD through the Azure portal using any of the following methods:

Uploading the certificate file

In the Azure app registration for the client application:

1. Select **Certificates & secrets**.
2. Click on **Upload certificate** and select the certificate file to upload.
3. Click **Add**. Once the certificate is uploaded, the thumbprint, start date, and expiration values are displayed.

Updating the application manifest

Having hold of a certificate, you need to compute:

- `$base64Thumbprint`, which is the base64 encoding of the certificate hash
- `$base64Value`, which is the base64 encoding of the certificate raw data

You also need to provide a GUID to identify the key in the application manifest (`$keyId`).

In the Azure app registration for the client application:

1. Select **Manifest** to open the application manifest.
2. Replace the *keyCredentials* property with your new certificate information using the following schema.

```
"keyCredentials": [  
    {  
        "customKeyIdentifier": "$base64Thumbprint",  
        "keyId": "$keyid",  
        "type": "AsymmetricX509Cert",  
        "usage": "Verify",  
        "value": "$base64Value"  
    }  
]
```

3. Save the edits to the application manifest and then upload the manifest to Azure AD.

The `keyCredentials` property is multi-valued, so you may upload multiple certificates for richer key management.

Code sample

NOTE

You must calculate the X5T header by using the certificate's hash and converting it to a base64 string. In C# it would look something similar to that of: `System.Convert.ToBase64String(cert.GetCertHash());`

The code sample on [Authenticating to Azure AD in daemon apps with certificates](#) shows how an application uses its own credentials for authentication. It also shows how you can [create a self-signed certificate](#) using the `New-SelfSignedCertificate` Powershell command. You can also take advantage and use the [app creation scripts](#) to create the certificates, compute the thumbprint, and so on.

Application and service principal objects in Azure Active Directory

10/23/2019 • 4 minutes to read • [Edit Online](#)

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.

Application registration

When you register an Azure AD application in the [Azure portal](#), two objects are created in your Azure AD tenant:

- An application object, and
- A service principal object

Application object

An Azure AD application is defined by its one and only application object, which resides in the Azure AD tenant where the application was registered, known as the application's "home" tenant. 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 AD 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 AD tenant. This enables core features such as authentication of the user/application during sign-in, and authorization during resource access.

When an application is given permission to access resources in a tenant (upon registration or [consent](#)), a service principal object is created. The Microsoft Graph [ServicePrincipal entity](#) defines the schema for a service principal object's properties.

Application and service principal relationship

Consider the application object as the *global* representation of your application for use across all tenants, and the service principal as 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 therefore has a 1:1 relationship with the software application, and a 1:many relationships with its corresponding service principal object(s).

A service principal must be created in each tenant where the application is used, enabling it to establish an identity for sign-in and/or access to resources being secured by the tenant. A single-tenant application has only one service principal (in its home tenant), created and consented for use during application registration. A multi-tenant Web application/API also has a service principal created in each tenant where a user from that tenant has consented to its use.

NOTE

Any changes you make to your application object, are also reflected in its service principal object in the application's home tenant only (the tenant where it was registered). For multi-tenant applications, changes to the application object are not reflected in any consumer tenants' service principal objects, until the access is removed through the [Application Access Panel](#) and granted again.

Also note that native applications are registered as multi-tenant by default.

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. |

Next steps

- You can use the [Microsoft Graph Explorer](#) to query both the application and service principal objects.
- You can access an application's application object using the Microsoft Graph API, the [Azure portal's](#) application manifest editor, or [Azure AD PowerShell cmdlets](#), as represented by its OData [Application entity](#).
- You can access an application's service principal object through the Microsoft Graph API or [Azure AD PowerShell cmdlets](#), as represented by its OData [ServicePrincipal entity](#).

How and why applications are added to Azure AD

10/23/2019 • 7 minutes to read • [Edit Online](#)

There are two representations of applications in Azure AD:

- [Application objects](#) - Although there are [exceptions](#), application objects can be considered the definition of an application.
- [Service principals](#) - Can be considered an instance of an application. Service principals generally reference an application object, and one application object can be referenced by multiple service principals across directories.

What are application objects and where do they come from?

You can manage [application objects](#) in the Azure portal through the [App Registrations](#) experience. Application objects describe the application to Azure AD and can be considered the definition of the application, allowing the service to know how to issue tokens to the application based on its settings. The application object will only exist in its home directory, even if it's a multi-tenant application supporting service principals in other directories. The application object may include any of the following (as well as additional information not mentioned here):

- Name, logo, and publisher
- Redirect URLs
- Secrets (symmetric and/or asymmetric keys used to authenticate the application)
- API dependencies (OAuth)
- Published APIs/resources/scopes (OAuth)
- App roles (RBAC)
- SSO metadata and configuration
- User provisioning metadata and configuration
- Proxy metadata and configuration

Application objects can be created through multiple pathways, including:

- Application registrations in the Azure portal
- Creating a new application using Visual Studio and configuring it to use Azure AD authentication
- When an admin adds an application from the app gallery (which will also create a service principal)
- Using the Microsoft Graph API, Azure AD Graph API, or PowerShell to create a new application
- Many others including various developer experiences in Azure and in API explorer experiences across developer centers

What are service principals and where do they come from?

You can manage [service principals](#) in the Azure portal through the [Enterprise Applications](#) experience. Service principals are what govern an application connecting to Azure AD and can be considered the instance of the application in your directory. For any given application, it can have at most one application object (which is registered in a "home" directory) and one or more service principal objects representing instances of the application in every directory in which it acts.

The service principal can include:

- A reference back to an application object through the application ID property
- Records of local user and group application-role assignments
- Records of local user and admin permissions granted to the application

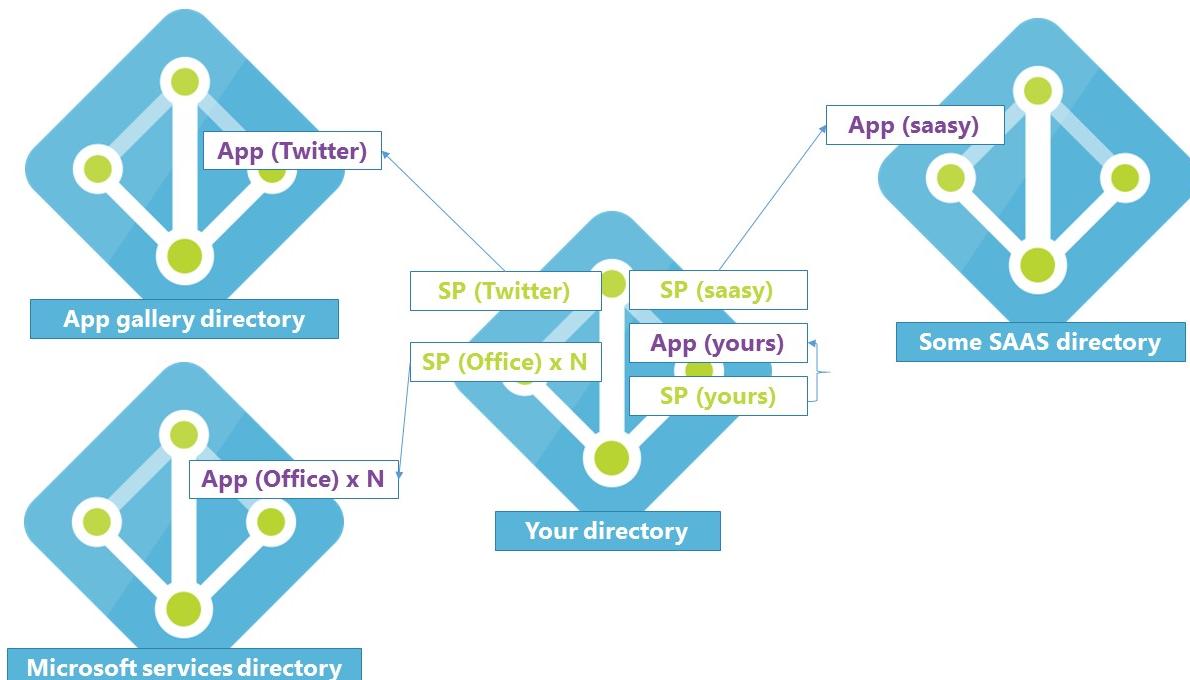
- For example: permission for the application to access a particular user's email
- Records of local policies including Conditional Access policy
- Records of alternate local settings for an application
 - Claims transformation rules
 - Attribute mappings (User provisioning)
 - Directory-specific app roles (if the application supports custom roles)
 - Directory-specific name or logo

Like application objects, service principals can also be created through multiple pathways including:

- When users sign in to a third-party application integrated with Azure AD
 - During sign-in, users are asked to give permission to the application to access their profile and other permissions. The first person to give consent causes a service principal that represents the application to be added to the directory.
- When users sign in to Microsoft online services like [Office 365](#)
 - When you subscribe to Office 365 or begin a trial, one or more service principals are created in the directory representing the various services that are used to deliver all of the functionality associated with Office 365.
 - Some Office 365 services like SharePoint create service principals on an ongoing basis to allow secure communication between components including workflows.
- When an admin adds an application from the app gallery (this will also create an underlying app object)
- Add an application to use the [Azure AD Application Proxy](#)
- Connect an application for single sign on using SAML or password single sign-on (SSO)
- Programmatically via the Azure AD Graph API or PowerShell

How are application objects and service principals related to each other?

An application has one application object in its home directory that is referenced by one or more service principals in each of the directories where it operates (including the application's home directory).



In the preceding diagram, Microsoft maintains two directories internally (shown on the left) that it uses to publish

applications:

- One for Microsoft Apps (Microsoft services directory)
- One for pre-integrated third-party applications (App gallery directory)

Application publishers/vendors who integrate with Azure AD are required to have a publishing directory (shown on the right as "Some SaaS Directory").

Applications that you add yourself (represented as **App (yours)** in the diagram) include:

- Apps you developed (integrated with Azure AD)
- Apps you connected for single-sign-on
- Apps you published using the Azure AD application proxy

Notes and exceptions

- Not all service principals point back to an application object. When Azure AD was originally built the services provided to applications were more limited and the service principal was sufficient for establishing an application identity. The original service principal was closer in shape to the Windows Server Active Directory service account. For this reason, it's still possible to create service principals through different pathways, such as using Azure AD PowerShell, without first creating an application object. The Azure AD Graph API requires an application object before creating a service principal.
- Not all of the information described above is currently exposed programmatically. The following are only available in the UI:
 - Claims transformation rules
 - Attribute mappings (User provisioning)
- For more detailed information on the service principal and application objects, see the Azure AD Graph REST API reference documentation:
 - [Application](#)
 - [Service Principal](#)

Why do applications integrate with Azure AD?

Applications are added to Azure AD to leverage one or more of the services it provides including:

- Application authentication and authorization
- User authentication and authorization
- SSO using federation or password
- User provisioning and synchronization
- Role-based access control - Use the directory to define application roles to perform role-based authorization checks in an application
- OAuth authorization services - Used by Office 365 and other Microsoft applications to authorize access to APIs/resources
- Application publishing and proxy - Publish an application from a private network to the internet

Who has permission to add applications to my Azure AD instance?

While there are some tasks that only global administrators can do (such as adding applications from the app gallery and configuring an application to use the Application Proxy) by default all users in your directory have rights to register application objects that they are developing and discretion over which applications they share/give access to their organizational data through consent. If a person is the first user in your directory to sign in to an application and grant consent, that will create a service principal in your tenant; otherwise, the consent grant information will be stored on the existing service principal.

Allowing users to register and consent to applications might initially sound concerning, but keep the following in mind:

- Applications have been able to leverage Windows Server Active Directory for user authentication for many years without requiring the application to be registered or recorded in the directory. Now the organization will have improved visibility to exactly how many applications are using the directory and for what purpose.
- Delegating these responsibilities to users negates the need for an admin-driven application registration and publishing process. With Active Directory Federation Services (ADFS) it was likely that an admin had to add an application as a relying party on behalf of their developers. Now developers can self-service.
- Users signing in to applications using their organization accounts for business purposes is a good thing. If they subsequently leave the organization they will automatically lose access to their account in the application they were using.
- Having a record of what data was shared with which application is a good thing. Data is more transportable than ever and it's useful to have a clear record of who shared what data with which applications.
- API owners who use Azure AD for OAuth decide exactly what permissions users are able to grant to applications and which permissions require an admin to agree to. Only admins can consent to larger scopes and more significant permissions, while user consent is scoped to the users' own data and capabilities.
- When a user adds or allows an application to access their data, the event can be audited so you can view the Audit Reports within the Azure portal to determine how an application was added to the directory.

If you still want to prevent users in your directory from registering applications and from signing in to applications without administrator approval, there are two settings that you can change to turn off those capabilities:

- To prevent users from consenting to applications on their own behalf:
 1. In the Azure portal, go to the [User settings](#) section under Enterprise applications.
 2. Change **Users can consent to apps accessing company data on their behalf** to **No**.

NOTE

If you decide to turn off user consent, an admin will be required to consent to any new application a user needs to use.

- To prevent users from registering their own applications:
 1. In the Azure portal, go to the [User settings](#) section under Azure Active Directory
 2. Change **Users can register applications** to **No**.

NOTE

Microsoft itself uses the default configuration with users able to register applications and consent to applications on their own behalf.

Redirect URI/reply URL restrictions and limitations

10/15/2019 • 4 minutes to read • [Edit Online](#)

A redirect URI, or reply URL, is the location that the authorization server will send the user to once the app has been successfully authorized, and granted an authorization code or access token. The code or token is contained in the redirect URI or reply token so it's important that you register the correct location as part of the app registration process.

Maximum number of redirect URIs

The following table shows the maximum number of redirect URIs that you can add when you register your app.

| ACCOUNTS BEING SIGNED IN | MAXIMUM NUMBER OF REDIRECT URIS | DESCRIPTION |
|--|---------------------------------|--|
| Microsoft work or school accounts in any organization's Azure Active Directory (Azure AD) tenant | 256 | <code>signInAudience</code> field in the application manifest is set to either <code>AzureADMyOrg</code> or <code>AzureADMultipleOrgs</code> |
| Personal Microsoft accounts and work and school accounts | 100 | <code>signInAudience</code> field in the application manifest is set to <code>AzureADandPersonalMicrosoftAccount</code> |

Maximum URI length

You can use a maximum of 256 characters for each redirect URI that you add to an app registration.

Supported schemes

The Azure AD application model today supports both HTTP and HTTPS schemes for apps that sign in Microsoft work or school accounts in any organization's Azure Active Directory (Azure AD) tenant. That is `signInAudience` field in the application manifest is set to either `AzureADMyOrg` or `AzureADMultipleOrgs`. For the apps that sign in Personal Microsoft accounts and work and school accounts (that is `signInAudience` set to `AzureADandPersonalMicrosoftAccount`) only HTTPS scheme is allowed.

NOTE

The new [App registrations](#) experience doesn't allow developers to add URIs with HTTP scheme on the UI. Adding HTTP URIs for apps that sign in work or school accounts is supported only through the app manifest editor. Going forward, new apps won't be able to use HTTP schemes in the redirect URI. However, older apps that contain HTTP schemes in redirect URIs will continue to work. Developers must use HTTPS schemes in the redirect URIs.

Restrictions using a wildcard in URIs

Wildcard URIs, such as `https://*.contoso.com`, are convenient but should be avoided. Using wildcards in the redirect URI has security implications. According to the OAuth 2.0 specification ([section 3.1.2 of RFC 6749](#)), a redirection endpoint URI must be an absolute URI.

The Azure AD application model doesn't support wildcard URIs for apps that are configured to sign in personal Microsoft accounts and work or school accounts. However, wildcard URIs are allowed for apps that are configured

to sign in work or school accounts in an organization's Azure AD tenant today.

NOTE

The new [App registrations](#) experience doesn't allow developers to add wildcard URIs on the UI. Adding wilcard URI for apps that sign in work or school accounts is supported only through the app manifest editor. Going forward, new apps won't be able to use wildcards in the redirect URI. However, older apps that contain wildcards in redirect URIs will continue to work.

If your scenario requires more redirect URIs than the maximum limit allowed, instead of adding a wildcard redirect URI, consider one of the following approaches.

Use a state parameter

If you have a number of sub-domains, and if your scenario requires you to redirect users upon successful authentication to the same page where they started, using a state parameter might be helpful.

In this approach:

1. Create a "shared" redirect URI per application to process the security tokens you receive from the authorization endpoint.
2. Your application can send application-specific parameters (such as sub-domain URL where the user originated or anything like branding information) in the state parameter. When using a state parameter, guard against CSRF protection as specified in [section 10.12 of RFC 6749](#).
3. The application-specific parameters will include all the information needed for the application to render the correct experience for the user, that is, construct the appropriate application state. The Azure AD authorization endpoint strips HTML from the state parameter so make sure you are not passing HTML content in this parameter.
4. When Azure AD sends a response to the "shared" redirect URI, it will send the state parameter back to the application.
5. The application can then use the value in the state parameter to determine which URL to further send the user to. Make sure you validate for CSRF protection.

NOTE

This approach allows a compromised client to modify the additional parameters sent in the state parameter, thereby redirecting the user to a different URL, which is the [open redirector threat](#) described in RFC 6819. Therefore, the client must protect these parameters by encrypting the state or verifying it by some other means such as validating domain name in the redirect URI against the token.

Add redirect URIs to service principals

Another approach is to add redirect URIs to the [service principals](#) that represent your app registration in any Azure AD tenant. You can use this approach when you can't use a state parameter or your scenario requires you to add new redirect URIs to your app registration for every new tenant you support.

Next steps

- Learn about the [Application manifest](#)

Validation differences by supported account types (signInAudience)

10/15/2019 • 2 minutes to read • [Edit Online](#)

When registering an application with the Microsoft identity platform for developers, you are asked to select which account types your application supports. In the application object and manifest, this property is `signInAudience`.

The options include the following:

- *AzureADMyOrg*: Only accounts in the organizational directory where the app is registered (single-tenant)
- *AzureADMultipleOrgs*: Accounts in any organizational directory (multi-tenant)
- *AzureADandPersonalMicrosoftAccount*: Accounts in any organizational directory (multi-tenant) and personal Microsoft accounts (for example, Skype, Xbox, and Outlook.com)

For registered applications, you can find the value for Supported account types on the **Authentication** section of an application. You can also find it under the `signInAudience` property in the **Manifest**.

The value you select for this property has implications on other app object properties. As a result, if you change this property you may need to change other properties first.

See the following table for the validation differences of various properties for different supported account types.

| PROPERTY | AZUREADMYORG | AZUREADMULTIPLEORGs | AZUREADANDPERSONALMICROSOFTACCOUNT |
|---|---|--|---|
| Application ID URI (<code>identifierURIs</code>) | Must be unique in the tenant

urn:// schemes are supported

Wildcards are not supported

Query strings and fragments are supported

Maximum length of 255 characters

No limit* on number of identifierURIs | Must be globally unique

urn:// schemes are supported

Wildcards are not supported

Query strings and fragments are supported

Maximum length of 255 characters

No limit* on number of identifierURIs | Must be globally unique

urn:// schemes are not supported

Wildcards, fragments and query strings are not supported

Maximum length of 120 characters

Maximum of 50 identifierURIs |
| Certificates (<code>keyCredentials</code>) | Symmetric signing key | Symmetric signing key | Encryption and asymmetric signing key |
| Client secrets (<code>passwordCredentials</code>) | No limit* | No limit* | If liveSDK is enabled:
Maximum of 2 client secrets |
| Redirect URIs (<code>replyURLs</code>) | See Redirect URI/reply URL restrictions and limitations for more info. | | |

| PROPERTY | AZUREADMYORG | AZUREADMULTIPLEORGS | AZUREADANDPERSONALMICROSOFTACCOUNT |
|--|--|--|---|
| API permissions (
<code>requiredResourceAccess</code>) | No limit* | No limit* | Maximum of 30 permissions per resource allowed (e.g. Microsoft Graph) |
| Scopes defined by this API (
<code>oauth2Permissions</code>) | Maximum scope name length of 120 characters

No limit* on the number of scopes defined | Maximum scope name length of 120 characters

No limit* on the number of scopes defined | Maximum scope name length of 40 characters

Maximum of 100 scopes defined |
| Authorized client applications (
<code>preauthorizedApplications</code>) | No limit* | No limit* | Total maximum of 500

Maximum of 100 client apps defined

Maximum of 30 scopes defined per client |
| appRoles | Supported
No limit* | Supported
No limit* | Not supported |
| Logout URL | http://localhost is allowed

Maximum length of 255 characters | http://localhost is allowed

Maximum length of 255 characters | https://localhost is allowed, http://localhost fails for MSA

Maximum length of 255 characters

HTTP scheme is not allowed

Wildcards are not supported |

*There is a global limit of about 1000 items across all the collection properties on the app object

Next steps

- Learn about [Application registration](#)
- Learn about the [Application manifest](#)

Tenancy in Azure Active Directory

10/23/2019 • 2 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) organizes objects like users and apps into groups called *tenants*. Tenants allow an administrator to set policies on the users within the organization and the apps that the organization owns to meet their security and operational policies.

Who can sign in to your app?

When it comes to developing apps, developers can choose to configure their app to be either single-tenant or multi-tenant during app registration in the [Azure portal](#).

- Single-tenant apps are only available in the tenant they were registered in, also known as their home tenant.
- Multi-tenant apps are available to users in both their home tenant and other tenants.

In the Azure portal, you can configure your app to be single-tenant or multi-tenant by setting the audience as follows.

| AUDIENCE | SINGLE/MULTI-TENANT | WHO CAN SIGN IN |
|---|---------------------|---|
| Accounts in this directory only | Single tenant | All user and guest accounts in your directory can use your application or API.
<i>Use this option if your target audience is internal to your organization.</i> |
| Accounts in any Azure AD directory | Multi-tenant | All users and guests with a work or school account from Microsoft can use your application or API. This includes schools and businesses that use Office 365.
<i>Use this option if your target audience is business or educational customers.</i> |
| Accounts in any Azure AD directory and personal Microsoft accounts (such as Skype, Xbox, Outlook.com) | Multi-tenant | All users with a work or school, or personal Microsoft account can use your application or API. It includes schools and businesses that use Office 365 as well as personal accounts that are used to sign in to services like Xbox and Skype.
<i>Use this option to target the widest set of Microsoft accounts.</i> |

Best practices for multi-tenant apps

Building great multi-tenant apps can be challenging because of the number of different policies that IT administrators can set in their tenants. If you choose to build a multi-tenant app, follow these best practices:

- Test your app in a tenant that has configured [Conditional Access policies](#).
- Follow the principle of least user access to ensure that your app only requests permissions it actually needs. Avoid requesting permissions that require admin consent as this may prevent users from acquiring your app at all in some organizations.

- Provide appropriate names and descriptions for any permissions you expose as part of your app. This helps users and admins know what they are agreeing to when they attempt to use your app's APIs. For more information, see the best practices section in the [permissions guide](#).

Next steps

- [How to convert an app to be multi-tenant](#)

Permissions and consent in the Microsoft identity platform endpoint

11/12/2019 • 18 minutes to read • [Edit Online](#)

Applies to:

- Microsoft identity platform endpoint

Applications that integrate with Microsoft identity platform follow an authorization model that gives users and administrators control over how data can be accessed. The implementation of the authorization model has been updated on the Microsoft identity platform endpoint, and it changes how an app must interact with the Microsoft identity platform. This article covers the basic concepts of this authorization model, including scopes, permissions, and consent.

NOTE

The Microsoft identity platform endpoint does not support all scenarios and features. To determine whether you should use the Microsoft identity platform endpoint, read about [Microsoft identity platform limitations](#).

Scopes and permissions

The Microsoft identity platform implements the [OAuth 2.0](#) 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*. For example, some of Microsoft's web-hosted resources include:

- Microsoft Graph: <https://graph.microsoft.com>
- Office 365 Mail API: <https://outlook.office.com>
- Azure AD Graph: <https://graph.windows.net>

NOTE

We strongly recommend that you use Microsoft Graph instead of Azure AD Graph, Office 365 Mail API, etc.

The same is true for any third-party resources that have integrated with the Microsoft identity platform. Any of these resources also can define a set of permissions that can be used to divide the functionality of that resource into smaller chunks. As an example, [Microsoft Graph](#) has defined permissions to do the following tasks, among others:

- Read a user's calendar
- Write to a user's calendar
- Send mail as a user

By defining these types of permissions, the resource has fine-grained control over its data and how API functionality is exposed. A third-party app can request these permissions from users and administrators, who must approve the request before the app can access data or act on a user's behalf. By chunking the

resource's functionality into smaller permission sets, third-party apps can be built to request only the specific permissions that they need to perform their function. Users and administrators can know exactly what data the app has access to, and they can be more confident that it isn't behaving with malicious intent. Developers should always abide by the concept of least privilege, asking for only the permissions they need for their applications to function.

In OAuth 2.0, these types of permissions are called *scopes*. They are also often referred to as *permissions*. A permission is represented in the Microsoft identity platform as a string value. Continuing with the Microsoft Graph example, the string value for each permission is:

- Read a user's calendar by using `Calendars.Read`
- Write to a user's calendar by using `Calendars.ReadWrite`
- Send mail as a user using `Mail.Send`

An app most commonly requests these permissions by specifying the scopes in requests to the Microsoft identity platform authorize endpoint. However, certain high privilege permissions can only be granted through administrator consent and requested/granted using the [administrator consent endpoint](#). Read on to learn more.

Permission types

Microsoft identity platform supports two types of permissions: **delegated permissions** and **application 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 the target resource. Some delegated permissions can be consented to by non-administrative users, but some higher-privileged permissions require [administrator consent](#). To learn which administrator roles can consent to delegated permissions, see [Administrator role permissions in Azure AD](#).
- **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](#).

Effective permissions are the permissions that your app will have when making requests to the target resource. It's important to understand the difference between the delegated and application permissions that your app is granted and its effective permissions when making calls to the target resource.

- For delegated permissions, the *effective permissions* of your app will be the least privileged intersection of the delegated permissions the app has been granted (via 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. 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 isn't 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 will be 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.

OpenID Connect scopes

The Microsoft identity platform implementation of OpenID Connect has a few well-defined scopes that do not apply to a specific resource: `openid`, `email`, `profile`, and `offline_access`. The `address` and `phone` OpenID Connect scopes are not supported.

openid

If an app performs sign-in by using [OpenID Connect](#), it must request the `openid` scope. The `openid` scope shows on the work account consent page as the "Sign you in" permission, and on the personal Microsoft account consent page as the "View your profile and connect to apps and services using your Microsoft account" permission. With this permission, an app can receive a unique identifier for the user in the form of the `sub` claim. It also gives the app access to the UserInfo endpoint. The `openid` scope can be used at the Microsoft identity platform token endpoint to acquire ID tokens, which can be used by the app for authentication.

email

The `email` scope can be used with the `openid` scope and any others. It gives the app access to the user's primary email address in the form of the `email` claim. The `email` claim is included in a token only if an email address is associated with the user account, which isn't always the case. If it uses the `email` scope, your app should be prepared to handle a case in which the `email` claim does not exist in the token.

profile

The `profile` scope can be used with the `openid` scope and any others. It gives the app access to a substantial amount of information about the user. The information it can access includes, but isn't limited to, the user's given name, surname, preferred username, and object ID. For a complete list of the profile claims available in the `id_tokens` parameter for a specific user, see the [id_tokens reference](#).

offline_access

The `offline_access` scope gives your app access to resources on behalf of the user for an extended time. On the consent page, this scope appears as the "Maintain access to data you have given it access to" permission. When a user approves the `offline_access` scope, your app can receive refresh tokens from the Microsoft identity platform token endpoint. Refresh tokens are long-lived. Your app can get new access tokens as older ones expire.

If your app does not explicitly request the `offline_access` scope, it won't receive refresh tokens. This means that when you redeem an authorization code in the [OAuth 2.0 authorization code flow](#), you'll receive only an access token from the `/token` endpoint. The access token is valid for a short time. The access token usually expires in one hour. At that point, your app needs to redirect the user back to the `/authorize` endpoint to get a new authorization code. During this redirect, depending on the type of app, the user might need to enter their credentials again or consent again to permissions. While the `offline_access` scope is automatically requested by the server, your client must still request it in order to receive the refresh tokens.

For more information about how to get and use refresh tokens, see the [Microsoft identity platform protocol reference](#).

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 in to an app, the app sends a request like the following example (with line breaks added for legibility):

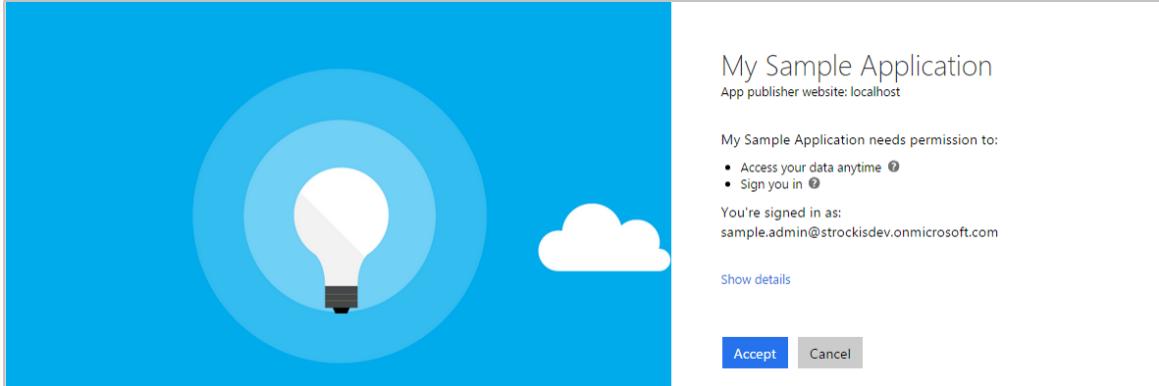
```
GET https://login.microsoftonline.com/common/oauth2/v2.0/authorize?  
client_id=6731de76-14a6-49ae-97bc-6eba6914391e  
&response_type=code  
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F  
&response_mode=query  
&scope=  
https%3A%2F%2Fgraph.microsoft.com%2Fcalendars.read%20  
https%3A%2F%2Fgraph.microsoft.com%2Fmail.send  
&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 endpoint checks for a matching record of *user consent*. If the user has not consented to any of the requested permissions in the past, nor has an administrator consented to these permissions on behalf of the entire organization, the Microsoft identity platform endpoint asks the user to grant the requested permissions.

NOTE

At this time, the `offline_access` ("Maintain access to data you have given it access to") and `user.read` ("Sign you in and read your profile") permissions are automatically included in the initial consent to an application. These permissions are generally required for proper app functionality - `offline_access` gives the app access to refresh tokens, critical for native and web apps, while `user.read` gives access to the `sub` claim, allowing the client or app to correctly identify the user over time and access rudimentary user information.



When the user approves the permission request, consent is recorded and the user doesn't have to consent again on subsequent sign-ins to the application.

Requesting consent for an entire tenant

Often, when an organization purchases a license or subscription for an application, the organization wants to proactively set up the application for use by all members of the organization. As part of this process, an administrator can grant consent for the application to act on behalf of any user in the tenant. If the admin grants consent for the entire tenant, the organization's users won't see a consent page for the application.

To request consent for delegated permissions for all users in a tenant, your app can use the admin consent endpoint.

Additionally, applications must use the admin consent endpoint to request Application Permissions.

Admin-restricted permissions

Some high-privilege permissions in the Microsoft ecosystem can be set to *admin-restricted*. Examples of these kinds of permissions include the following:

- Read all user's full profiles by using `User.Read.All`
- Write data to an organization's directory by using `Directory.ReadWrite.All`
- Read all groups in an organization's directory by using `Groups.Read.All`

Although a consumer user might grant an application access to this kind of data, organizational users are restricted from granting access to the same set of sensitive company data. If your application requests access to one of these permissions from an organizational user, the user receives an error message that says they're not authorized to consent to your app's permissions.

If your app requires access to admin-restricted scopes for organizations, you should request them directly from a company administrator, also by using the admin consent endpoint, described next.

If the application is requesting high privilege delegated permissions and an administrator grants these permissions via the admin consent endpoint, consent is granted for all users in the tenant.

If the application is requesting application permissions and an administrator grants these permissions via the admin consent endpoint, this grant isn't done on behalf of any specific user. Instead, the client application is granted permissions *directly*. These types of permissions are only used by daemon services and other non-interactive applications that run in the background.

Using the admin consent endpoint

NOTE

Please note after granting admin consent using the admin consent endpoint, you have finished granting admin consent and users do not need to perform any further additional actions. After granting admin consent, users can get an access token via a typical auth flow and the resulting access token will have the consented permissions.

When a Company Administrator uses your application and is directed to the authorize endpoint, Microsoft identity platform will detect the user's role and ask them if they would like to consent on behalf of the entire tenant for the permissions you have requested. However, there is also a dedicated admin consent endpoint you can use if you would like to proactively request that an administrator grants permission on behalf of the entire tenant. Using this endpoint is also necessary for requesting Application Permissions (which can't be requested using the authorize endpoint).

If you follow these steps, your app can request permissions for all users in a tenant, including admin-restricted scopes. This is a high privilege operation and should only be done if necessary for your scenario.

To see a code sample that implements the steps, see the [admin-restricted scopes sample](#).

Request the permissions in the app registration portal

The admin consent does not accept a scope parameter, so any permissions being requested must be statically defined in the application's registration. In general, it's best practice to ensure that the permissions statically defined for a given application are a superset of the permissions that it will be requesting dynamically/incrementally.

To configure the list of statically requested permissions for an application

1. Go to your application in the [Azure portal – App registrations](#) experience, or [create an app](#) if you

haven't already.

2. Locate the **API Permissions** section, and within the API permissions click Add a permission.
3. Select **Microsoft Graph** from the list of available APIs and then add the permissions that your app requires.
4. **Save** the app registration.

Recommended: Sign the user into your app

Typically, when you build an application that uses the admin consent endpoint, the app needs a page or view in which the admin can approve the app's permissions. This page can be part of the app's sign-up flow, part of the app's settings, or it can be a dedicated "connect" flow. In many cases, it makes sense for the app to show this "connect" view only after a user has signed in with a work or school Microsoft account.

When you sign the user into your app, you can identify the organization to which the admin belongs before asking them to approve the necessary permissions. Although not strictly necessary, it can help you create a more intuitive experience for your organizational users. To sign the user in, follow our [Microsoft identity platform protocol tutorials](#).

Request the permissions from a directory admin

When you're ready to request permissions from your organization's admin, you can redirect the user to the Microsoft identity platform *admin consent endpoint*.

```
// Line breaks are for legibility only.  
GET https://login.microsoftonline.com/{tenant}/v2.0/adminconsent?  
client_id=6731de76-14a6-49ae-97bc-6eba6914391e  
&state=12345  
&redirect_uri=http://localhost/myapp/permissions  
&scope=  
https://graph.microsoft.com/calendars.read  
https://graph.microsoft.com/mail.send
```

| PARAMETER | CONDITION | DESCRIPTION |
|--------------|-----------|---|
| tenant | Required | The directory tenant that you want to request permission from. Can be provided in GUID or friendly name format OR generically referenced with <code>common</code> as seen in the example. |
| client_id | Required | The Application (client) ID that the Azure portal – App registrations experience assigned to your app. |
| redirect_uri | Required | The redirect URI where you want the response to be sent for your app to handle. It must exactly match one of the redirect URIs that you registered in the app registration portal. |

| PARAMETER | CONDITION | DESCRIPTION |
|--------------------|-------------|--|
| <code>state</code> | Recommended | A value included in the request that will also be returned in the token response. It can be a string of any content you want. Use the state 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. |
| <code>scope</code> | Required | Defines the set of permissions being requested by the application. This can be either static (using <code>/.default</code>) or dynamic scopes. This can include the OIDC scopes (<code>openid</code> , <code>profile</code> , <code>email</code>). |

At this point, Azure AD requires a tenant administrator to sign in to complete the request. The administrator is asked to approve all the permissions that you have requested in the `scope` parameter. If you've used a static (`/.default`) value, it will function like the v1.0 admin consent endpoint and request consent for all scopes found in the required permissions for the app.

Successful response

If the admin approves the permissions for your app, the successful response looks like this:

```
GET http://localhost/myapp/permissions?tenant=a8990e1f-ff32-408a-9f8e-78d3b9139b95&state=state=12345&admin_consent=True
```

| PARAMETER | DESCRIPTION |
|----------------------------|--|
| <code>tenant</code> | The directory tenant that granted your application the permissions it requested, in GUID format. |
| <code>state</code> | A value included in the request that also will be returned in the token response. It can be a string of any content you want. The state is 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. |
| <code>admin_consent</code> | Will be set to <code>True</code> . |

Error response

If the admin does not approve the permissions for your app, the failed response looks like this:

```
GET http://localhost/myapp/permissions?
error=permission_denied&error_description=The+admin+canceled+the+request
```

| PARAMETER | DESCRIPTION |
|--------------------|---|
| <code>error</code> | An error code string that can be used to classify types of errors that occur, and can be used to react to errors. |

| PARAMETER | DESCRIPTION |
|--------------------------------|---|
| <code>error_description</code> | A specific error message that can help a developer identify the root cause of an error. |

After you've received a successful response from the admin consent endpoint, your app has gained the permissions it requested. Next, you can request a token for the resource you want.

Using permissions

After the user consents to permissions for your app, your app can acquire access tokens that represent your app's permission to access a resource in some capacity. An access token can be used only for a single resource, but encoded inside the access token is every permission that your app has been granted for that resource. To acquire an access token, your app can make a request to the Microsoft identity platform token endpoint, like this:

```
POST common/oauth2/v2.0/token HTTP/1.1
Host: https://login.microsoftonline.com
Content-Type: application/json

{
  "grant_type": "authorization_code",
  "client_id": "6731de76-14a6-49ae-97bc-6eba6914391e",
  "scope": "https://outlook.office.com/mail.read https://outlook.office.com/mail.send",
  "code": "AwABAAAAvPM1KaPlrEqdFSBzjqfTGBcmLdgfSTLEMPGYuNHSUYBrq...",
  "redirect_uri": "https://localhost/myapp",
  "client_secret": "zc53fwe80980293klaj9823" // NOTE: Only required for web apps
}
```

You can use the resulting access token in HTTP requests to the resource. It reliably indicates to the resource that your app has the proper permission to perform a specific task.

For more information about the OAuth 2.0 protocol and how to get access tokens, see the [Microsoft identity platform endpoint protocol reference](#).

The `/.default` scope

You can use the `/.default` scope to help migrate your apps from the v1.0 endpoint to the Microsoft identity platform endpoint. This is a built-in scope for every application that refers to the static list of permissions configured on the application registration. A `scope` value of

`https://graph.microsoft.com/.default` is functionally the same as the v1.0 endpoints

`resource=https://graph.microsoft.com` - namely, it requests a token with the scopes on Microsoft Graph that the application has registered for in the Azure portal.

The `/.default` scope can be used in any OAuth 2.0 flow, but is necessary in the [On-Behalf-Of flow](#) and [client credentials flow](#).

NOTE

Clients can't combine static (`/.default`) and dynamic consent in a single request. Thus,

`scope=https://graph.microsoft.com/.default+mail.read` will result in an error due to the combination of scope types.

`/.default` and consent

The `/.default` scope triggers the v1.0 endpoint behavior for `prompt=consent` as well. It requests

consent for all permissions registered by the application, regardless of the resource. If included as part of the request, the `/.default` scope returns a token that contains the scopes for the resource requested.

`/.default` when the user has already given consent

Because `/.default` is functionally identical to the `resource`-centric v1.0 endpoint's behavior, it brings with it the consent behavior of the v1.0 endpoint as well. Namely, `/.default` only triggers a consent prompt if no permission has been granted between the client and the resource by the user. If any such consent exists, then a token will be returned containing all scopes granted by the user for that resource. However, if no permission has been granted, or the `prompt=consent` parameter has been provided, a consent prompt will be shown for all scopes registered by the client application.

Example 1: The user, or tenant admin, has granted permissions

The user (or a tenant administrator) has granted the client the Microsoft Graph permissions `mail.read` and `user.read`. If the client makes a request for `scope=https://graph.microsoft.com/.default`, then no consent prompt will be shown regardless of the contents of the client applications registered permissions for Microsoft Graph. A token would be returned containing the scopes `mail.read` and `user.read`.

Example 2: The user hasn't granted permissions between the client and the resource

No consent for the user exists between the client and Microsoft Graph. The client has registered for the `user.read` and `contacts.read` permissions, as well as the Azure Key Vault scope `https://vault.azure.net/user_impersonation`. When the client requests a token for `scope=https://graph.microsoft.com/.default`, the user will see a consent screen for the `user.read`, `contacts.read`, and the Key Vault `user_impersonation` scopes. The token returned will have just the `user.read` and `contacts.read` scopes in it.

Example 3: The user has consented and the client requests additional scopes

The user has already consented to `mail.read` for the client. The client has registered for the `contacts.read` scope in its registration. When the client makes a request for a token using `scope=https://graph.microsoft.com/.default` and requests consent through `prompt=consent`, then the user will see a consent screen for only and all the permissions registered by the application. `contacts.read` will be present in the consent screen, but `mail.read` will not. The token returned will be for Microsoft Graph and will contain `mail.read` and `contacts.read`.

Using the `/.default` scope with the client

A special case of the `/.default` scope exists where a client requests its own `/.default` scope. The following example demonstrates this scenario.

```
// Line breaks are for legibility only.

GET https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize?
response_type=token           //code or a hybrid flow is also possible here
&client_id=9ada6f8a-6d83-41bc-b169-a306c21527a5
&scope=9ada6f8a-6d83-41bc-b169-a306c21527a5/.default
&redirect_uri=https%3A%2F%2Flocalhost
&state=1234
```

This produces a consent screen for all registered permissions (if applicable based on the above descriptions of consent and `/.default`), then returns an `id_token`, rather than an access token. This behavior exists for certain legacy clients moving from ADAL to MSAL, and should not be used by new clients targeting the Microsoft identity platform endpoint.

Troubleshooting permissions and consent

If you or your application's users are seeing unexpected errors during the consent process, see this

article for troubleshooting steps: [Unexpected error when performing consent to an application](#).

Azure Active Directory consent framework

10/23/2019 • 3 minutes to read • [Edit Online](#)

The Azure Active Directory (Azure AD) consent framework makes it easy to develop multi-tenant web and native client applications. These applications allow sign-in by user accounts from an Azure AD tenant that's different from the one where the application is registered. They may also need to access web APIs such as the Microsoft Graph API (to access Azure AD, Intune, and services in Office 365) and other Microsoft services' APIs, in addition to your own web APIs.

The framework is based on a user or an administrator giving consent to an application that asks to be registered in their directory, which may involve accessing directory data. For example, if a web client application needs to read calendar information about the user from Office 365, that user is required to consent to the client application first. After consent is given, the client application will be able to call the Microsoft Graph API on behalf of the user, and use the calendar information as needed. The [Microsoft Graph API](#) provides access to data in Office 365 (like calendars and messages from Exchange, sites and lists from SharePoint, documents from OneDrive, notebooks from OneNote, tasks from Planner, and workbooks from Excel), as well as users and groups from Azure AD and other data objects from more Microsoft cloud services.

The consent framework is built on OAuth 2.0 and its various flows, such as authorization code grant and client credentials grant, using public or confidential clients. By using OAuth 2.0, Azure AD makes it possible to build many different types of client applications--such as on a phone, tablet, server, or a web application--and gain access to the required resources.

For more info about using the consent framework with OAuth2.0 authorization grants, see [Authorize access to web applications using OAuth 2.0 and Azure AD](#) and [Authentication scenarios for Azure AD](#). For info about getting authorized access to Office 365 through Microsoft Graph, see [App authentication with Microsoft Graph](#).

Consent experience - an example

The following steps show you how the consent experience works for both the application developer and the user.

1. Assume you have a web client application that needs to request specific permissions to access a resource/API. You'll learn how to do this configuration in the next section, but essentially the Azure portal is used to declare permission requests at configuration time. Like other configuration settings, they become part of the application's Azure AD registration:

Fourth Coffee Web App - API permissions

Overview Quickstart Manage Branding Authentication Certificates & secrets API permissions Expose an API Owners Manifest Support + Troubleshooting Troubleshooting New support request

API permissions

Permissions have changed. Users and/or admins will have to consent even if they have already done so previously.

API / PERMISSIONS NAME TYPE DESCRIPTION ADMIN CONSENT REQUIRED

Microsoft Graph (1)

| API / PERMISSIONS NAME | TYPE | DESCRIPTION | ADMIN CONSENT REQUIRED |
|------------------------|-----------|-------------------------------|------------------------|
| User.Read | Delegated | Sign in and read user profile | - |

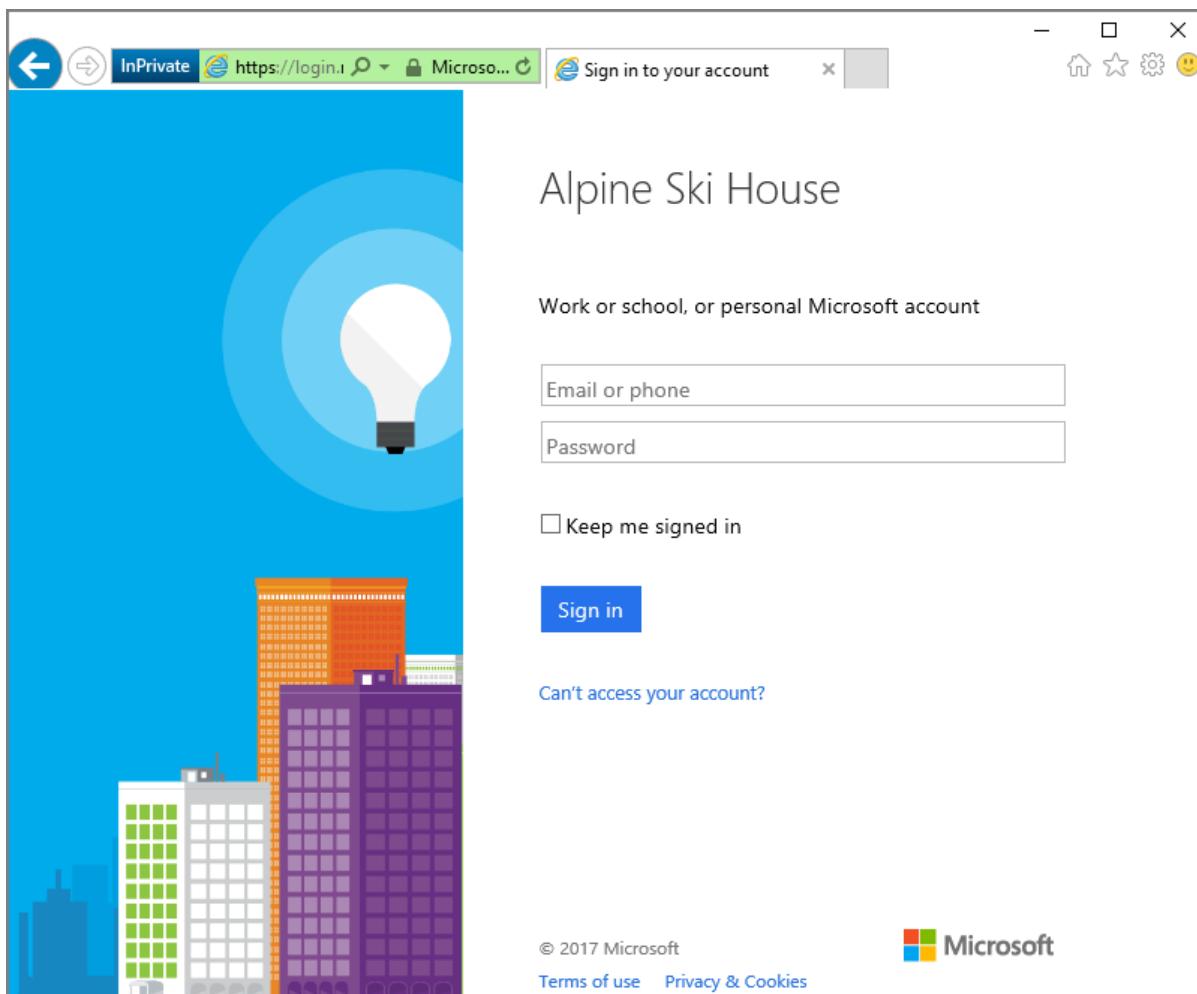
These are the permissions that this application requests statically. You may also request user consentable permissions dynamically through code. [See best practices for requesting permissions](#)

Grant consent

As an administrator, you can grant consent on behalf of all users in this directory. Granting admin consent for all users means that end users will not be shown a consent screen when using the application.

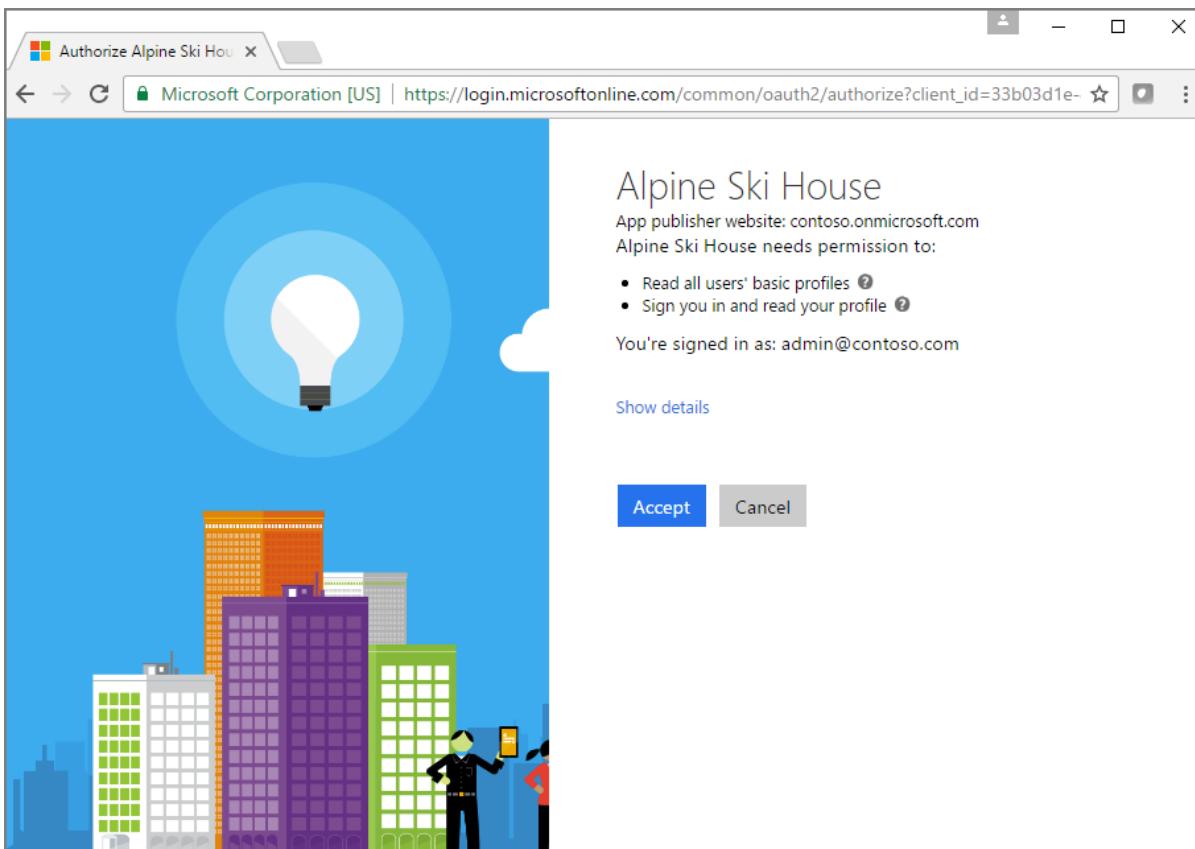
[Grant admin consent for Fourth Coffee](#)

- Consider that your application's permissions have been updated, the application is running, and a user is about to use it for the first time. First, the application needs to obtain an authorization code from Azure AD's `/authorize` endpoint. The authorization code can then be used to acquire a new access and refresh token.
- If the user is not already authenticated, Azure AD's `/authorize` endpoint prompts the user to sign in.



- After the user has signed in, Azure AD will determine if the user needs to be shown a consent page. This determination is based on whether the user (or their organization's administrator) has already granted the application consent. If consent has not already been granted, Azure AD prompts the user for consent and displays the required permissions it needs to function. The set of permissions that are displayed in the

consent dialog match the ones selected in the **Delegated permissions** in the Azure portal.



5. After the user grants consent, an authorization code is returned to your application, which is redeemed to acquire an access token and refresh token. For more information about this flow, see [Web API app type](#).
6. As an administrator, you can also consent to an application's delegated permissions on behalf of all the users in your tenant. Administrative consent prevents the consent dialog from appearing for every user in the tenant, and can be done in the [Azure portal](#) by users with the administrator role. To learn which administrator roles can consent to delegated permissions, see [Administrator role permissions in Azure AD](#).

To consent to an app's delegated permissions

- a. Go to the **API permissions** page for your application
- b. Click on the **Grant admin consent** button.

A screenshot of the Azure portal showing the "Fourth Coffee Web App - API permissions" page. The left sidebar shows navigation options like Overview, Quickstart, Manage, Branding, Authentication, Certificates & secrets, API permissions (which is highlighted with a red box), Expose an API, Owners, Manifest, Support + Troubleshooting, Troubleshooting, and New support request. The main content area shows a warning message: "Permissions have changed. Users and/or admins will have to consent even if they have already done so previously." Below this is the "API permissions" section, which lists "Microsoft Graph (1)" with a single permission entry: "User.Read" (Type: Delegated, Description: Sign in and read user profile). A note below states: "These are the permissions that this application requests statically. You may also request user consentable permissions dynamically through code. See [best practices for requesting permissions](#)". At the bottom is the "Grant consent" section with the "Grant admin consent for Fourth Coffee" button, which is also highlighted with a red box.

IMPORTANT

Granting explicit consent using the **Grant permissions** button is currently required for single-page applications (SPA) that use ADAL.js. Otherwise, the application fails when the access token is requested.

Next steps

- See [how to convert an app to be multi-tenant](#)
- For more depth, learn [how consent is supported at the OAuth 2.0 protocol layer during the authorization code grant flow](#).

Admin consent on the Microsoft identity platform

9/26/2019 • 3 minutes to read • [Edit Online](#)

Some permissions require consent from an administrator before they can be granted within a tenant. You can also use the admin consent endpoint to grant permissions to an entire tenant.

Recommended: Sign the user into your app

Typically, when you build an application that uses the admin consent endpoint, the app needs a page or view in which the admin can approve the app's permissions. This page can be part of the app's sign-up flow, part of the app's settings, or it can be a dedicated "connect" flow. In many cases, it makes sense for the app to show this "connect" view only after a user has signed in with a work or school Microsoft account.

When you sign the user into your app, you can identify the organization to which the admin belongs before asking them to approve the necessary permissions. Although not strictly necessary, it can help you create a more intuitive experience for your organizational users. To sign the user in, follow our [Microsoft identity platform protocol tutorials](#).

Request the permissions from a directory admin

When you're ready to request permissions from your organization's admin, you can redirect the user to the Microsoft identity platform *admin consent endpoint*.

```
// Line breaks are for legibility only.  
GET https://login.microsoftonline.com/{tenant}/v2.0/adminconsent?  
client_id=6731de76-14a6-49ae-97bc-6eba6914391e  
&state=12345  
&redirect_uri=http://localhost/myapp/permissions  
&scope=  
https://graph.microsoft.com/calendars.read  
https://graph.microsoft.com/mail.send
```

| PARAMETER | CONDITION | DESCRIPTION |
|---------------------------|-------------|--|
| <code>tenant</code> | Required | The directory tenant that you want to request permission from. Can be provided in GUID or friendly name format OR generically referenced with <code>common</code> as seen in the example. |
| <code>client_id</code> | Required | The Application (client) ID that the Azure portal – App registrations experience assigned to your app. |
| <code>redirect_uri</code> | Required | The redirect URI where you want the response to be sent for your app to handle. It must exactly match one of the redirect URIs that you registered in the app registration portal. |
| <code>state</code> | Recommended | A value included in the request that will also be returned in the token response. It can be a string of any content you want. Use the state 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. |
| <code>scope</code> | Required | Defines the set of permissions being requested by the application. This can be either static (using <code>/default</code>) or dynamic scopes. This can include the OIDC scopes (<code>openid</code> , <code>profile</code> , <code>email</code>). |

At this point, Azure AD requires a tenant administrator to sign in to complete the request. The administrator is asked to approve all the permissions that you have requested in the `scope` parameter. If you've used a static (`/default`) value, it will function like the v1.0 admin consent endpoint and request consent for all scopes found in the required permissions for the app.

Successful response

If the admin approves the permissions for your app, the successful response looks like this:

```
http://localhost/myapp/permissions?admin_consent=True&tenant=fa00d692-e9c7-4460-a743-29f2956fd429&state=12345&scope=https%3a%2f%2fgraph.microsoft.com%2fCalendars.Read+https%3a%2f%2fgraph.microsoft.com%2fMail.Send
```

| PARAMETER | DESCRIPTION |
|---------------|--|
| tenant | The directory tenant that granted your application the permissions it requested, in GUID format. |
| state | A value included in the request that also will be returned in the token response. It can be a string of any content you want. The state is 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. |
| scope | The set of permissions that were granted access to, for the application. |
| admin_consent | Will be set to <code>True</code> . |

Error response

```
http://localhost/myapp/permissions?  
error=consent_required&error_description=AADSTS65004%3a+The+resource+owner+or+authorization+server+denied+the+request.%0d%0aTrace+ID%3a+d320620c-  
3d56-42bc-bc45-4cdd85c41f00%0d%0aCorrelation+ID%3a+8478d534-5b2c-4325-8c2c-51395c342c89%0d%0aTimestamp%3a+2019-09-  
24+18%3a34%3a26Z&admin_consent=True&tenant=fa15d692-e9c7-4460-a743-29f2956fd429&state=12345
```

Adding to the parameters seen in a successful response, error parameters are seen as below.

| 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 error. |
| tenant | The directory tenant that granted your application the permissions it requested, in GUID format. |
| state | A value included in the request that also will be returned in the token response. It can be a string of any content you want. The state is 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. |
| admin_consent | Will be set to <code>True</code> to indicate that this response occurred on an admin consent flow. |

Next steps

- See [how to convert an app to be multi-tenant](#)
- Learn how [consent is supported at the OAuth 2.0 protocol layer during the authorization code grant flow](#).
- Learn how [a multi-tenant application can use the consent framework](#) to implement "user" and "admin" consent, supporting more advanced multi-tier application patterns.
- Understanding [Azure AD application consent experiences](#)

Understanding Azure AD application consent experiences

10/23/2019 • 4 minutes to read • [Edit Online](#)

Learn more about the Azure Active Directory (Azure AD) application consent user experience. So you can intelligently manage applications for your organization and/or develop applications with a more seamless consent experience.

Consent and permissions

Consent is the process of a user granting authorization to an application to access protected resources on their behalf. An admin or user can be asked for consent to allow access to their organization/individual data.

The actual user experience of granting consent will differ depending on policies set on the user's tenant, the user's scope of authority (or role), and the type of [permissions](#) being requested by the client application. This means that application developers and tenant admins have some control over the consent experience. Admins have the flexibility of setting and disabling policies on a tenant or app to control the consent experience in their tenant. Application developers can dictate what types of permissions are being requested and if they want to guide users through the user consent flow or the admin consent flow.

- **User consent flow** is when an application developer directs users to the authorization endpoint with the intent to record consent for only the current user.
- **Admin consent flow** is when an application developer directs users to the admin consent endpoint with the intent to record consent for the entire tenant. To ensure the admin consent flow works properly, application developers must list all permissions in the `RequiredResourceAccess` property in the application manifest. For more info, see [Application manifest](#).

Building blocks of the consent prompt

The consent prompt is designed to ensure users have enough information to determine if they trust the client application to access protected resources on their behalf. Understanding the building blocks will help users granting consent make more informed decisions and it will help developers build better user experiences.

The following diagram and table provide information about the building blocks of the consent prompt.



① kelly@contoso.com

② Permissions requested

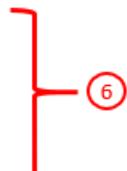
③  Contoso Test App ④
zawad.co ⑤

This app would like to:

✓ Read and write your files

✓ Read your calendar

✗ Sign you in and read your profile



⑦ [Allows you to sign in to the app with your organizational account and let the app read your profile. It also allows the app to read basic company information.

Accepting these permissions means that you allow this app

⑧ to use your data as specified in their [terms of service](#) and [privacy statement](#). You can change these permissions at

⑨ <https://myapps.microsoft.com>.

Only accept if you trust the publisher and if you selected this app from a store or website you trust. Ask your admin if you're not sure. Microsoft is not involved in licensing this app to you. [Hide details](#)

Cancel

Accept

| # | COMPONENT | PURPOSE |
|---|-----------------|---|
| 1 | User identifier | This identifier represents the user that the client application is requesting to access protected resources on behalf of. |
| 2 | Title | The title changes based on whether the users are going through the user or admin consent flow. In user consent flow, the title will be "Permissions requested" while in the admin consent flow the title will have an additional line "Accept for your organization". |
| 3 | App logo | This image should help users have a visual cue of whether this app is the app they intended to access. This image is provided by application developers and the ownership of this image isn't validated. |

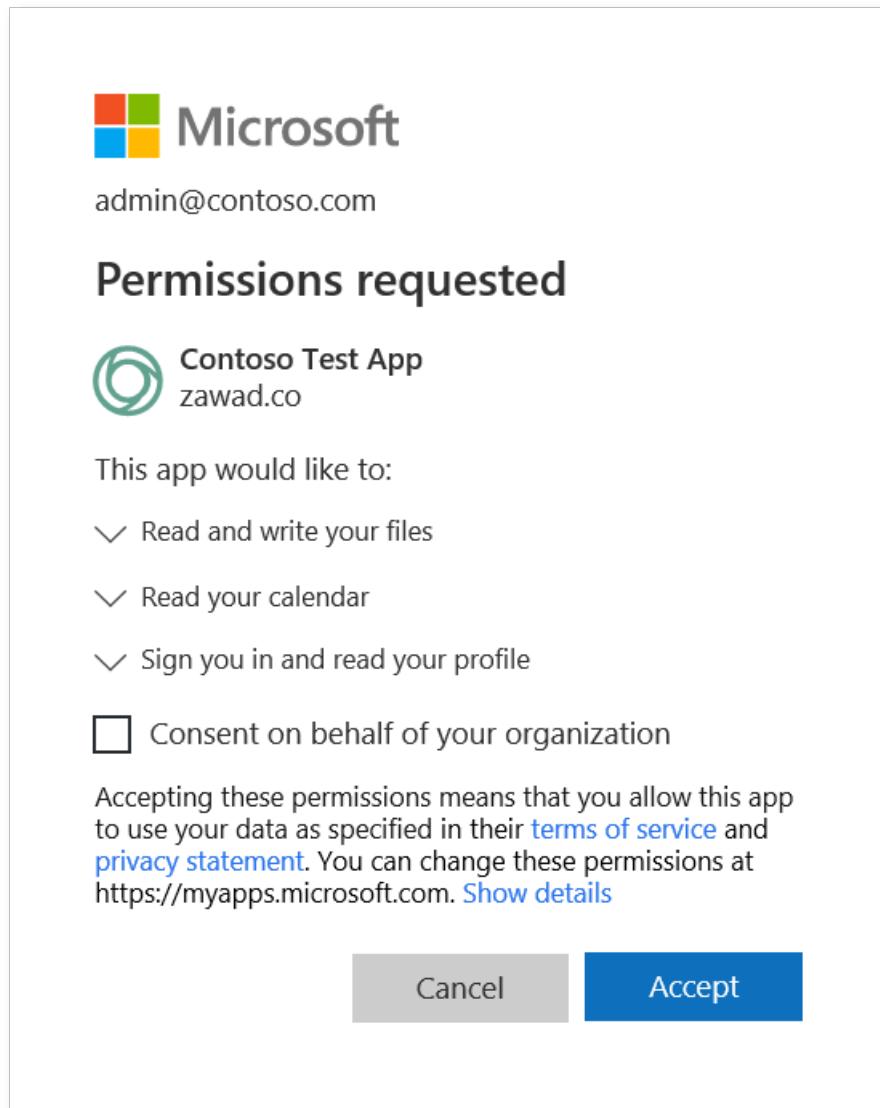
| # | COMPONENT | PURPOSE |
|---|---|--|
| 4 | App name | This value should inform users which application is requesting access to their data. Note this name is provided by the developers and the ownership of this app name isn't validated. |
| 5 | Publisher domain | This value should provide users with a domain they may be able to evaluate for trustworthiness. This domain is provided by the developers and the ownership of this publisher domain is validated. |
| 6 | Permissions | This list contains the permissions being requested by the client application. Users should always evaluate the types of permissions being requested to understand what data the client application will be authorized to access on their behalf if they accept. As an application developer it is best to request access, to the permissions with the least privilege. |
| 7 | Permission description | This value is provided by the service exposing the permissions. To see the permission descriptions, you must toggle the chevron next to the permission. |
| 8 | App terms | These terms contain links to the terms of service and privacy statement of the application. The publisher is responsible for outlining their rules in their terms of service. Additionally, the publisher is responsible for disclosing the way they use and share user data in their privacy statement. If the publisher doesn't provide links to these values for multi-tenant applications, there will be a bolded warning on the consent prompt. |
| 9 | https://myapps.microsoft.com | This is the link where users can review and remove any non-Microsoft applications that currently have access to their data. |

Common consent scenarios

Here are the consent experiences that a user may see in the common consent scenarios:

1. Individuals accessing an app that directs them to the user consent flow while requiring a permission set that is within their scope of authority.
 - a. Admins will see an additional control on the traditional consent prompt that will allow them consent on behalf of the entire tenant. The control will be defaulted to off, so only when admins explicitly check the box will consent be granted on behalf of the entire tenant. As of today, this check box will

only show for the Global Admin role, so Cloud Admin and App Admin will not see this checkbox.



- b. Users will see the traditional consent prompt.



user@contoso.com

Permissions requested



Contoso Test App
zawad.co

This app would like to:

- ✓ Read and write your files
- ✓ Read your calendar
- ✓ Sign you in and read your profile

Accepting these permissions means that you allow this app to use your data as specified in their [terms of service](#) and [privacy statement](#). You can change these permissions at <https://myapps.microsoft.com>. [Show details](#)

Cancel

Accept

2. Individuals accessing an app that requires at least one permission that is outside their scope of authority.
 - a. Admins will see the same prompt as 1.i shown above.
 - b. Users will be blocked from granting consent to the application, and they will be told to ask their admin for access to the app.



user@contoso.com

Need admin approval



Contoso Test App

zawad.co

Contoso Test App needs permission to access resources in your organization that only an admin can grant. Please ask an admin to grant permission to this app before you can use it.

[Have an admin account? Sign in with that account](#)

[Return to the application without granting consent](#)

3. Individuals that navigate or are directed to the admin consent flow.

- a. Admin users will see the admin consent prompt. The title and the permission descriptions changed on this prompt, the changes highlight the fact that accepting this prompt will grant the app access to the requested data on behalf of the entire tenant.



user@contoso.com

Permissions requested Accept for your organization



Contoso Test App

zawad.co

This app would like to:

- ✓ Read user and shared contacts
- ✓ Read user and shared calendars
- ✓ Sign in and read user profile

Accepting these permissions means that you allow this app to use your data as specified in their [terms of service](#) and [privacy statement](#). You can change these permissions at <https://myapps.microsoft.com>. [Show details](#)

Cancel

Accept

- b. Non-admin users will see the same screen as 2.ii shown above.

Next steps

- Get a step-by-step overview of [how the Azure AD consent framework implements consent](#).
- For more depth, learn [how a multi-tenant application can use the consent framework to implement "user" and "admin" consent](#), supporting more advanced multi-tier application patterns.
- Learn [how to configure the app's publisher domain](#).

National clouds

10/23/2019 • 2 minutes to read • [Edit Online](#)

National clouds are physically isolated instances of Azure. These regions of Azure are designed to make sure that data residency, sovereignty, and compliance requirements are honored within geographical boundaries.

Including the global cloud, Azure ActiveDirectory(Azure AD) is deployed in the following national clouds:

- Azure Government
- Azure Germany
- Azure China 21Vianet

National clouds are unique and a separate environment from Azure global. It's important to be aware of key differences while developing your application for these environments. Differences include registering applications, acquiring tokens, and configuring endpoints.

App registration endpoints

There's a separate Azure portal for each one of the national clouds. To integrate applications with the Microsoft identity platform in a national cloud, you're required to register your application separately in each Azure portal that's specific to the environment.

The following table lists the base URLs for the Azure AD endpoints used to register an application for each national cloud.

| NATIONAL CLOUD | AZURE AD PORTAL ENDPOINT |
|-------------------------------------|---|
| Azure AD for US Government | https://portal.azure.us |
| Azure AD Germany | https://portal.microsoftazure.de |
| Azure AD China operated by 21Vianet | https://portal.azure.cn |
| Azure AD (global service) | https://portal.azure.com |

Azure AD authentication endpoints

All the national clouds authenticate users separately in each environment and have separate authentication endpoints.

The following table lists the base URLs for the Azure AD endpoints used to acquire tokens for each national cloud.

| NATIONAL CLOUD | AZURE AD AUTHENTICATION ENDPOINT |
|-------------------------------------|---|
| Azure AD for US Government | https://login.microsoftonline.us |
| Azure AD Germany | https://login.microsoftonline.de |
| Azure AD China operated by 21Vianet | https://login.chinacloudapi.cn |

| NATIONAL CLOUD | AZURE AD AUTHENTICATION ENDPOINT |
|---------------------------|---|
| Azure AD (global service) | https://login.microsoftonline.com |

You can form requests to the Azure AD authorization or token endpoints by using the appropriate region-specific base URL. For example, for Azure Germany:

- Authorization common endpoint is <https://login.microsoftonline.de/common/oauth2/authorize>.
- Token common endpoint is <https://login.microsoftonline.de/common/oauth2/token>.

For single-tenant applications, replace "common" in the previous URLs with your tenant ID or name. An example is <https://login.microsoftonline.de/contoso.com>.

Microsoft Graph API

To learn how to call the Microsoft Graph APIs in a national cloud environment, go to [Microsoft Graph in national cloud deployments](#).

IMPORTANT

Certain services and features that are in specific regions of the global service might not be available in all of the national clouds. To find out what services are available, go to [Products available by region](#).

To learn how to build an application by using the Microsoft identity platform, follow the [Microsoft Authentication Library \(MSAL\) tutorial](#). Specifically, this app will sign in a user and get an access token to call the Microsoft Graph API.

Next steps

Learn more about:

- [Azure Government](#)
- [Azure China 21Vianet](#)
- [Azure Germany](#)
- [Azure AD authentication basics](#)

Use MSAL in a national cloud environment

9/25/2019 • 4 minutes to read • [Edit Online](#)

National clouds are physically isolated instances of Azure. These regions of Azure help make sure that data residency, sovereignty, and compliance requirements are honored within geographical boundaries.

In addition to the Microsoft worldwide cloud, the Microsoft Authentication Library (MSAL) enables application developers in national clouds to acquire tokens in order to authenticate and call secured web APIs. These web APIs can be Microsoft Graph or other Microsoft APIs.

Including the global cloud, Azure ActiveDirectory (Azure AD) is deployed in the following national clouds:

- Azure Government
- Azure China 21Vianet
- Azure Germany

This guide demonstrates how to sign in to work and school accounts, get an access token, and call the Microsoft Graph API in the [Azure Government cloud](#) environment.

Prerequisites

Before you start, make sure that you meet these prerequisites.

Choose the appropriate identities

Azure Government applications can use Azure AD Government identities and Azure AD Public identities to authenticate users. Because you can use any of these identities, you need to decide which authority endpoint you should choose for your scenario:

- Azure AD Public: Commonly used if your organization already has an Azure AD Public tenant to support Office 365 (Public or GCC) or another application.
- Azure AD Government: Commonly used if your organization already has an Azure AD Government tenant to support Office 365 (GCC High or DoD) or is creating a new tenant in Azure AD Government.

After you decide, a special consideration is where you perform your app registration. If you choose Azure AD Public identities for your Azure Government application, you must register the application in your Azure AD Public tenant.

Get an Azure Government subscription

To get an Azure Government subscription, see [Managing and connecting to your subscription in Azure Government](#).

If you don't have an Azure Government subscription, create a [free account](#) before you begin.

JavaScript

Step 1: Register your application

1. Sign in to the [Azure portal](#).

To find Azure portal endpoints for other national clouds, see [App registration endpoints](#).

2. If your account gives you access to more than one tenant, select your account in the upper-right corner, and set your portal session to the desired Azure AD tenant.

3. Go to the [App registrations](#) page on the Microsoft identity platform for developers.
4. When the **Register an application** page appears, enter a name for your application.
5. Under **Supported account types**, select **Accounts in any organizational directory**.
6. In the **Redirect URI** section, select the **Web** platform and set the value to the application's URL based on your web server. See the next sections for instructions on how to set and obtain the redirect URL in Visual Studio and Node.
7. Select **Register**.
8. On the app **Overview** page, note down the **Application (client) ID** value.
9. This tutorial requires you to enable the [implicit grant flow](#). In the left pane of the registered application, select **Authentication**.
10. In **Advanced settings**, under **Implicit grant**, select the **ID tokens** and **Access tokens** check boxes. ID tokens and access tokens are required because this app needs to sign in users and call an API.
11. Select **Save**.

Step 2: Set up your web server or project

- [Download the project files](#) for a local web server, such as Node.
or
- [Download the Visual Studio project](#).

Then skip to [Configure your JavaScript SPA](#) to configure the code sample before running it.

Step 3: Use the Microsoft Authentication Library to sign in the user

Follow steps in the [JavaScript tutorial](#) to create your project and integrate with MSAL to sign in the user.

Step 4: Configure your JavaScript SPA

In the `index.html` file created during project setup, add the application registration information. Add the following code at the top within the `<script></script>` tags in the body of your `index.html` file:

```
const msalConfig = {
    auth:{
        clientId: "Enter_the_Application_Id_here",
        authority: "https://login.microsoftonline.us/Enter_the_Tenant_Info_Here",
    }
}

const graphConfig = {
    graphEndpoint: "https://graph.microsoft.us",
    graphScopes: ["user.read"],
}

// create UserAgentApplication instance
const myMSALObj = new UserAgentApplication(msalConfig);
```

In that code:

- `Enter_the_Application_Id_here` is the **Application (client) ID** value for the application that you registered.
- `Enter_the_Tenant_Info_Here` is set to one of the following options:
 - If your application supports **Accounts in this organizational directory**, replace this value with the tenant ID or tenant name (for example, contoso.microsoft.com).

- If your application supports **Accounts in any organizational directory**, replace this value with `organizations`.

To find authentication endpoints for all the national clouds, see [Azure AD authentication endpoints](#).

NOTE

Personal Microsoft accounts are not supported in national clouds.

- `graphEndpoint` is the Microsoft Graph endpoint for the Microsoft cloud for US government.

To find Microsoft Graph endpoints for all the national clouds, see [Microsoft Graph endpoints in national clouds](#).

.NET

You can use MSAL.NET to sign in users, acquire tokens, and call the Microsoft Graph API in national clouds.

The following tutorials demonstrate how to build a .NET Core 2.2 MVC Web app. The app uses OpenID Connect to sign in users with a work and school account in an organization that belongs to a national cloud.

- To sign in users and acquire tokens, follow [this tutorial](#).
- To call the Microsoft Graph API, follow [this tutorial](#).

MSAL for iOS and macOS

MSAL for iOS and macOS can be used to acquire tokens in national clouds, but it requires additional configuration when creating `MSALPublicClientApplication`.

For instance, if you want your application to be a multi-tenant application in a national cloud (here US Government), you could write:

Objective-C:

```
MSALAADAuthority *aadAuthority =
    [[MSALAADAuthority alloc] initWithCloudInstance:MSALAzureUsGovernmentCloudInstance
                                            audienceType:MSALAzureADMultipleOrgsAudience
                                              rawTenant:nil
                                              error:nil];

MSALPublicClientApplicationConfig *config =
    [[MSALPublicClientApplicationConfig alloc] initWithClientId:@"<your-client-id-here>"
                                                redirectUri:@"<your-redirect-uri-here>"
                                              authority:aadAuthority];

NSError *applicationError = nil;
MSALPublicClientApplication *application =
    [[MSALPublicClientApplication alloc] initWithConfiguration:config error:&applicationError];
```

Swift:

```
let authority = try? MSALAADAuthority(cloudInstance: .usGovernmentCloudInstance, audienceType:
    .azureADMultipleOrgsAudience, rawTenant: nil)

let config = MSALPublicClientApplicationConfig(clientId: "<your-client-id-here>", redirectUri: "<your-
    redirect-uri-here>", authority: authority)
if let application = try? MSALPublicClientApplication(configuration: config) { /* Use application */}
```

Next steps

Learn more about:

- [Azure Government](#)
- [Azure China 21Vianet](#)
- [Azure Germany](#)

How to: Configure the role claim issued in the SAML token for enterprise applications

10/23/2019 • 6 minutes to read • [Edit Online](#)

By using Azure Active Directory (Azure AD), you can customize the claim type for the role claim in the response token that you receive after you authorize an app.

Prerequisites

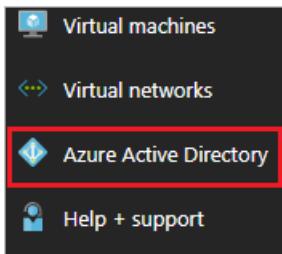
- An Azure AD subscription with directory setup.
- A subscription that has single sign-on (SSO) enabled. You must configure SSO with your application.

When to use this feature

If your application expects custom roles to be passed in a SAML response, you need to use this feature. You can create as many roles as you need to be passed back from Azure AD to your application.

Create roles for an application

1. In the [Azure portal](#), in the left pane, select the **Azure Active Directory** icon.



2. Select **Enterprise applications**. Then select **All applications**.

The screenshot shows two side-by-side Azure Active Directory management pages. The left page is titled 'Contoso Azure Active Directory' and has a sidebar with various icons. The right page is titled 'Enterprise applications Contoso - Azure Active Directory' and lists several management sections: Overview, Quick start, MANAGE (with 'All applications' highlighted), Application proxy, Conditional access, ACTIVITY (Sign-ins, Audit logs), SUPPORT + TROUBLESHOOTING (New support request, Troubleshoot). The 'Enterprise applications' link in the sidebar of the left page is also highlighted.

3. To add a new application, select the **New application** button on the top of the dialog box.



4. In the search box, type the name of your application, and then select your application from the result panel. Select the **Add** button to add the application.

A screenshot of the 'Add from the gallery' dialog box. The search bar contains a placeholder text. Below it, a message says '1 applications matched [redacted]. Choose one below or...'. A table follows, with columns 'NAME' and 'CATEGORY'. One row is visible, with the 'NAME' column highlighted by a red box.

5. After the application is added, go to the **Properties** page and copy the object ID.

MANAGE

- Properties **Properties**
- Users and groups
- Single sign-on
- Provisioning
- Self-service

SECURITY

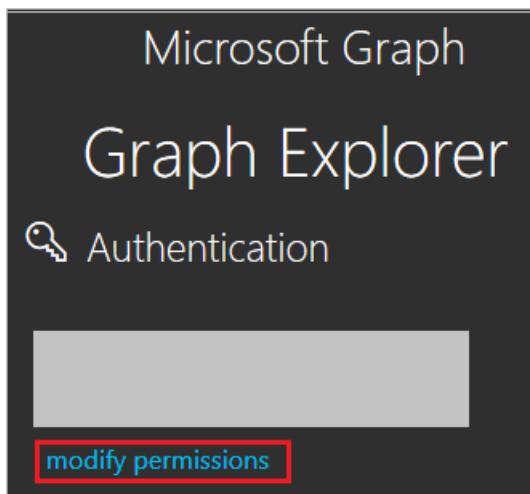
- Conditional access
- Permissions

ACTIVITY

- Sign-ins
- Audit logs

6. Open [Microsoft Graph Explorer](#) in another window and take the following steps:

- Sign in to the Graph Explorer site by using the global admin or coadmin credentials for your tenant.
- You need sufficient permissions to create the roles. Select **modify permissions** to get the permissions.



- Select the following permissions from the list (if you don't have these already) and select **Modify Permissions**.

| | |
|--|-------|
| <input checked="" type="checkbox"/> Directory.AccessAsUser.All | Admin |
| <input checked="" type="checkbox"/> Directory.Read.All | Admin |
| <input checked="" type="checkbox"/> Directory.ReadWrite.All | Admin |
| <input type="checkbox"/> Group.Read.All | Admin |

(i) You have selected permissions that only an administrator can grant. To get access, an administrator can grant [access to your entire organization](#).

Modify Permissions **Close**

NOTE

Cloud App Administrator and App Administrator role will not work in this scenario as we need the Global Admin permissions for Directory Read and Write.

d. Accept the consent. You're logged in to the system again.

e. Change the version to **beta**, and fetch the list of service principals from your tenant by using the following query:

```
https://graph.microsoft.com/beta/servicePrincipals
```

If you're using multiple directories, follow this pattern:

```
https://graph.microsoft.com/beta/contoso.com/servicePrincipals
```

The screenshot shows the Microsoft Graph Explorer interface. The URL is set to `https://graph.microsoft.com/beta/servicePrincipals`. The request method is `GET` and the version is `beta`. The response body shows a single service principal object:

```
{  
    "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity",  
    "id": "8164e784-8a01-46c9-89fd-3076bd9656d0",  
    "deletedDateTime": null,  
    "accountEnabled": true,  
    "appDisplayName": "",  
    "appId": "8e1d26f3-9519-4d66-8577-65fa3261c7ff",  
    "appOwnerOrganizationId": "0ac53016-3006-4227-9eeb-89d63f8055b6",  
    "appRoleAssignmentRequired": true,  
    "displayName": ""  
}
```

The status bar indicates `Success - Status Code 200, 6146ms`.

NOTE

We are already in the process of upgrading the APIs so customers might see some disruption in the service.

f. From the list of fetched service principals, get the one that you need to modify. You can also use Ctrl+F to search the application from all the listed service principals. Search for the object ID that you copied from the **Properties** page, and use the following query to get to the service principal:

```
https://graph.microsoft.com/beta/servicePrincipals/<objectID>
```

GET beta https://graph.microsoft.com/beta/servicePrincipals/8164e784-8a01-46c9-89fd-3076bd9656d0

Request Body Request Headers

```
{
    "customKeyIdentifier": "SUFNX1NTMF80LzYvMjAxOCAxMjoxMTozNyBBTQ==",
    "endDate": "2021-04-06T00:11:36Z",
    "keyId": "e101c4ec-3e89-4d3b-9ba3-4f524cb6eeda",
    "startDateTime": "2018-04-06T00:11:36Z",
    "secretText": null,
    "hint": null
}
```

Success - Status Code 200, 312ms

Response Preview Response Headers

```
* {
    "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity",
    "id": "8164e784-8a01-46c9-89fd-3076bd9656d0",
    "deletedDateTime": null,
    "accountEnabled": true,
```

g. Extract the **appRoles** property from the service principal object.

```
"appRoles": [
    {
        "allowedMemberTypes": [
            "User"
        ],
        "description": "msiam_access",
        "displayName": "msiam_access",
        "id": "7dfd756e-8c27-4472-b2b7-38c17fc5de5e",
        "isEnabled": true,
        "origin": "Application",
        "value": null
    },
    ...
]
```

NOTE

If you're using the custom app (not the Azure Marketplace app), you see two default roles: user and msiam_access. For the Marketplace app, msiam_access is the only default role. You don't need to make any changes in the default roles.

h. Generate new roles for your application.

The following JSON is an example of the **appRoles** object. Create a similar object to add the roles that you want for your application.

```
{
  "appRoles": [
    {
      "allowedMemberTypes": [
        "User"
      ],
      "description": "msiam_access",
      "displayName": "msiam_access",
      "id": "b9632174-c057-4f7e-951b-be3adc52bfe6",
      "isEnabled": true,
      "origin": "Application",
      "value": null
    },
    {
      "allowedMemberTypes": [
        "User"
      ],
      "description": "Administrators Only",
      "displayName": "Admin",
      "id": "4f8f8640-f081-492d-97a0-caf24e9bc134",
      "isEnabled": true,
      "origin": "ServicePrincipal",
      "value": "Administrator"
    }
  ]
}
```

NOTE

You can only add new roles after msiam_access for the patch operation. Also, you can add as many roles as your organization needs. Azure AD will send the value of these roles as the claim value in the SAML response. To generate the GUID values for the ID of new roles use the web tools like [this](#)

- i. Go back to Graph Explorer and change the method from **GET** to **PATCH**. Patch the service principal object to have the desired roles by updating the **appRoles** property like the one shown in the preceding example. Select **Run Query** to execute the patch operation. A success message confirms the creation of the role.

The screenshot shows the Microsoft Graph Explorer interface. The top bar has 'PATCH' selected, 'beta' dropdown, URL 'https://graph.microsoft.com/beta/servicePrincipals/8164e784-8a01-46c9-89fd-3076bd9656d0', and a 'Run Query' button. Below the URL is a 'Request Body' tab with the JSON payload from the previous code block. The bottom status bar shows 'Success - Status Code 204, 776ms'.

7. After the service principal is patched with more roles, you can assign users to the respective roles. You can assign the users by going to portal and browsing to the application. Select the **Users and groups** tab. This tab lists all the users and groups that are already assigned to the app. You can add new users on the new roles. You can also select an existing user and select **Edit** to change the role.

To assign the role to any user, select the new role and select the **Assign** button on the bottom of the page.

NOTE

You need to refresh your session in the Azure portal to see new roles.

8. Update the **Attributes** table to define a customized mapping of the role claim.
9. In the **User Claims** section on the **User Attributes** dialog, perform the following steps to add SAML token attribute as shown in the below table:

| ATTRIBUTE NAME | ATTRIBUTE VALUE |
|----------------|--------------------|
| Role name | user.assignedroles |

NOTE

If the role claim value is null, then Azure AD will not send this value in the token and this is default as per design.

- a. click **Edit** icon to open **User Attributes & Claims** dialog.

2

| User Attributes | |
|------------------------|------------------------|
| givenname | user.givenname |
| surname | user.surname |
| emailaddress | user.mail |
| name | user.userprincipalname |
| Unique User Identifier | user.userprincipalname |



b. In the **Manage user claims** dialog, add the SAML token attribute by clicking on **Add new claim**.

User Attributes & Claims

+ Add new claim

Manage user claims

* Name:

Namespace:

Source: Attribute Transformation

* Source attribute:

Save

c. In the **Name** box, type the attribute name as needed. This example uses **Role Name** as the claim name.

d. Leave the **Namespace** box blank.

e. From the **Source attribute** list, type the attribute value shown for that row.

f. Select **Save**.

10. To test your application in a single sign-on that's initiated by an identity provider, sign in to the [Access Panel](#) and select your application tile. In the SAML token, you should see all the assigned roles for the user with the claim name that you have given.

Update an existing role

To update an existing role, perform the following steps:

1. Open [Microsoft Graph Explorer](#).

- Sign in to the Graph Explorer site by using the global admin or coadmin credentials for your tenant.
- Change the version to **beta**, and fetch the list of service principals from your tenant by using the following query:

```
https://graph.microsoft.com/beta/servicePrincipals
```

If you're using multiple directories, follow this pattern:

```
https://graph.microsoft.com/beta/contoso.com/servicePrincipals
```

Request Body

```
{
  "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity",
  "id": "8164e784-8a01-46c9-89fd-3076bd9656d0",
  "deletedDateTime": null,
  "accountEnabled": true,
  "appDisplayName": "",
  "appId": "8e1a2ef3-9519-4d66-8577-65fa3261c7ff",
  "appOwnerOrganizationId": "0ac53016-3006-4227-9eeb-89d63f8055b6",
  "appRoleAssignmentRequired": true,
  "displayName": ""
}
```

Response Preview

```
{
  "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals",
  "@odata.nextLink": "https://graph.microsoft.com/beta/servicePrincipals?$skiptoken=X%274453707402000100000035536572766963655072696E36970616C5F31643466396561652D32&$top=1",
  "value": [
    {
      "id": "00008fa9-7197-4f17-b74b-cf702e697ddc",
      "deletedDateTime": null,
      "accountEnabled": true
    }
  ]
}
```

- From the list of fetched service principals, get the one that you need to modify. You can also use Ctrl+F to search the application from all the listed service principals. Search for the object ID that you copied from the **Properties** page, and use the following query to get to the service principal:

```
https://graph.microsoft.com/beta/servicePrincipals/<objectID>
```

Request Body

```
{
  "customKeyIdentifier": "SUFNX1NTMF80LzYvMjAxOCAxMjoxMTozNyBBTQ==",
  "endDateTime": "2021-04-06T00:11:36Z",
  "keyId": "e101c4ec-3e89-4d3b-9ba3-4f524cb6eeda",
  "startDateTime": "2018-04-06T00:11:36Z",
  "secretText": null,
  "hint": null
}
```

Response Preview

```
{
  "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity",
  "id": "8164e784-8a01-46c9-89fd-3076bd9656d0",
  "deletedDateTime": null,
  "accountEnabled": true
}
```

- Extract the **appRoles** property from the service principal object.

```
"appRoles": [
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "msiam_access",
    "displayName": "msiam_access",
    "id": "7dfd756e-8c27-4472-b2b7-38c17fc5de5e",
    "isEnabled": true,
    "origin": "Application",
    "value": null
  },
]
```

6. To update the existing role, use the following steps.

The screenshot shows the Microsoft Graph Explorer interface. The method dropdown is set to **PATCH**, the version dropdown is set to **beta**, and the URL is **https://graph.microsoft.com/beta/servicePrincipals/**. The **Run Query** button is highlighted with a red box. Below the URL, there are two tabs: **Request Body** (selected) and **Request Headers**. The Request Body contains a JSON object with a single role definition. The **description** and **displayName** fields are highlighted with red boxes, indicating they are the values to be updated.

```
{"appRoles": [
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "student",
    "displayName": "student",
    "id": "631a14ab-5319-4bb1-a668-5626904561be",
    "isEnabled": true,
    "origin": "ServicePrincipal"
  }
]}
```

- Change the method from **GET** to **PATCH**.
- Copy the existing roles and paste them under **Request Body**.
- Update the value of a role by updating the role description, role value, or role display name as needed.
- After you update all the required roles, select **Run Query**.

Delete an existing role

To delete an existing role, perform the following steps:

- Open [Microsoft Graph Explorer](#) in another window.
- Sign in to the Graph Explorer site by using the global admin or coadmin credentials for your tenant.
- Change the version to **beta**, and fetch the list of service principals from your tenant by using the following query:

```
https://graph.microsoft.com/beta/servicePrincipals
```

If you're using multiple directories, follow this pattern:

```
https://graph.microsoft.com/beta/contoso.com/servicePrincipals
```

The screenshot shows the Microsoft Graph Explorer interface. The URL bar contains `https://graph.microsoft.com/beta/servicePrincipals`. The request method is set to `GET` and the version is `beta`. A red box highlights the `beta` dropdown. The `Request Body` section is empty. The `Request Headers` section is also empty. The `Run Query` button is at the top right. Below the request area, a green status bar indicates `Success - Status Code 200, 6146ms`. The `Response Preview` section shows the JSON response, which includes the service principal's ID, deleted date, account enabled status, and app display name. A red box highlights the `beta` dropdown in the URL bar.

```
{
  "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity",
  "id": "8164e784-8a01-46c9-89fd-3076bd9656d0",
  "deletedDateTime": null,
  "accountEnabled": true,
  "appDisplayName": "",
  "appId": "8e1d26f3-9519-4d66-8577-65fa3261c7ff",
  "appOwnerOrganizationId": "0ac53016-3006-4227-9eeb-89d63f8055b6",
  "appRoleAssignmentRequired": true,
  "displayName": ""
}
```

4. From the list of fetched service principals, get the one that you need to modify. You can also use `Ctrl+F` to search the application from all the listed service principals. Search for the object ID that you copied from the **Properties** page, and use the following query to get to the service principal:

The screenshot shows the Microsoft Graph Explorer interface. The URL bar contains `https://graph.microsoft.com/beta/servicePrincipals/8164e784-8a01-46c9-89fd-3076bd9656d0`. The request method is set to `GET` and the version is `beta`. A red box highlights the `beta` dropdown. The `Request Body` section is empty. The `Request Headers` section is also empty. The `Run Query` button is at the top right. Below the request area, a green status bar indicates `Success - Status Code 200, 312ms`. The `Response Preview` section shows the JSON response, which includes the service principal's ID, deleted date, account enabled status, and app roles. A red box highlights the `beta` dropdown in the URL bar.

```
{
  "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity",
  "id": "8164e784-8a01-46c9-89fd-3076bd9656d0",
  "deletedDateTime": null,
  "accountEnabled": true,
  "appRoles": [
    {
      "customKeyIdentifier": "SUFNX1NTMF80LzYvMjAxOCAxMjoxMTozNyBBTQ==",
      "endDateTime": "2021-04-06T00:11:36Z",
      "keyId": "e101c4ec-3e89-4d3b-9ba3-4f524cb6eeda",
      "startDateTime": "2018-04-06T00:11:36Z",
      "secretText": null,
      "hint": null
    }
  ]
}
```

5. Extract the **appRoles** property from the service principal object.

```
{
  "allowedMemberTypes": [
    "User"
  ],
  "description": "msiam_access",
  "displayName": "msiam_access",
  "id": "b9632174-c057-4f7e-951b-be3adc52bfe6",
  "isEnabled": true,
  "origin": "Application",
  "value": null
},
{
  "allowedMemberTypes": [
    "User"
  ],
  "description": "Administrators Only",
  "displayName": "Admin",
  "id": "4f8f8640-f081-492d-97a0-caf24e9bc134",
  "isEnabled": true,
  "origin": "ServicePrincipal",
  "value": "Administrator"
}
}
```

6. To delete the existing role, use the following steps.

The screenshot shows the Microsoft Graph Explorer interface. The URL is https://graph.microsoft.com/beta/servicePrincipals/8164e784-8a01-46c9-89fd-3076bd9656d0. The method is set to PATCH. The Request Body contains the following JSON:

```

PATCH beta https://graph.microsoft.com/beta/servicePrincipals/8164e784-8a01-46c9-89fd-3076bd9656d0
Request Body Request Headers
{
  "allowedMemberTypes": [
    "User"
  ],
  "description": "Administrators Only",
  "displayName": "Admin",
  "id": "4f8f8640-f081-492d-97a0-caf24e9bc134",
  "isEnabled": false,
  "origin": "ServicePrincipal",
  "value": "Administrator"
}

```

- Change the method from **GET** to **PATCH**.
- Copy the existing roles from the application and paste them under **Request Body**.
- Set the **Enabled** value to **false** for the role that you want to delete.
- Select **Run Query**.

NOTE

Make sure that you have the msiam_access role, and the ID is matching in the generated role.

7. After the role is disabled, delete that role block from the **appRoles** section. Keep the method as **PATCH**, and select **Run Query**.

8. After you run the query, the role is deleted.

NOTE

The role needs to be disabled before it can be removed.

Next steps

For additional steps, see the [app documentation](#).

How to: customize claims issued in the SAML token for enterprise applications

10/24/2019 • 10 minutes to read • [Edit Online](#)

Today, Azure Active Directory (Azure AD) supports single sign-on (SSO) with most enterprise applications, including both applications pre-integrated in the Azure AD app gallery as well as custom applications. When a user authenticates to an application through Azure AD using the SAML 2.0 protocol, Azure AD sends a token to the application (via an HTTP POST). And then, the application validates and uses the token to log the user in instead of prompting for a username and password. These SAML tokens contain pieces of information about the user known as *claims*.

A *claim* is information that an identity provider states about a user inside the token they issue for that user. In [SAML token](#), this data is typically contained in the SAML Attribute Statement. The user's unique ID is typically represented in the SAML Subject also called as Name Identifier.

By default, Azure AD issues a SAML token to your application that contains a `NameIdentifier` claim with a value of the user's username (also known as the user principal name) in Azure AD, which can uniquely identify the user. The SAML token also contains additional claims containing the user's email address, first name, and last name.

To view or edit the claims issued in the SAML token to the application, open the application in Azure portal. Then open the **User Attributes & Claims** section.

The screenshot shows the 'User Attributes & Claims' configuration page in the Azure portal. At the top, there are navigation links: 'Change single sign-on mode', 'Switch to the old experience', and 'Test this application'. A purple banner at the top says 'Welcome to the new experience for configuring SAML based SSO. Please click here to provide feedback.' Below the banner, the title 'Set up Single Sign-On with SAML - Preview' is displayed. A note below it says 'Read the [configuration guide](#) for help integrating Atlassian Cloud.' The configuration is divided into two sections:

- 1 Basic SAML Configuration**: This section contains fields for 'Reply URL (Assertion Consumer Service URL)', 'Identifier (Entity ID)', 'Sign on URL', and 'Relay State'. Each field has a corresponding value and an optional note: 'Optional'.
- 2 User Attributes & Claims**: This section lists user attributes and their corresponding SAML claim URIs. The attributes are: Givenname, Surname, Emailaddress, Name, and Unique User Identifier. The claim URIs are: user.givenname, user.surname, user.mail, user.userprincipalname, and user.userprincipalname respectively.

There are two possible reasons why you might need to edit the claims issued in the SAML token:

- The application requires the `NameIdentifier` or NameID claim to be something other than the username (or user principal name) stored in Azure AD.
- The application has been written to require a different set of claim URIs or claim values.

Editing nameID

To edit the NameID (name identifier value):

1. Open the **Name identifier value** page.
2. Select the attribute or transformation you want to apply to the attribute. Optionally, you can specify the format you want the NameID claim to have.

Manage claim

Save Discard changes

* Name:

Namespace:

Choose name identifier format

* Source: Attribute Transformation

* Source attribute:

Claim Conditions

NameID format

If the SAML request contains the element NameIDPolicy with a specific format, then Azure AD will honor the format in the request.

If the SAML request doesn't contain an element for NameIDPolicy, then Azure AD will issue the NameID with the format you specify. If no format is specified Azure AD will use the default source format associated with the claim source selected.

From the **Choose name identifier format** dropdown, you can select one of the following options.

| NAMEID FORMAT | DESCRIPTION |
|---------------|--|
| Default | Azure AD will use the default source format. |
| Persistent | Azure AD will use Persistent as the NameID format. |
| EmailAddress | Azure AD will use EmailAddress as the NameID format. |
| Unspecified | Azure AD will use Unspecified as the NameID format. |

Transient NameID is also supported, but is not available in the dropdown and cannot be configured on Azure's side. To learn more about the NameIDPolicy attribute, see [Single Sign-On SAML protocol](#).

Attributes

Select the desired source for the **NameIdentifier** (or NameID) claim. You can select from the following options.

| NAME | DESCRIPTION |
|---------------------------|---|
| Email | Email address of the user |
| userprincipalName | User principal name (UPN) of the user |
| onpremisessamaccount | SAM account name that has been synced from on-premises Azure AD |
| objectid | Objectid of the user in Azure AD |
| employeeid | Employee ID of the user |
| Directory extensions | Directory extensions synced from on-premises Active Directory using Azure AD Connect Sync |
| Extension Attributes 1-15 | On-premises extension attributes used to extend the Azure AD schema |

For more info, see [Table 3: Valid ID values per source](#).

You can also assign any constant (static) value to any claims which you define in Azure AD. Please follow the below steps to assign a constant value:

1. In the [Azure portal](#), on the **User Attributes & Claims** section, click on the **Edit** icon to edit the claims.
2. Click on the required claim which you want to modify.
3. Enter the constant value without quotes in the **Source attribute** as per your organization and click **Save**.

Manage claim

Save **Discard changes**

| | | |
|-------------------------|---|---|
| ★ Name | OrganizationID | ✓ |
| Namespace | http://schemas.xmlsoap.org/ws/2005... | |
| ★ Source | <input checked="" type="radio"/> Attribute <input type="radio"/> Transformation | |
| ★ Source attribute | "Contoso1234" | |
| Claim conditions | | |

4. The constant value will be displayed as below.

2 User Attributes & Claims

| | |
|------------------------|------------------------|
| Givenname | user.givenname |
| Surname | user.surname |
| Emailaddress | user.mail |
| Name | user.userprincipalname |
| OrganizationID | "Contoso1234" |
| Unique User Identifier | user.userprincipalname |

Special claims - transformations

You can also use the claims transformations functions.

| FUNCTION | DESCRIPTION |
|----------------------------|---|
| ExtractMailPrefix() | Removes the domain suffix from either the email address or the user principal name. This extracts only the first part of the user name being passed through (for example, "joe_smith" instead of joe_smith@contoso.com). |
| Join() | Joins an attribute with a verified domain. If the selected user identifier value has a domain, it will extract the username to append the selected verified domain. For example, if you select the email (joe_smith@contoso.com) as the user identifier value and select contoso.onmicrosoft.com as the verified domain, this will result in joe_smith@contoso.onmicrosoft.com. |
| ToLower() | Converts the characters of the selected attribute into lowercase characters. |
| ToUpper() | Converts the characters of the selected attribute into uppercase characters. |

Adding application-specific claims

To add application-specific claims:

1. In **User Attributes & Claims**, select **Add new claim** to open the **Manage user claims** page.
2. Enter the **name** of the claims. The value doesn't strictly need to follow a URI pattern, per the SAML spec. If you need a URI pattern, you can put that in the **Namespace** field.
3. Select the **Source** where the claim is going to retrieve its value. You can select a user attribute from the source attribute dropdown or apply a transformation to the user attribute before emitting it as a claim.

Claim transformations

To apply a transformation to a user attribute:

1. In **Manage claim**, select *Transformation* as the claim source to open the **Manage transformation** page.
2. Select the function from the transformation dropdown. Depending on the function selected, you will have to provide parameters and a constant value to evaluate in the transformation. Refer to the table below for more information about the available functions.
3. To apply multiple transformation, click on **Add transformation**. You can apply a maximum of two transformation to a claim. For example, you could first extract the email prefix of the `user.mail`. Then, make the string upper case.

Manage transformation

| | |
|--|-------------------------------------|
| * Transformation | ExtractMailPrefix() |
| * Parameter 1 | user.mail |
| * Transformation | ToUppercase() |
| Parameter 1 | Output from previous transformation |
| ✖ Remove transformation | |

You can use the following functions to transform claims.

| FUNCTION | DESCRIPTION |
|----------------------------|---|
| ExtractMailPrefix() | Removes the domain suffix from either the email address or the user principal name. This extracts only the first part of the user name being passed through (for example, "joe_smith" instead of joe_smith@contoso.com). |
| Join() | Creates a new value by joining two attributes. Optionally, you can use a separator between the two attributes. For NameID claim transformation, the join is restricted to a verified domain. If the selected user identifier value has a domain, it will extract the username to append the selected verified domain. For example, if you select the email (joe_smith@contoso.com) as the user identifier value and select contoso.onmicrosoft.com as the verified domain, this will result in joe_smith@contoso.onmicrosoft.com. |
| ToLower() | Converts the characters of the selected attribute into lowercase characters. |
| ToUpper() | Converts the characters of the selected attribute into uppercase characters. |
| Contains() | Outputs an attribute or constant if the input matches the specified value. Otherwise, you can specify another output if there's no match.
For example, if you want to emit a claim where the value is the user's email address if it contains the domain "@contoso.com", otherwise you want to output the user principal name. To do this, you would configure the following values:
<i>Parameter 1 (input): user.email
Value: "@contoso.com"</i>
<i>Parameter 2 (output): user.email</i>
<i>Parameter 3 (output if there's no match): user.userprincipalname</i> |

| FUNCTION | DESCRIPTION |
|-------------------------------------|---|
| EndWith() | <p>Outputs an attribute or constant if the input ends with the specified value. Otherwise, you can specify another output if there's no match.</p> <p>For example, if you want to emit a claim where the value is the user's employee ID if the employee ID ends with "000", otherwise you want to output an extension attribute. To do this, you would configure the following values:</p> <p><i>Parameter 1(input):</i> user.employeeid
 <i>Value:</i> "000"</p> <p><i>Parameter 2 (output):</i> user.employeeid
 <i>Parameter 3 (output if there's no match):</i> user.extensionattribute1</p> |
| StartWith() | <p>Outputs an attribute or constant if the input starts with the specified value. Otherwise, you can specify another output if there's no match.</p> <p>For example, if you want to emit a claim where the value is the user's employee ID if the country/region starts with "US", otherwise you want to output an extension attribute. To do this, you would configure the following values:</p> <p><i>Parameter 1(input):</i> user.country
 <i>Value:</i> "US"</p> <p><i>Parameter 2 (output):</i> user.employeeid
 <i>Parameter 3 (output if there's no match):</i> user.extensionattribute1</p> |
| Extract() - After matching | <p>Returns the substring after it matches the specified value.</p> <p>For example, if the input's value is "Finance_BSimon", the matching value is "Finance_", then the claim's output is "BSimon".</p> |
| Extract() - Before matching | <p>Returns the substring until it matches the specified value.</p> <p>For example, if the input's value is "BSimon_US", the matching value is "_US", then the claim's output is "BSimon".</p> |
| Extract() - Between matching | <p>Returns the substring until it matches the specified value.</p> <p>For example, if the input's value is "Finance_BSimon_US", the first matching value is "Finance_", the second matching value is "_US", then the claim's output is "BSimon".</p> |
| ExtractAlpha() - Prefix | <p>Returns the prefix alphabetical part of the string.</p> <p>For example, if the input's value is "BSimon_123", then it returns "BSimon".</p> |
| ExtractAlpha() - Suffix | <p>Returns the suffix alphabetical part of the string.</p> <p>For example, if the input's value is "123_Simon", then it returns "Simon".</p> |
| ExtractNumeric() - Prefix | <p>Returns the prefix numerical part of the string.</p> <p>For example, if the input's value is "123_BSimon", then it returns "123".</p> |
| ExtractNumeric() - Suffix | <p>Returns the suffix numerical part of the string.</p> <p>For example, if the input's value is "BSimon_123", then it returns "123".</p> |

| FUNCTION | DESCRIPTION |
|---------------------|---|
| IfEmpty() | <p>Outputs an attribute or constant if the input is null or empty. For example, if you want to output an attribute stored in an extensionattribute if the employee ID for a given user is empty. To do this, you would configure the following values:</p> <ul style="list-style-type: none"> Parameter 1(input): user.employeeid Parameter 2 (output): user.extensionattribute1 Parameter 3 (output if there's no match): user.employeeid |
| IfNotEmpty() | <p>Outputs an attribute or constant if the input is not null or empty. For example, if you want to output an attribute stored in an extensionattribute if the employee ID for a given user is not empty. To do this, you would configure the following values:</p> <ul style="list-style-type: none"> Parameter 1(input): user.employeeid Parameter 2 (output): user.extensionattribute1 |

If you need additional transformations, submit your idea in the [feedback forum in Azure AD](#) under the *SaaS application* category.

Emitting claims based on conditions

You can specify the source of a claim based on user type and the group to which the user belongs.

The user type can be:

- **Any:** All users are allowed to access the application.
- **Members:** Native member of the tenant
- **All guests:** User is brought over from an external organization with or without Azure AD.
- **AAD guests:** Guest user belongs to another organization using Azure AD.
- **External guests:** Guest user belongs to an external organization that doesn't have Azure AD.

One scenario where this is helpful is when the source of a claim is different for a guest and an employee accessing an application. You may want to specify that if the user is an employee the NameID is sourced from user.email, but if the user is a guest then the NameID is sourced from user.extensionattribute1.

To add a claim condition:

1. In **Manage claim**, expand the Claim conditions.
2. Select the user type.
3. Select the group(s) to which the user should belong. You can select up to 10 unique groups across all claims for a given application.
4. Select the **Source** where the claim is going to retrieve its value. You can select a user attribute from the source attribute dropdown or apply a transformation to the user attribute before emitting it as a claim.

The order in which you add the conditions are important. Azure AD evaluates the conditions from top to bottom to decide which value to emit in the claim.

For example, Brita Simon is a guest user in the Contoso tenant. She belongs to another organization that also uses Azure AD. Given the below configuration for the Fabrikam application, when Brita tries to sign in to Fabrikam, Azure AD will evaluate the conditions as follow.

First, Azure AD verifies if Brita's user type is `All guests`. Since, this is true then Azure AD assigns the source for the claim to `user.extensionattribute1`. Second, Azure AD verifies if Brita's user type is `AAD guests`, since this is also true then Azure AD assigns the source for the claim to `user.mail`. Finally, the claim is emitted with value `user.email` for Brita.

Manage claim

 Save  Discard changes

* Name ✓

Namespace

* Source Attribute Transformation

* Source attribute

Claim Conditions

Returns the claim only if all the conditions below are met.



Multiple conditions can be applied to a claim. When adding conditions, order of operation is important. [Read the documentation](#) for more information.

| USER TYPE | SCOPED GROUPS | SOURCE | VALUE | ... |
|-----------------------|---------------|--|--------------------------|-----|
| All Guests | 0 groups | Attribute | user.extensionattribute1 | ... |
| AAD Guests | 0 groups | Attribute | user.mail | ... |
| Select from drop down | Select Groups | <input type="radio"/> Attribute <input type="radio"/> Transformation | | |

Next steps

- [Application management in Azure AD](#)
- [Configure single sign-on on applications that are not in the Azure AD application gallery](#)
- [Troubleshoot SAML-based single sign-on](#)

How to: Customize claims emitted in tokens for a specific app in a tenant (Preview)

10/23/2019 • 13 minutes to read • [Edit Online](#)

NOTE

This feature replaces and supersedes the [claims customization](#) offered through the portal today. On the same application, if you customize claims using the portal in addition to the Graph/PowerShell method detailed in this document, tokens issued for that application will ignore the configuration in the portal. Configurations made through the methods detailed in this document will not be reflected in the portal.

This feature is used by tenant admins to customize the claims emitted in tokens for a specific application in their tenant. You can use claims-mapping policies to:

- Select which claims are included in tokens.
- Create claim types that do not already exist.
- Choose or change the source of data emitted in specific claims.

NOTE

This capability currently is in public preview. Be prepared to revert or remove any changes. The feature is available in any Azure Active Directory (Azure AD) subscription during public preview. However, when the feature becomes generally available, some aspects of the feature might require an Azure AD premium subscription. This feature supports configuring claim mapping policies for WS-Fed, SAML, OAuth, and OpenID Connect protocols.

Claims mapping policy type

In Azure AD, a **Policy** object represents a set of rules enforced on individual applications or on all applications in an organization. Each type of policy has a unique structure, with a set of properties that are then applied to objects to which they are assigned.

A claims mapping policy is a type of **Policy** object that modifies the claims emitted in tokens issued for specific applications.

Claim sets

There are certain sets of claims that define how and when they're used in tokens.

| CLAIM SET | DESCRIPTION |
|-----------------|---|
| Core claim set | Are present in every token regardless of the policy. These claims are also considered restricted, and can't be modified. |
| Basic claim set | Includes the claims that are emitted by default for tokens (in addition to the core claim set). You can omit or modify basic claims by using the claims mapping policies. |

| CLAIM SET | DESCRIPTION |
|----------------------|---|
| Restricted claim set | Can't be modified using policy. The data source cannot be changed, and no transformation is applied when generating these claims. |

Table 1: JSON Web Token (JWT) restricted claim set

| CLAIM TYPE (NAME) |
|-------------------|
| _claim_names |
| _claim_sources |
| access_token |
| account_type |
| acr |
| actor |
| actortoken |
| aio |
| altsecid |
| amr |
| app_chain |
| app_displayname |
| app_res |
| appctx |
| appctxsender |
| appid |
| appidacr |
| assertion |
| at_hash |
| aud |
| auth_data |
| auth_time |

CLAIM TYPE (NAME)

authorization_code

azp

azpacr

c_hash

ca_enf

cc

cert_token_use

client_id

cloud_graph_host_name

cloud_instance_name

cnf

code

controls

credential_keys

csr

csr_type

deviceid

dns_names

domain_dns_name

domain_netbios_name

e_exp

email

endpoint

enfpolids

exp

CLAIM TYPE (NAME)

expires_on

grant_type

graph

group_sids

groups

hasgroups

hash_alg

home_oid

`http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationinstant``http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationmethod``http://schemas.microsoft.com/ws/2008/06/identity/claims/expiration``http://schemas.microsoft.com/ws/2008/06/identity/claims/expired``http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress``http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name``http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier`

iat

identityprovider

idp

in_corp

instance

ipaddr

isbrowserhostedapp

iss

jwk

key_id

CLAIM TYPE (NAME)

key_type

mam_compliance_url

mam_enrollment_url

mam_terms_of_use_url

mdm_compliance_url

mdm_enrollment_url

mdm_terms_of_use_url

nameid

nbf

netbios_name

nonce

oid

on_prem_id

onprem_sam_account_name

onprem_sid

openid2_id

password

platf

polids

pop_jwk

preferred_username

previous_refresh_token

primary_sid

puid

pwd_exp

CLAIM TYPE (NAME)

pwd_url

redirect_uri

refresh_token

refreshtoken

request_nonce

resource

role

roles

scope

scp

sid

signature

signin_state

src1

src2

sub

tbid

tenant_display_name

tenant_region_scope

thumbnail_photo

tid

tokenAutologonEnabled

trustedfordelegation

unique_name

upn

| CLAIM TYPE (NAME) |
|--------------------------|
| user_setting_sync_url |
| username |
| uti |
| ver |
| verified_primary_email |
| verified_secondary_email |
| wids |
| win_ver |

Table 2: SAML restricted claim set

| CLAIM TYPE (URI) |
|--|
| <code>http://schemas.microsoft.com/ws/2008/06/identity/claims/expiration</code> |
| <code>http://schemas.microsoft.com/ws/2008/06/identity/claims/expired</code> |
| <code>http://schemas.microsoft.com/identity/claims/accesstoken</code> |
| <code>http://schemas.microsoft.com/identity/claims/openid2_id</code> |
| <code>http://schemas.microsoft.com/identity/claims/identityprovider</code> |
| <code>http://schemas.microsoft.com/identity/claims/objectidentifier</code> |
| <code>http://schemas.microsoft.com/identity/claims/puid</code> |
| <code>http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier [MR1]</code> |
| <code>http://schemas.microsoft.com/identity/claims/tenantid</code> |
| <code>http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationinstant</code> |
| <code>http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationmethod</code> |
| <code>http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider</code> |
| <code>http://schemas.microsoft.com/ws/2008/06/identity/claims/groups</code> |
| <code>http://schemas.microsoft.com/claims/groups.link</code> |
| <code>http://schemas.microsoft.com/ws/2008/06/identity/claims/role</code> |

CLAIM TYPE (URI)

`http://schemas.microsoft.com/ws/2008/06/identity/claims/wids`

`http://schemas.microsoft.com/2014/09/devicecontext/claims/iscompliant`

`http://schemas.microsoft.com/2014/02/devicecontext/claims/isknown`

`http://schemas.microsoft.com/2012/01/devicecontext/claims/ismanaged`

`http://schemas.microsoft.com/2014/03/psto`

`http://schemas.microsoft.com/claims/authnmethodsreferences`

`http://schemas.xmlsoap.org/ws/2009/09/identity/claims/actor`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/samlissuername`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/confirmationkey`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsaccountname`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/primarygroupsid`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/authorizationdecision`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/authentication`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/sid`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/denyonlyprimarygroupsid`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/denyonlyprimarysid`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/denyonlysid`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/denyonlywindowsdevicegroup`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsdeviceclaim`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsdevicegroup`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsfqbnversion`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowssubauthority`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsuserclaim`

CLAIM TYPE (URI)

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/x500distinguishedname`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/spn`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/ispersistent`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/privatepersonalidentifier`

`http://schemas.microsoft.com/identity/claims/scope`

Claims mapping policy properties

To control what claims are emitted and where the data comes from, use the properties of a claims mapping policy. If a policy is not set, the system issues tokens that include the core claim set, the basic claim set, and any [optional claims](#) that the application has chosen to receive.

Include basic claim set

String: `IncludeBasicClaimSet`

Data type: Boolean (True or False)

Summary: This property determines whether the basic claim set is included in tokens affected by this policy.

- If set to True, all claims in the basic claim set are emitted in tokens affected by the policy.
- If set to False, claims in the basic claim set are not in the tokens, unless they are individually added in the claims schema property of the same policy.

NOTE

Claims in the core claim set are present in every token, regardless of what this property is set to.

Claims schema

String: `ClaimsSchema`

Data type: JSON blob with one or more claim schema entries

Summary: This property defines which claims are present in the tokens affected by the policy, in addition to the basic claim set and the core claim set. For each claim schema entry defined in this property, certain information is required. Specify where the data is coming from (**Value** or **Source/ID pair**), and which claim the data is emitted as (**Claim Type**).

Claim schema entry elements

Value: The Value element defines a static value as the data to be emitted in the claim.

Source/ID pair: The Source and ID elements define where the data in the claim is sourced from.

Set the Source element to one of the following values:

- "user": The data in the claim is a property on the User object.

- "application": The data in the claim is a property on the application (client) service principal.
- "resource": The data in the claim is a property on the resource service principal.
- "audience": The data in the claim is a property on the service principal that is the audience of the token (either the client or resource service principal).
- "company": The data in the claim is a property on the resource tenant's Company object.
- "transformation": The data in the claim is from claims transformation (see the "Claims transformation" section later in this article).

If the source is transformation, the **TransformationID** element must be included in this claim definition as well.

The ID element identifies which property on the source provides the value for the claim. The following table lists the values of ID valid for each value of Source.

Table 3: Valid ID values per source

| SOURCE | ID | DESCRIPTION |
|--------|------------------------------|---------------------------------|
| User | surname | Family Name |
| User | givenname | Given Name |
| User | displayname | Display Name |
| User | objectid | ObjectID |
| User | mail | Email Address |
| User | userprincipalname | User Principal Name |
| User | department | Department |
| User | onpremisesamaccountname | On-premises SAM Account Name |
| User | netbiosname | NetBios Name |
| User | dnsdomainname | DNS Domain Name |
| User | onpremisessecurityidentifier | On-premises Security Identifier |
| User | companynamne | Organization Name |
| User | streetaddress | Street Address |
| User | postalcode | Postal Code |
| User | preferredlanguage | Preferred Language |
| User | onpremisesuserprincipalname | On-premises UPN |
| User | mailnickname | Mail Nickname |
| User | extensionattribute1 | Extension Attribute 1 |

| SOURCE | ID | DESCRIPTION |
|---------------------------------|--------------------------|----------------------------|
| User | extensionattribute2 | Extension Attribute 2 |
| User | extensionattribute3 | Extension Attribute 3 |
| User | extensionattribute4 | Extension Attribute 4 |
| User | extensionattribute5 | Extension Attribute 5 |
| User | extensionattribute6 | Extension Attribute 6 |
| User | extensionattribute7 | Extension Attribute 7 |
| User | extensionattribute8 | Extension Attribute 8 |
| User | extensionattribute9 | Extension Attribute 9 |
| User | extensionattribute10 | Extension Attribute 10 |
| User | extensionattribute11 | Extension Attribute 11 |
| User | extensionattribute12 | Extension Attribute 12 |
| User | extensionattribute13 | Extension Attribute 13 |
| User | extensionattribute14 | Extension Attribute 14 |
| User | extensionattribute15 | Extension Attribute 15 |
| User | othermail | Other Mail |
| User | country | Country |
| User | city | City |
| User | state | State |
| User | jobtitle | Job Title |
| User | employeeid | Employee ID |
| User | facsimiletelephonenumber | Facsimile Telephone Number |
| application, resource, audience | displayname | Display Name |
| application, resource, audience | objected | ObjectID |
| application, resource, audience | tags | Service Principal Tag |
| Company | tenantcountry | Tenant's country |

TransformationID: The TransformationID element must be provided only if the Source element is set to

"transformation".

- This element must match the ID element of the transformation entry in the **ClaimsTransformation** property that defines how the data for this claim is generated.

Claim Type: The **JwtClaimType** and **SamlClaimType** elements define which claim this claim schema entry refers to.

- The JwtClaimType must contain the name of the claim to be emitted in JWTs.
- The SamlClaimType must contain the URI of the claim to be emitted in SAML tokens.

NOTE

Names and URIs of claims in the restricted claim set cannot be used for the claim type elements. For more information, see the "Exceptions and restrictions" section later in this article.

Claims transformation

String: ClaimsTransformation

Data type: JSON blob, with one or more transformation entries

Summary: Use this property to apply common transformations to source data, to generate the output data for claims specified in the Claims Schema.

ID: Use the ID element to reference this transformation entry in the TransformationID Claims Schema entry. This value must be unique for each transformation entry within this policy.

TransformationMethod: The TransformationMethod element identifies which operation is performed to generate the data for the claim.

Based on the method chosen, a set of inputs and outputs is expected. Define the inputs and outputs by using the **InputClaims**, **InputParameters** and **OutputClaims** elements.

Table 4: Transformation methods and expected inputs and outputs

| TRANSFORMATIONMETHOD | EXPECTED INPUT | EXPECTED OUTPUT | DESCRIPTION |
|----------------------|-----------------------------|-----------------|---|
| Join | string1, string2, separator | outputClaim | Joins input strings by using a separator in between. For example:
string1:"foo@bar.com",
string2:"sandbox",
separator:"." results in
outputClaim:"foo@bar.com.s
andbox" |
| ExtractMailPrefix | mail | outputClaim | Extracts the local part of an email address. For example:
mail:"foo@bar.com" results in
outputClaim:"foo". If no @ sign is present, then the original input string is returned as is. |

InputClaims: Use an InputClaims element to pass the data from a claim schema entry to a transformation. It has two attributes: **ClaimTypeReferenceId** and **TransformationClaimType**.

- **ClaimTypeReferenceId** is joined with ID element of the claim schema entry to find the appropriate input claim.

- **TransformationClaimType** is used to give a unique name to this input. This name must match one of the expected inputs for the transformation method.

InputParameters: Use an InputParameters element to pass a constant value to a transformation. It has two attributes: **Value** and **ID**.

- **Value** is the actual constant value to be passed.
- **ID** is used to give a unique name to the input. The name must match one of the expected inputs for the transformation method.

OutputClaims: Use an OutputClaims element to hold the data generated by a transformation, and tie it to a claim schema entry. It has two attributes: **ClaimTypeReferenceId** and **TransformationClaimType**.

- **ClaimTypeReferenceId** is joined with the ID of the claim schema entry to find the appropriate output claim.
- **TransformationClaimType** is used to give a unique name to the output. The name must match one of the expected outputs for the transformation method.

Exceptions and restrictions

SAML NameID and UPN: The attributes from which you source the NameID and UPN values, and the claims transformations that are permitted, are limited. See table 5 and table 6 to see the permitted values.

Table 5: Attributes allowed as a data source for SAML NameID

| SOURCE | ID | DESCRIPTION |
|--------|-------------------------|------------------------------|
| User | mail | Email Address |
| User | userprincipalname | User Principal Name |
| User | onpremisesamaccountname | On Premises Sam Account Name |
| User | employeeid | Employee ID |
| User | extensionattribute1 | Extension Attribute 1 |
| User | extensionattribute2 | Extension Attribute 2 |
| User | extensionattribute3 | Extension Attribute 3 |
| User | extensionattribute4 | Extension Attribute 4 |
| User | extensionattribute5 | Extension Attribute 5 |
| User | extensionattribute6 | Extension Attribute 6 |
| User | extensionattribute7 | Extension Attribute 7 |
| User | extensionattribute8 | Extension Attribute 8 |
| User | extensionattribute9 | Extension Attribute 9 |
| User | extensionattribute10 | Extension Attribute 10 |
| User | extensionattribute11 | Extension Attribute 11 |

| SOURCE | ID | DESCRIPTION |
|--------|----------------------|------------------------|
| User | extensionattribute12 | Extension Attribute 12 |
| User | extensionattribute13 | Extension Attribute 13 |
| User | extensionattribute14 | Extension Attribute 14 |
| User | extensionattribute15 | Extension Attribute 15 |

Table 6: Transformation methods allowed for SAML NameID

| TRANSFORMATIONMETHOD | RESTRICTIONS |
|----------------------|---|
| ExtractMailPrefix | None |
| Join | The suffix being joined must be a verified domain of the resource tenant. |

Custom signing key

A custom signing key must be assigned to the service principal object for a claims mapping policy to take effect. This ensures acknowledgment that tokens have been modified by the creator of the claims mapping policy and protects applications from claims mapping policies created by malicious actors. Apps that have claims mapping enabled must check a special URI for their token signing keys by appending `appid={client_id}` to their [OpenID Connect metadata requests](#).

Cross-tenant scenarios

Claims mapping policies do not apply to guest users. If a guest user tries to access an application with a claims mapping policy assigned to its service principal, the default token is issued (the policy has no effect).

Claims mapping policy assignment

Claims mapping policies can only be assigned to service principal objects.

Example claims mapping policies

In Azure AD, many scenarios are possible when you can customize claims emitted in tokens for specific service principals. In this section, we walk through a few common scenarios that can help you grasp how to use the claims mapping policy type.

Prerequisites

In the following examples, you create, update, link, and delete policies for service principals. If you are new to Azure AD, we recommend that you [learn about how to get an Azure AD tenant](#) before you proceed with these examples.

To get started, do the following steps:

1. Download the latest [Azure AD PowerShell Module public preview release](#).
2. Run the Connect command to sign in to your Azure AD admin account. Run this command each time you start a new session.

```
Connect-AzureAD -Confirm
```

3. To see all policies that have been created in your organization, run the following command. We recommend that you run this command after most operations in the following scenarios, to check that your policies are being created as expected.

```
Get-AzureADPolicy
```

Example: Create and assign a policy to omit the basic claims from tokens issued to a service principal

In this example, you create a policy that removes the basic claim set from tokens issued to linked service principals.

1. Create a claims mapping policy. This policy, linked to specific service principals, removes the basic claim set from tokens.

- a. To create the policy, run this command:

```
New-AzureADPolicy -Definition @('{"ClaimsMappingPolicy": {"Version":1,"IncludeBasicClaimSet":"false"}}') -DisplayName "OmitBasicClaims" -Type "ClaimsMappingPolicy"
```

- b. To see your new policy, and to get the policy ObjectId, run the following command:

```
Get-AzureADPolicy
```

2. Assign the policy to your service principal. You also need to get the ObjectId of your service principal.

- a. To see all your organization's service principals, you can [query Microsoft Graph](#). Or, in [Graph Explorer](#), sign in to your Azure AD account.

- b. When you have the ObjectId of your service principal, run the following command:

```
Add-AzureADServicePrincipalPolicy -Id <ObjectId of the ServicePrincipal> -RefObjectId <ObjectId of the Policy>
```

Example: Create and assign a policy to include the EmployeeID and TenantCountry as claims in tokens issued to a service principal

In this example, you create a policy that adds the EmployeeID and TenantCountry to tokens issued to linked service principals. The EmployeeID is emitted as the name claim type in both SAML tokens and JWTs. The TenantCountry is emitted as the country claim type in both SAML tokens and JWTs. In this example, we continue to include the basic claims set in the tokens.

1. Create a claims mapping policy. This policy, linked to specific service principals, adds the EmployeeID and TenantCountry claims to tokens.

- a. To create the policy, run the following command:

```
New-AzureADPolicy -Definition @('{"ClaimsMappingPolicy": {"Version":1,"IncludeBasicClaimSet":"true", "ClaimsSchema": [{"Source":"user","ID":"employeeid","SamlClaimType":"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name","JwtClaimType":"name"}, {"Source":"company","ID":"tenantcountry","SamlClaimType":"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/country","JwtClaimType":"country"}]}') -DisplayName "ExtraClaimsExample" -Type "ClaimsMappingPolicy"
```

- b. To see your new policy, and to get the policy ObjectId, run the following command:

```
Get-AzureADPolicy
```

2. Assign the policy to your service principal. You also need to get the ObjectId of your service principal.

- a. To see all your organization's service principals, you can [query Microsoft Graph](#). Or, in [Graph Explorer](#), sign in to your Azure AD account.

- b. When you have the ObjectId of your service principal, run the following command:

```
Add-AzureADServicePrincipalPolicy -Id <ObjectId of the ServicePrincipal> -RefObjectId <ObjectId of the Policy>
```

Example: Create and assign a policy that uses a claims transformation in tokens issued to a service principal

In this example, you create a policy that emits a custom claim "JoinedData" to JWTs issued to linked service principals. This claim contains a value created by joining the data stored in the extensionattribute1 attribute on the user object with ".sandbox". In this example, we exclude the basic claims set in the tokens.

1. Create a claims mapping policy. This policy, linked to specific service principals, adds the EmployeeID and TenantCountry claims to tokens.

- a. To create the policy, run the following command:

```
New-AzureADPolicy -Definition @('{"ClaimsMappingPolicy": {"Version":1,"IncludeBasicClaimSet":"true", "ClaimsSchema": [{"Source":"user","ID":"extensionattribute1"}, {"Source":"transformation","ID":"DataJoin","TransformationId":"JoinTheData","JwtClaimType":"Joined Data"}], "ClaimsTransformations":[{"ID":"JoinTheData","TransformationMethod":"Join","InputClaims":[{"ClaimTypeReferenceId":"extensionattribute1","TransformationClaimType":"string1"}], "InputParameters": [{"ID":"string2","Value":"sandbox"}, {"ID":"separator","Value":(".")}], "OutputClaims": [{"ClaimTypeReferenceId":"DataJoin","TransformationClaimType":"outputClaim"}]}]}) -DisplayName "TransformClaimsExample" -Type "ClaimsMappingPolicy"
```

- b. To see your new policy, and to get the policy ObjectId, run the following command:

```
Get-AzureADPolicy
```

2. Assign the policy to your service principal. You also need to get the ObjectId of your service principal.

- a. To see all your organization's service principals, you can [query Microsoft Graph](#). Or, in [Graph Explorer](#), sign in to your Azure AD account.

- b. When you have the ObjectId of your service principal, run the following command:

```
Add-AzureADServicePrincipalPolicy -Id <ObjectId of the ServicePrincipal> -RefObjectId <ObjectId of the Policy>
```

See also

To learn how to customize claims issued in the SAML token through the Azure portal, see [How to: Customize claims issued in the SAML token for enterprise applications](#)

How to: Provide optional claims to your Azure AD app

10/23/2019 • 12 minutes to read • [Edit Online](#)

Application developers can use optional claims in their Azure AD apps to specify which claims they want in tokens sent to their application.

You can use optional claims to:

- Select additional claims to include in tokens for your application.
- Change the behavior of certain claims that Azure AD returns in tokens.
- Add and access custom claims for your application.

For the lists of standard claims, see the [access token](#) and [id_token](#) claims documentation.

While optional claims are supported in both v1.0 and v2.0 format tokens, as well as SAML tokens, they provide most of their value when moving from v1.0 to v2.0. One of the goals of the [v2.0 Microsoft identity platform endpoint](#) is smaller token sizes to ensure optimal performance by clients. As a result, several claims formerly included in the access and ID tokens are no longer present in v2.0 tokens and must be asked for specifically on a per-application basis.

Table 1: Applicability

| ACCOUNT TYPE | V1.0 TOKENS | V2.0 TOKENS |
|----------------------------|-------------|-------------|
| Personal Microsoft account | N/A | Supported |
| Azure AD account | Supported | Supported |

v1.0 and v2.0 optional claims set

The set of optional claims available by default for applications to use are listed below. To add custom optional claims for your application, see [Directory Extensions](#), below. When adding claims to the **access token**, this will apply to access tokens requested *for* the application (a web API), not those *by* the application. This ensures that no matter the client accessing your API, the right data is present in the access token they use to authenticate against your API.

NOTE

The majority of these claims can be included in JWTs for v1.0 and v2.0 tokens, but not SAML tokens, except where noted in the Token Type column. Consumer accounts support a subset of these claims, marked in the "User Type" column. Many of the claims listed do not apply to consumer users (they have no tenant, so `tenant_ctry` has no value).

Table 2: v1.0 and v2.0 optional claim set

| NAME | DESCRIPTION | TOKEN TYPE | USER TYPE | NOTES |
|------------------------|---|------------|-----------|-------|
| <code>auth_time</code> | Time when the user last authenticated. See OpenID Connect spec. | JWT | | |

| NAME | DESCRIPTION | TOKEN TYPE | USER TYPE | NOTES |
|---------------------------------------|---|------------|---------------------------------|--|
| <code>tenant_region_scope</code> | Region of the resource tenant | JWT | | |
| <code>home_oid</code> | For guest users, the object ID of the user in the user's home tenant. | JWT | | |
| <code>sid</code> | Session ID, used for per-session user sign out. | JWT | Personal and Azure AD accounts. | |
| <code>platf</code> | Device platform | JWT | | Restricted to managed devices that can verify device type. |
| <code>verified_primary_email</code> | Sourced from the user's PrimaryAuthoritativeEmail | JWT | | |
| <code>verified_secondary_email</code> | Sourced from the user's SecondaryAuthoritativeEmail | JWT | | |
| <code>enfpolids</code> | Enforced policy IDs. A list of the policy IDs that were evaluated for the current user. | JWT | | |
| <code>vnet</code> | VNET specifier information. | JWT | | |
| <code>fwd</code> | IP address. | JWT | | Adds the original IPv4 address of the requesting client (when inside a VNET) |
| <code>ctry</code> | User's country | JWT | | Azure AD returns the <code>ctry</code> optional claim if it's present and the value of the claim is a standard two-letter country code, such as FR, JP, SZ, and so on. |
| <code>tenant_ctry</code> | Resource tenant's country | JWT | | |

| NAME | DESCRIPTION | TOKEN TYPE | USER TYPE | NOTES |
|---------|---|------------|---------------|--|
| xms_pd1 | Preferred data location | JWT | | <p>For Multi-Geo tenants, this is the 3-letter code showing the geographic region the user is in. For more info, see the Azure AD Connect documentation about preferred data location.</p> <p>For example: <code>APC</code> for Asia Pacific.</p> |
| xms_pl | User preferred language | JWT | | <p>The user's preferred language, if set. Sourced from their home tenant, in guest access scenarios. Formatted LL-CC ("en-us").</p> |
| xms_tpl | Tenant preferred language | JWT | | <p>The resource tenant's preferred language, if set. Formatted LL ("en").</p> |
| ztdid | Zero-touch Deployment ID | JWT | | <p>The device identity used for Windows AutoPilot</p> |
| email | The addressable email for this user, if the user has one. | JWT, SAML | MSA, Azure AD | <p>This value is included by default if the user is a guest in the tenant. For managed users (those inside the tenant), it must be requested through this optional claim or, on v2.0 only, with the OpenID scope. For managed users, the email address must be set in the Office admin portal.</p> |
| groups | Optional formatting for group claims | JWT, SAML | | <p>Used in conjunction with the <code>GroupMembershipClaims</code> setting in the application manifest, which must be set as well. For details see [Group claims] (#Configuring-group-optional claims) below. For more information on group claims see How to configure group claims</p> |

| NAME | DESCRIPTION | TOKEN TYPE | USER TYPE | NOTES |
|------|---------------------------------|------------|-----------|---|
| acct | Users account status in tenant. | JWT, SAML | | If the user is a member of the tenant, the value is <code>0</code> . If they are a guest, the value is <code>1</code> . |
| upn | UserPrincipalName claim. | JWT, SAML | | Although this claim is automatically included, you can specify it as an optional claim to attach additional properties to modify its behavior in the guest user case. |

v2.0 optional claims

These claims are always included in v1.0 Azure AD tokens, but not included in v2.0 tokens unless requested. These claims are only applicable for JWTs (ID tokens and Access Tokens).

Table 3: v2.0-only optional claims

| JWT CLAIM | NAME | DESCRIPTION | NOTES |
|-------------|---------------------------------|---|---|
| ipaddr | IP Address | The IP address the client logged in from. | |
| onprem_sid | On-Premises Security Identifier | | |
| pwd_exp | Password Expiration Time | The datetime at which the password expires. | |
| pwd_url | Change Password URL | A URL that the user can visit to change their password. | |
| in_corp | Inside Corporate Network | Signals if the client is logging in from the corporate network. If they're not, the claim isn't included. | Based off of the trusted IPs settings in MFA. |
| nickname | Nickname | An additional name for the user, separate from first or last name. | |
| family_name | Last Name | Provides the last name, surname, or family name of the user as defined in the user object.
"family_name": "Miller" | Supported in MSA and Azure AD |
| given_name | First name | Provides the first or "given" name of the user, as set on the user object.
"given_name": "Frank" | Supported in MSA and Azure AD |

| JWT CLAIM | NAME | DESCRIPTION | NOTES |
|-----------|---------------------|---|---|
| upn | User Principal Name | An identifier for the user that can be used with the username_hint parameter. Not a durable identifier for the user and should not be used to key data. | See additional properties below for configuration of the claim. |

Additional properties of optional claims

Some optional claims can be configured to change the way the claim is returned. These additional properties are mostly used to help migration of on-premises applications with different data expectations (for example,

`include_externally_authenticated_upn_without_hash` helps with clients that cannot handle hash marks (#) in the UPN)

Table 4: Values for configuring optional claims

| PROPERTY NAME | ADDITIONAL PROPERTY NAME | DESCRIPTION |
|---------------|---|---|
| upn | | Can be used for both SAML and JWT responses, and for v1.0 and v2.0 tokens. |
| | <code>include_externally_authenticated_upn</code> | Includes the guest UPN as stored in the resource tenant. For example,
<code>foo_hometenant.com#EXT#@resourcetenant.com</code> |
| | <code>include_externally_authenticated_upn_with_hash</code> | Same as above, except that the hash marks (#) are replaced with underscores (_), for example
<code>foo_hometenant.com_EXT_@resourcetenant.com</code> |

Additional properties example

```
"optionalClaims":  
{  
    "idToken": [  
        {  
            "name": "upn",  
            "essential": false,  
            "additionalProperties": [ "include_externally_authenticated_upn"]  
        }  
    ]  
}
```

This OptionalClaims object causes the ID token returned to the client to include another upn with the additional home tenant and resource tenant information. This will only change the `upn` claim in the token if the user is a guest in the tenant (that uses a different IDP for authentication).

Configuring optional claims

You can configure optional claims for your application by modifying the application manifest (See example below). For more info, see the [Understanding the Azure AD application manifest article](#).

IMPORTANT

Access tokens are **always** generated using the manifest of the resource, not the client. So in the request

`...scope=https://graph.microsoft.com/user.read...` the resource is Graph. Thus, the access token is created using the Graph manifest, not the client's manifest. Changing the manifest for your application will never cause tokens for Graph to look different. In order to validate that your `accessToken` changes are in effect, request a token for your application, not another app.

Sample schema:

```
"optionalClaims":  
{  
    "idToken": [  
        {  
            "name": "auth_time",  
            "essential": false  
        }  
    ],  
    "accessToken": [  
        {  
            "name": "ipaddr",  
            "essential": false  
        }  
    ],  
    "saml2Token": [  
        {  
            "name": "upn",  
            "essential": false  
        },  
        {  
            "name": "extension_ab603c56068041afb2f6832e2a17e237_skypeId",  
            "source": "user",  
            "essential": false  
        }  
    ]  
}
```

OptionalClaims type

Declares the optional claims requested by an application. An application can configure optional claims to be returned in each of three types of tokens (ID token, access token, SAML 2 token) it can receive from the security token service. The application can configure a different set of optional claims to be returned in each token type. The OptionalClaims property of the Application entity is an OptionalClaims object.

Table 5: OptionalClaims type properties

| NAME | TYPE | DESCRIPTION |
|--------------------------|----------------------------|---|
| <code>idToken</code> | Collection (OptionalClaim) | The optional claims returned in the JWT ID token. |
| <code>accessToken</code> | Collection (OptionalClaim) | The optional claims returned in the JWT access token. |
| <code>saml2Token</code> | Collection (OptionalClaim) | The optional claims returned in the SAML token. |

OptionalClaim type

Contains an optional claim associated with an application or a service principal. The idToken, accessToken, and

saml2Token properties of the [OptionalClaims](#) type is a collection of [OptionalClaim](#). If supported by a specific claim, you can also modify the behavior of the [OptionalClaim](#) using the [AdditionalProperties](#) field.

Table 6: OptionalClaim type properties

| NAME | TYPE | DESCRIPTION |
|-----------------------------------|-------------------------|---|
| <code>name</code> | Edm.String | The name of the optional claim. |
| <code>source</code> | Edm.String | The source (directory object) of the claim. There are predefined claims and user-defined claims from extension properties. If the source value is null, the claim is a predefined optional claim. If the source value is user, the value in the name property is the extension property from the user object. |
| <code>essential</code> | Edm.Boolean | If the value is true, the claim specified by the client is necessary to ensure a smooth authorization experience for the specific task requested by the end user. The default value is false. |
| <code>additionalProperties</code> | Collection (Edm.String) | Additional properties of the claim. If a property exists in this collection, it modifies the behavior of the optional claim specified in the name property. |

Configuring directory extension optional claims

In addition to the standard optional claims set, you can also configure tokens to include directory schema extensions. For more info, see [Directory schema extensions](#). This feature is useful for attaching additional user information that your app can use – for example, an additional identifier or important configuration option that the user has set.

NOTE

- Directory schema extensions are an Azure AD-only feature, so if your application manifest requests a custom extension and an MSA user logs into your app, these extensions will not be returned.
- Azure AD optional claims only work with the Azure AD extension and doesn't work with the Microsoft Graph directory extension. Both APIs require the `Directory.ReadWriteAll` permission, which can only be consented by admins.

Directory extension formatting

For extension attributes, use the full name of the extension (in the format: `extension_<appid>_<attributename>`) in the application manifest. The `<appid>` must match the ID of the application requesting the claim.

Within the JWT, these claims will be emitted with the following name format: `extn.<attributename>`.

Within the SAML tokens, these claims will be emitted with the following URI format:

`http://schemas.microsoft.com/identity/claims/extn.<attributename>`

Configuring group optional claims

NOTE

The ability to emit group names for users and groups synced from on-premises is Public Preview.

This section covers the configuration options under optional claims for changing the group attributes used in group claims from the default group objectID to attributes synced from on-premises Windows Active Directory.

IMPORTANT

See [Configure group claims for applications with Azure AD](#) for more details including important caveats for the public preview of group claims from on-premises attributes.

1. In the portal -> Azure Active Directory -> Application Registrations-> Select Application-> Manifest
2. Enable group membership claims by changing the groupMembershipClaim

The valid values are:

- "All"
- "SecurityGroup"
- "DistributionList"
- "DirectoryRole"

For example:

```
"groupMembershipClaims": "SecurityGroup"
```

By default Group ObjectIDs will be emitted in the group claim value. To modify the claim value to contain on premises group attributes, or to change the claim type to role, use OptionalClaims configuration as follows:

3. Set group name configuration optional claims.

If you want to groups in the token to contain the on premises AD group attributes in the optional claims section specify which token type optional claim should be applied to, the name of optional claim requested and any additional properties desired. Multiple token types can be listed:

- idToken for the OIDC ID token
- accessToken for the OAuth/OIDC access token
- Saml2Token for SAML tokens.

NOTE

The Saml2Token type applies to both SAML1.1 and SAML2.0 format tokens

For each relevant token type, modify the groups claim to use the OptionalClaims section in the manifest. The OptionalClaims schema is as follows:

```
{
  "name": "groups",
  "source": null,
  "essential": false,
  "additionalProperties": []
}
```

| OPTIONAL CLAIMS SCHEMA | VALUE |
|------------------------------|---|
| name: | Must be "groups" |
| source: | Not used. Omit or specify null |
| essential: | Not used. Omit or specify false |
| additionalProperties: | List of additional properties. Valid options are
"sam_account_name",
"dns_domain_and_sam_account_name",
"netbios_domain_and_sam_account_name", "emit_as_roles" |

In additionalProperties only one of "sam_account_name", "dns_domain_and_sam_account_name", "netbios_domain_and_sam_account_name" are required. If more than one is present, the first is used and any others ignored.

Some applications require group information about the user in the role claim. To change the claim type to from a group claim to a role claim, add "emit_as_roles" to additional properties. The group values will be emitted in the role claim.

NOTE

If "emit_as_roles" is used any Application Roles configured that the user is assigned will not appear in the role claim

Examples: Emit groups as group names in OAuth access tokens in dnsDomainName\sAMAccountName format

```
"optionalClaims": {
  "accessToken": [
    {
      "name": "groups",
      "additionalProperties": ["dns_domain_and_sam_account_name"]
    }
}
```

To emit group names to be returned in netbiosDomain\sAMAccountName format as the roles claim in SAML and OIDC ID Tokens:

```
"optionalClaims": {
  "saml2Token": [
    {
      "name": "groups",
      "additionalProperties": ["netbios_name_and_sam_account_name", "emit_as_roles"]
    },
    {
      "idToken": [
        {
          "name": "groups",
          "additionalProperties": ["netbios_name_and_sam_account_name", "emit_as_roles"]
        }
      ]
    }
}
```

Optional claims example

In this section, you can walk through a scenario to see how you can use the optional claims feature for your application. There are multiple options available for updating the properties on an application's identity configuration to enable and configure optional claims:

- You can modify the application manifest. The example below will use this method to do the configuration. Read the [Understanding the Azure AD application manifest document](#) first for an introduction to the manifest.
- It's also possible to write an application that uses the [Graph API](#) to update your application. The [Entity and complex type reference](#) in the Graph API reference guide can help you with configuring the optional claims.

Example: In the example below, you will modify an application's manifest to add claims to access, ID, and SAML tokens intended for the application.

1. Sign in to the [Azure portal](#).
2. After you've authenticated, choose your Azure AD tenant by selecting it from the top right corner of the page.
3. Select **App Registrations** from the left hand side.
4. Find the application you want to configure optional claims for in the list and click on it.
5. From the application page, click **Manifest** to open the inline manifest editor.
6. You can directly edit the manifest using this editor. The manifest follows the schema for the [Application entity](#), and auto-formats the manifest once saved. New elements will be added to the `optionalClaims` property.

```

"optionalClaims": [
  {
    "idToken": [
      {
        "name": "upn",
        "essential": false,
        "additionalProperties": [ "include_externally_authenticated_upn" ]
      }
    ],
    "accessToken": [
      {
        "name": "auth_time",
        "essential": false
      }
    ],
    "saml2Token": [
      {
        "name": "extension_ab603c56068041afb2f6832e2a17e237_skypeId",
        "source": "user",
        "essential": true
      }
    ]
  }
]
}

```

In this case, different optional claims were added to each type of token that the application can receive. The ID tokens will now contain the UPN for federated users in the full form (

`<upn>_<homedomain>#EXT#@<resourcedomain>`). The access tokens that other clients request for this application will now include the auth_time claim. The SAML tokens will now contain the skypeId directory schema extension (in this example, the app ID for this app is ab603c56068041afb2f6832e2a17e237). The SAML tokens will expose the Skype ID as `extension_skypeId` .

7. When you're finished updating the manifest, click **Save** to save the manifest

Next steps

Learn more about the standard claims provided by Azure AD.

- [ID tokens](#)
- [Access tokens](#)

Configurable token lifetimes in Azure Active Directory (Preview)

11/4/2019 • 22 minutes to read • [Edit Online](#)

You can specify the lifetime of a token issued by Azure Active Directory (Azure AD). You can set token lifetimes for all apps in your organization, for a multi-tenant (multi-organization) application, or for a specific service principal in your organization.

IMPORTANT

After hearing from customers during the preview, we've implemented [authentication session management capabilities](#) in Azure AD Conditional Access. You can use this new feature to configure refresh token lifetimes by setting sign in frequency. After May 1, 2020 you will not be able to use Configurable Token Lifetime policy to configure session and refresh tokens. You can still configure access token lifetimes after the deprecation.

In Azure AD, a policy object represents a set of rules that are enforced on individual applications or on all applications in an organization. Each policy type has a unique structure, with a set of properties that are applied to objects to which they are assigned.

You can designate a policy as the default policy for your organization. The policy is applied to any application in the organization, as long as it is not overridden by a policy with a higher priority. You also can assign a policy to specific applications. The order of priority varies by policy type.

NOTE

Configurable token lifetime policy is not supported for SharePoint Online. Even though you have the ability to create this policy via PowerShell, SharePoint Online will not acknowledge this policy. Refer to the [SharePoint Online blog](#) to learn more about configuring idle session timeouts.

- The default lifetime for the SharePoint Online access token is 1 hour.
- The default max inactive time of the SharePoint Online refresh token is 90 days.

Token types

You can set token lifetime policies for refresh tokens, access tokens, SAML tokens, session tokens, and ID tokens.

Access tokens

Clients use access tokens to access a protected resource. An access token can be used only for a specific combination of user, client, and resource. Access tokens cannot be revoked and are valid until their expiry. A malicious actor that has obtained an access token can use it for extent of its lifetime. Adjusting the lifetime of an access token is a trade-off between improving system performance and increasing the amount of time that the client retains access after the user's account is disabled. Improved system performance is achieved by reducing the number of times a client needs to acquire a fresh access token. The default is 1 hour - after 1 hour, the client must use the refresh token to (usually silently) acquire a new refresh token and access token.

SAML tokens

SAML tokens are used by many web based SAAS applications, and are obtained using Azure Active Directory's SAML2 protocol endpoint. They are also consumed by applications using WS-Federation. The default lifetime of the token is 1 hour. After From and applications perspective the validity period of the token is specified by the NotOnOrAfter value of the <conditions ...> element in the token. After the token validity period the client must initiate a new authentication request, which will often be satisfied without interactive sign in as a result of the Single Sign On

(SSO) Session token.

The value of NotOnOrAfter can be changed using the AccessTokenLifetime parameter in a TokenLifetimePolicy. It will be set to the lifetime configured in the policy if any, plus a clock skew factor of five minutes.

Note that the subject confirmation NotOnOrAfter specified in the element is not affected by the Token Lifetime configuration.

Refresh tokens

When a client acquires an access token to access a protected resource, the client also receives a refresh token. The refresh token is used to obtain new access/refresh token pairs when the current access token expires. A refresh token is bound to a combination of user and client. A refresh token can be [revoked at any time](#), and the token's validity is checked every time the token is used. Refresh tokens are not revoked when used to fetch new access tokens - it's best practice, however, to securely delete the old token when getting a new one.

It's important to make a distinction between confidential clients and public clients, as this impacts how long refresh tokens can be used. For more information about different types of clients, see [RFC 6749](#).

Token lifetimes with confidential client refresh tokens

Confidential clients are applications that can securely store a client password (secret). They can prove that requests are coming from the secured client application and not from a malicious actor. For example, a web app is a confidential client because it can store a client secret on the web server. It is not exposed. Because these flows are more secure, the default lifetimes of refresh tokens issued to these flows is `until-revoked`, cannot be changed by using policy, and will not be revoked on voluntary password resets.

Token lifetimes with public client refresh tokens

Public clients cannot securely store a client password (secret). For example, an iOS/Android app cannot obfuscate a secret from the resource owner, so it is considered a public client. You can set policies on resources to prevent refresh tokens from public clients older than a specified period from obtaining a new access/refresh token pair. (To do this, use the Refresh Token Max Inactive Time property (`MaxInactiveTime`.) You also can use policies to set a period beyond which the refresh tokens are no longer accepted. (To do this, use the Refresh Token Max Age property.) You can adjust the lifetime of a refresh token to control when and how often the user is required to reenter credentials, instead of being silently reauthenticated, when using a public client application.

NOTE

The Max Age property is the length of time a single token can be used.

ID tokens

ID tokens are passed to websites and native clients. ID tokens contain profile information about a user. An ID token is bound to a specific combination of user and client. ID tokens are considered valid until their expiry. Usually, a web application matches a user's session lifetime in the application to the lifetime of the ID token issued for the user. You can adjust the lifetime of an ID token to control how often the web application expires the application session, and how often it requires the user to be reauthenticated with Azure AD (either silently or interactively).

Single sign-on session tokens

When a user authenticates with Azure AD, a single sign-on session (SSO) is established with the user's browser and Azure AD. The SSO token, in the form of a cookie, represents this session. The SSO session token is not bound to a specific resource/client application. SSO session tokens can be revoked, and their validity is checked every time they are used.

Azure AD uses two kinds of SSO session tokens: persistent and nonpersistent. Persistent session tokens are stored as persistent cookies by the browser. Nonpersistent session tokens are stored as session cookies. (Session cookies are destroyed when the browser is closed.) Usually, a nonpersistent session token is stored. But, when the user selects the **Keep me signed in** check box during authentication, a persistent session token is stored.

Nonpersistent session tokens have a lifetime of 24 hours. Persistent tokens have a lifetime of 180 days. Anytime an

SSO session token is used within its validity period, the validity period is extended another 24 hours or 180 days, depending on the token type. If an SSO session token is not used within its validity period, it is considered expired and is no longer accepted.

You can use a policy to set the time after the first session token was issued beyond which the session token is no longer accepted. (To do this, use the Session Token Max Age property.) You can adjust the lifetime of a session token to control when and how often a user is required to reenter credentials, instead of being silently authenticated, when using a web application.

Token lifetime policy properties

A token lifetime policy is a type of policy object that contains token lifetime rules. Use the properties of the policy to control specified token lifetimes. If no policy is set, the system enforces the default lifetime value.

Configurable token lifetime properties

| PROPERTY | POLICY PROPERTY STRING | AFFECTS | DEFAULT | MINIMUM | MAXIMUM |
|-------------------------------------|----------------------------------|---|---------------|------------|----------------------------|
| Access Token Lifetime | AccessTokenLifetime ² | Access tokens, ID tokens, SAML2 tokens | 1 hour | 10 minutes | 1 day |
| Refresh Token Max Inactive Time | MaxInactiveTime | Refresh tokens | 90 days | 10 minutes | 90 days |
| Single-Factor Refresh Token Max Age | MaxAgeSingleFactor | Refresh tokens (for any users) | Until-revoked | 10 minutes | Until-revoked ¹ |
| Multi-Factor Refresh Token Max Age | MaxAgeMultiFactor | Refresh tokens (for any users) | Until-revoked | 10 minutes | Until-revoked ¹ |
| Single-Factor Session Token Max Age | MaxAgeSessionSingleFactor | Session tokens (persistent and nonpersistent) | Until-revoked | 10 minutes | Until-revoked ¹ |
| Multi-Factor Session Token Max Age | MaxAgeSessionMultiFactor | Session tokens (persistent and nonpersistent) | Until-revoked | 10 minutes | Until-revoked ¹ |

- ¹365 days is the maximum explicit length that can be set for these attributes.
- ²To ensure the Microsoft Teams Web client works, it is recommended to keep AccessTokenLifetime to greater than 15 minutes for Microsoft Teams.

Exceptions

| PROPERTY | AFFECTS | DEFAULT |
|---|--|---------------|
| Refresh Token Max Age (issued for federated users who have insufficient revocation information ¹) | Refresh tokens (issued for federated users who have insufficient revocation information ¹) | 12 hours |
| Refresh Token Max Inactive Time (issued for confidential clients) | Refresh tokens (issued for confidential clients) | 90 days |
| Refresh Token Max Age (issued for confidential clients) | Refresh tokens (issued for confidential clients) | Until-revoked |

- ¹Federated users who have insufficient revocation information include any users who do not have the

"LastPasswordChangeTimestamp" attribute synced. These users are given this short Max Age because AAD is unable to verify when to revoke tokens that are tied to an old credential (such as a password that has been changed) and must check back in more frequently to ensure that the user and associated tokens are still in good standing. To improve this experience, tenant admins must ensure that they are syncing the "LastPasswordChangeTimestamp" attribute (this can be set on the user object using Powershell or through AADSync).

Policy evaluation and prioritization

You can create and then assign a token lifetime policy to a specific application, to your organization, and to service principals. Multiple policies might apply to a specific application. The token lifetime policy that takes effect follows these rules:

- If a policy is explicitly assigned to the service principal, it is enforced.
- If no policy is explicitly assigned to the service principal, a policy explicitly assigned to the parent organization of the service principal is enforced.
- If no policy is explicitly assigned to the service principal or to the organization, the policy assigned to the application is enforced.
- If no policy has been assigned to the service principal, the organization, or the application object, the default values are enforced. (See the table in [Configurable token lifetime properties](#).)

For more information about the relationship between application objects and service principal objects, see [Application and service principal objects in Azure Active Directory](#).

A token's validity is evaluated at the time the token is used. The policy with the highest priority on the application that is being accessed takes effect.

All timespans used here are formatted according to the C# `TimeSpan` object - D.HH:MM:SS. So 80 days and 30 minutes would be `80.00:30:00`. The leading D can be dropped if zero, so 90 minutes would be `00:90:00`.

NOTE

Here's an example scenario.

A user wants to access two web applications: Web Application A and Web Application B.

Factors:

- Both web applications are in the same parent organization.
- Token Lifetime Policy 1 with a Session Token Max Age of eight hours is set as the parent organization's default.
- Web Application A is a regular-use web application and isn't linked to any policies.
- Web Application B is used for highly sensitive processes. Its service principal is linked to Token Lifetime Policy 2, which has a Session Token Max Age of 30 minutes.

At 12:00 PM, the user starts a new browser session and tries to access Web Application A. The user is redirected to Azure AD and is asked to sign in. This creates a cookie that has a session token in the browser. The user is redirected back to Web Application A with an ID token that allows the user to access the application.

At 12:15 PM, the user tries to access Web Application B. The browser redirects to Azure AD, which detects the session cookie. Web Application B's service principal is linked to Token Lifetime Policy 2, but it's also part of the parent organization, with default Token Lifetime Policy 1. Token Lifetime Policy 2 takes effect because policies linked to service principals have a higher priority than organization default policies. The session token was originally issued within the last 30 minutes, so it is considered valid. The user is redirected back to Web Application B with an ID token that grants them access.

At 1:00 PM, the user tries to access Web Application A. The user is redirected to Azure AD. Web Application A is not linked to any policies, but because it is in an organization with default Token Lifetime Policy 1, that policy takes effect. The session cookie that was originally issued within the last eight hours is detected. The user is silently redirected back to Web Application A with a new ID token. The user is not required to authenticate.

Immediately afterward, the user tries to access Web Application B. The user is redirected to Azure AD. As before, Token Lifetime Policy 2 takes effect. Because the token was issued more than 30 minutes ago, the user is prompted to reenter their sign-in credentials. A brand-new session token and ID token are issued. The user can then access Web Application B.

Configurable policy property details

Access Token Lifetime

String: AccessTokenLifetime

Affects: Access tokens, ID tokens, SAML tokens

Summary: This policy controls how long access and ID tokens for this resource are considered valid. Reducing the Access Token Lifetime property mitigates the risk of an access token or ID token being used by a malicious actor for an extended period of time. (These tokens cannot be revoked.) The trade-off is that performance is adversely affected, because the tokens have to be replaced more often.

Refresh Token Max Inactive Time

String: MaxInactiveTime

Affects: Refresh tokens

Summary: This policy controls how old a refresh token can be before a client can no longer use it to retrieve a new access/refresh token pair when attempting to access this resource. Because a new refresh token usually is returned when a refresh token is used, this policy prevents access if the client tries to access any resource by using the current refresh token during the specified period of time.

This policy forces users who have not been active on their client to reauthenticate to retrieve a new refresh token.

The Refresh Token Max Inactive Time property must be set to a lower value than the Single-Factor Token Max Age and the Multi-Factor Refresh Token Max Age properties.

Single-Factor Refresh Token Max Age

String: MaxAgeSingleFactor

Affects: Refresh tokens

Summary: This policy controls how long a user can use a refresh token to get a new access/refresh token pair after they last authenticated successfully by using only a single factor. After a user authenticates and receives a new refresh token, the user can use the refresh token flow for the specified period of time. (This is true as long as the current refresh token is not revoked, and it is not left unused for longer than the inactive time.) At that point, the user is forced to reauthenticate to receive a new refresh token.

Reducing the max age forces users to authenticate more often. Because single-factor authentication is considered less secure than multi-factor authentication, we recommend that you set this property to a value that is equal to or lesser than the Multi-Factor Refresh Token Max Age property.

Multi-Factor Refresh Token Max Age

String: MaxAgeMultiFactor

Affects: Refresh tokens

Summary: This policy controls how long a user can use a refresh token to get a new access/refresh token pair after they last authenticated successfully by using multiple factors. After a user authenticates and receives a new refresh token, the user can use the refresh token flow for the specified period of time. (This is true as long as the current refresh token is not revoked, and it is not unused for longer than the inactive time.) At that point, users are forced to reauthenticate to receive a new refresh token.

Reducing the max age forces users to authenticate more often. Because single-factor authentication is considered less secure than multi-factor authentication, we recommend that you set this property to a value that is equal to or greater than the Single-Factor Refresh Token Max Age property.

Single-Factor Session Token Max Age

String: MaxAgeSessionSingleFactor

Affects: Session tokens (persistent and nonpersistent)

Summary: This policy controls how long a user can use a session token to get a new ID and session token after they last authenticated successfully by using only a single factor. After a user authenticates and receives a new session token, the user can use the session token flow for the specified period of time. (This is true as long as the current session token is not revoked and has not expired.) After the specified period of time, the user is forced to reauthenticate to receive a new session token.

Reducing the max age forces users to authenticate more often. Because single-factor authentication is considered less secure than multi-factor authentication, we recommend that you set this property to a value that is equal to or less than the Multi-Factor Session Token Max Age property.

Multi-Factor Session Token Max Age

String: MaxAgeSessionMultiFactor

Affects: Session tokens (persistent and nonpersistent)

Summary: This policy controls how long a user can use a session token to get a new ID and session token after the last time they authenticated successfully by using multiple factors. After a user authenticates and receives a new session token, the user can use the session token flow for the specified period of time. (This is true as long as the current session token is not revoked and has not expired.) After the specified period of time, the user is forced to reauthenticate to receive a new session token.

Reducing the max age forces users to authenticate more often. Because single-factor authentication is considered less secure than multi-factor authentication, we recommend that you set this property to a value that is equal to or greater than the Single-Factor Session Token Max Age property.

Example token lifetime policies

Many scenarios are possible in Azure AD when you can create and manage token lifetimes for apps, service principals, and your overall organization. In this section, we walk through a few common policy scenarios that can help you impose new rules for:

- Token Lifetime
- Token Max Inactive Time
- Token Max Age

In the examples, you can learn how to:

- Manage an organization's default policy
- Create a policy for web sign-in
- Create a policy for a native app that calls a web API
- Manage an advanced policy

Prerequisites

In the following examples, you create, update, link, and delete policies for apps, service principals, and your overall organization. If you are new to Azure AD, we recommend that you learn about [how to get an Azure AD tenant](#) before you proceed with these examples.

To get started, do the following steps:

1. Download the latest [Azure AD PowerShell Module Public Preview release](#).
2. Run the `Connect` command to sign in to your Azure AD admin account. Run this command each time you start a new session.

```
Connect-AzureAD -Confirm
```

3. To see all policies that have been created in your organization, run the following command. Run this command after most operations in the following scenarios. Running the command also helps you get the *** of your policies.

```
Get-AzureADPolicy
```

Example: Manage an organization's default policy

In this example, you create a policy that lets your users' sign in less frequently across your entire organization. To do this, create a token lifetime policy for Single-Factor Refresh Tokens, which is applied across your organization. The policy is applied to every application in your organization, and to each service principal that doesn't already have a policy set.

1. Create a token lifetime policy.

- Set the Single-Factor Refresh Token to "until-revoked." The token doesn't expire until access is revoked.

Create the following policy definition:

```
@('{
    "TokenLifetimePolicy": {
        "Version":1,
        "MaxAgeSingleFactor":"until-revoked"
    }
}')
```

- To create the policy, run the following command:

```
$policy = New-AzureADPolicy -Definition @('{"TokenLifetimePolicy":{"Version":1,
    "MaxAgeSingleFactor":"until-revoked"}}') -DisplayName "OrganizationDefaultPolicyScenario" -
IsOrganizationDefault $true -Type "TokenLifetimePolicy"
```

- To see your new policy, and to get the policy's **ObjectId**, run the following command:

```
Get-AzureADPolicy -Id $policy.Id
```

2. Update the policy.

You might decide that the first policy you set in this example is not as strict as your service requires. To set your Single-Factor Refresh Token to expire in two days, run the following command:

```
Set-AzureADPolicy -Id $policy.Id -DisplayName $policy.DisplayName -Definition @('{"TokenLifetimePolicy": {
    "Version":1,"MaxAgeSingleFactor":"2.00:00:00"}')
```

Example: Create a policy for web sign-in

In this example, you create a policy that requires users to authenticate more frequently in your web app. This policy sets the lifetime of the access/ID tokens and the max age of a multi-factor session token to the service principal of your web app.

1. Create a token lifetime policy.

This policy, for web sign-in, sets the access/ID token lifetime and the max single-factor session token age to two hours.

- To create the policy, run this command:

```
$policy = New-AzureADPolicy -Definition @('{"TokenLifetimePolicy": {"Version":1,"AccessTokenLifetime":"02:00:00","MaxAgeSessionSingleFactor":"02:00:00"} }') -DisplayName "WebPolicyScenario" -IsOrganizationDefault $false -Type "TokenLifetimePolicy"
```

- b. To see your new policy, and to get the policy **ObjectId**, run the following command:

```
Get-AzureADPolicy -Id $policy.Id
```

2. Assign the policy to your service principal. You also need to get the **ObjectId** of your service principal.

- a. Use the [Get-AzureADServicePrincipal](#) cmdlet to see all your organization's service principals or a single service principal.

```
# Get ID of the service principal  
$sp = Get-AzureADServicePrincipal -Filter "DisplayName eq '<service principal display name>'"
```

- b. When you have the service principal, run the following command:

```
# Assign policy to a service principal  
Add-AzureADServicePrincipalPolicy -Id $sp.ObjectId -RefObjectId $policy.Id
```

Example: Create a policy for a native app that calls a web API

In this example, you create a policy that requires users to authenticate less frequently. The policy also lengthens the amount of time a user can be inactive before the user must reauthenticate. The policy is applied to the web API. When the native app requests the web API as a resource, this policy is applied.

1. Create a token lifetime policy.

- a. To create a strict policy for a web API, run the following command:

```
$policy = New-AzureADPolicy -Definition @('{"TokenLifetimePolicy": {"Version":1,"MaxInactiveTime":"30.00:00:00","MaxAgeMultiFactor":"until-revoked","MaxAgeSingleFactor":"180.00:00:00"} }') -DisplayName "WebApiDefaultPolicyScenario" -IsOrganizationDefault $false -Type "TokenLifetimePolicy"
```

- b. To see your new policy, run the following command:

```
Get-AzureADPolicy -Id $policy.Id
```

2. Assign the policy to your web API. You also need to get the **ObjectId** of your application. Use the [Get-AzureADApplication](#) cmdlet to find your app's **ObjectId**, or use the [Azure portal](#).

Get the **ObjectId** of your app and assign the policy:

```
# Get the application  
$app = Get-AzureADApplication -Filter "DisplayName eq 'Fourth Coffee Web API'"  
  
# Assign the policy to your web API.  
Add-AzureADApplicationPolicy -Id $app.ObjectId -RefObjectId $policy.Id
```

Example: Manage an advanced policy

In this example, you create a few policies to learn how the priority system works. You also learn how to manage multiple policies that are applied to several objects.

1. Create a token lifetime policy.

- To create an organization default policy that sets the Single-Factor Refresh Token lifetime to 30 days, run the following command:

```
$policy = New-AzureADPolicy -Definition @('{"TokenLifetimePolicy": {"Version":1,"MaxAgeSingleFactor":"30.00:00:00"} }') -DisplayName "ComplexPolicyScenario" -IsOrganizationDefault $true -Type "TokenLifetimePolicy"
```

- To see your new policy, run the following command:

```
Get-AzureADPolicy -Id $policy.Id
```

2. Assign the policy to a service principal.

Now, you have a policy that applies to the entire organization. You might want to preserve this 30-day policy for a specific service principal, but change the organization default policy to the upper limit of "until-revoked."

- To see all your organization's service principals, you use the [Get-AzureADServicePrincipal](#) cmdlet.
- When you have the service principal, run the following command:

```
# Get ID of the service principal
$sp = Get-AzureADServicePrincipal -Filter "DisplayName eq '<service principal display name>'"

# Assign policy to a service principal
Add-AzureADServicePrincipalPolicy -Id $sp.ObjectId -RefObjectId $policy.Id
```

3. Set the `IsOrganizationDefault` flag to false:

```
Set-AzureADPolicy -Id $policy.Id -DisplayName "ComplexPolicyScenario" -IsOrganizationDefault $false
```

4. Create a new organization default policy:

```
New-AzureADPolicy -Definition @('{"TokenLifetimePolicy": {"Version":1,"MaxAgeSingleFactor":"until-revoked"} }') -DisplayName "ComplexPolicyScenarioTwo" -IsOrganizationDefault $true -Type "TokenLifetimePolicy"
```

You now have the original policy linked to your service principal, and the new policy is set as your organization default policy. It's important to remember that policies applied to service principals have priority over organization default policies.

Cmdlet reference

Manage policies

You can use the following cmdlets to manage policies.

New-AzureADPolicy

Creates a new policy.

```
New-AzureADPolicy -Definition <Array of Rules> -DisplayName <Name of Policy> -IsOrganizationDefault <boolean> -Type <Policy Type>
```

| PARAMETERS | DESCRIPTION | EXAMPLE |
|-------------------------|---|--|
| <code>Definition</code> | Array of stringified JSON that contains all the policy's rules. | <code>-Definition @('{"TokenLifetimePolicy": {"Version":1,"MaxInactiveTime":"20:00:00"} }')</code> |

| PARAMETERS | DESCRIPTION | EXAMPLE |
|--|--|---|
| <code>-DisplayName</code> | String of the policy name. | <code>-DisplayName "MyTokenPolicy"</code> |
| <code>-IsOrganizationDefault</code> | If true, sets the policy as the organization's default policy. If false, does nothing. | <code>-IsOrganizationDefault \$true</code> |
| <code>-Type</code> | Type of policy. For token lifetimes, always use "TokenLifetimePolicy." | <code>-Type "TokenLifetimePolicy"</code> |
| <code>-AlternativeIdentifier</code> [Optional] | Sets an alternative ID for the policy. | <code>-AlternativeIdentifier "myAltId"</code> |

Get-AzureADPolicy

Gets all Azure AD policies or a specified policy.

| Get-AzureADPolicy | | |
|-----------------------------|--|---|
| PARAMETERS | DESCRIPTION | EXAMPLE |
| <code>-Id</code> [Optional] | ObjectId (ID) of the policy you want. | <code>-Id <ObjectId of Policy></code> |

Get-AzureADPolicyAppliedObject

Gets all apps and service principals that are linked to a policy.

| Get-AzureADPolicyAppliedObject -Id <ObjectId of Policy> | | |
|---|--|---|
| PARAMETERS | DESCRIPTION | EXAMPLE |
| <code>-Id</code> | ObjectId (ID) of the policy you want. | <code>-Id <ObjectId of Policy></code> |

Set-AzureADPolicy

Updates an existing policy.

| Set-AzureADPolicy -Id <ObjectId of Policy> -DisplayName <string> | | |
|--|---|---|
| PARAMETERS | DESCRIPTION | EXAMPLE |
| <code>-Id</code> | ObjectId (ID) of the policy you want. | <code>-Id <ObjectId of Policy></code> |
| <code>-DisplayName</code> | String of the policy name. | <code>-DisplayName "MyTokenPolicy"</code> |
| <code>-Definition</code> [Optional] | Array of stringified JSON that contains all the policy's rules. | <code>-Definition @('{"TokenLifetimePolicy": {"Version":1,"MaxInactiveTime":"20:00:00"}}')</code> |

| PARAMETERS | DESCRIPTION | EXAMPLE |
|--|--|---|
| <code>-IsOrganizationDefault</code> [Optional] | If true, sets the policy as the organization's default policy. If false, does nothing. | <code>-IsOrganizationDefault \$true</code> |
| <code>-Type</code> [Optional] | Type of policy. For token lifetimes, always use "TokenLifetimePolicy." | <code>-Type "TokenLifetimePolicy"</code> |
| <code>-AlternativeIdentifier</code> [Optional] | Sets an alternative ID for the policy. | <code>-AlternativeIdentifier "myAltId"</code> |

Remove-AzureADPolicy

Deletes the specified policy.

```
Remove-AzureADPolicy -Id <ObjectId of Policy>
```

| PARAMETERS | DESCRIPTION | EXAMPLE |
|------------------|--|---|
| <code>-Id</code> | ObjectId (ID) of the policy you want. | <code>-Id <ObjectId of Policy></code> |

Application policies

You can use the following cmdlets for application policies.

Add-AzureADApplicationPolicy

Links the specified policy to an application.

```
Add-AzureADApplicationPolicy -Id <ObjectId of Application> -RefObjectId <ObjectId of Policy>
```

| PARAMETERS | DESCRIPTION | EXAMPLE |
|---------------------------|--|--|
| <code>-Id</code> | ObjectId (ID) of the application. | <code>-Id <ObjectId of Application></code> |
| <code>-RefObjectId</code> | ObjectId of the policy. | <code>-RefObjectId <ObjectId of Policy></code> |

Get-AzureADApplicationPolicy

Gets the policy that is assigned to an application.

```
Get-AzureADApplicationPolicy -Id <ObjectId of Application>
```

| PARAMETERS | DESCRIPTION | EXAMPLE |
|------------------|--|--|
| <code>-Id</code> | ObjectId (ID) of the application. | <code>-Id <ObjectId of Application></code> |

Remove-AzureADApplicationPolicy

Removes a policy from an application.

```
Remove-AzureADApplicationPolicy -Id <ObjectId of Application> -PolicyId <ObjectId of Policy>
```

| PARAMETERS | DESCRIPTION | EXAMPLE |
|-----------------|--|--------------------------------|
| Id | ObjectId (ID) of the application. | -Id <ObjectId of Application> |
| PolicyId | ObjectId of the policy. | -PolicyId <ObjectId of Policy> |

Service principal policies

You can use the following cmdlets for service principal policies.

Add-AzureADServicePrincipalPolicy

Links the specified policy to a service principal.

```
Add-AzureADServicePrincipalPolicy -Id <ObjectId of ServicePrincipal> -RefObjectId <ObjectId of Policy>
```

| PARAMETERS | DESCRIPTION | EXAMPLE |
|--------------------|--|-----------------------------------|
| Id | ObjectId (ID) of the application. | -Id <ObjectId of Application> |
| RefObjectId | ObjectId of the policy. | -RefObjectId <ObjectId of Policy> |

Get-AzureADServicePrincipalPolicy

Gets any policy linked to the specified service principal.

```
Get-AzureADServicePrincipalPolicy -Id <ObjectId of ServicePrincipal>
```

| PARAMETERS | DESCRIPTION | EXAMPLE |
|------------|--|-------------------------------|
| Id | ObjectId (ID) of the application. | -Id <ObjectId of Application> |

Remove-AzureADServicePrincipalPolicy

Removes the policy from the specified service principal.

```
Remove-AzureADServicePrincipalPolicy -Id <ObjectId of ServicePrincipal> -PolicyId <ObjectId of Policy>
```

| PARAMETERS | DESCRIPTION | EXAMPLE |
|-----------------|--|--------------------------------|
| Id | ObjectId (ID) of the application. | -Id <ObjectId of Application> |
| PolicyId | ObjectId of the policy. | -PolicyId <ObjectId of Policy> |

Transitioning from App registrations (Legacy) to the new App registrations experience in the Azure portal

11/12/2019 • 6 minutes to read • [Edit Online](#)

You can find many improvements in the new [App registrations](#) experience in the Azure portal. If you're familiar with the App registrations (legacy) experience in the Azure portal, use this training guide to get started using the new experience.

In Azure Active Directory, the new application registration experience described here is generally available (GA). In Azure Active Directory B2C (Azure AD B2C), this experience is in preview.

Key changes

- App registrations aren't limited to being either a *web app/API* or a *native* app. You can use the same app registration for all of these apps by registering the respective redirect URLs.
- The legacy experience supported apps that sign in by using organizational (Azure AD) accounts only. Apps were registered as single-tenant. Apps supported only organizational accounts from the directory that the app was registered in. Apps could be modified to be multi-tenant and support all organizational accounts. The new experience allows you to register apps that can support both those options as well as a third option: all organizational accounts as well as personal Microsoft accounts.
- The legacy experience was only available when signed into the Azure portal using an organizational account. With the new experience, you can use personal Microsoft accounts that aren't associated with a directory.

List of applications

The new app list shows applications that were registered through the legacy app registrations experience in the Azure portal. These apps sign in by using Azure AD accounts. The new app list also shows apps registered through the Application Registration Portal. These apps sign in by using Azure AD and personal Microsoft accounts.

NOTE

The Application Registration Portal has been deprecated.

The new app list doesn't have an **Application type** column because a single app registration can be several types. The list has two additional columns: **Created on** and **Certificates & secrets**. **Certificates & secrets** shows the status of credentials that have been registered on the app. Statuses include **Current**, **Expiring soon**, and **Expired**.

New app registration

In the legacy experience, to register an app you were required to provide: **Name**, **Application type**, and **Sign-on URL/Redirect URI**. The apps that were created were Azure AD only single-tenant applications. They only supported organizational accounts from the directory the app was registered in.

In the new experience, you must provide a **Name** for the app and choose the **Supported account types**. You can optionally provide a **Redirect URI**. If you provide a redirect URI, you'll need to specify whether it's web/public (mobile and desktop). For more information, see [Quickstart: Register an application with the Microsoft identity platform](#). For Azure AD B2C, see [Register an application in Azure Active Directory B2C](#).

Differences between the Application Registration Portal and App registrations page

The legacy Properties page

The legacy experience had a **Properties** page. **Properties** had the following fields:

- **Name**
- **Object ID**
- **Application ID**
- **App ID URI**
- **Logo**
- **Home page URL**
- **Logout URL**
- **Terms of service URL**
- **Privacy statement URL**
- **Application type**
- **Multi-tenant**

The new experience doesn't have that page. Here's where you can find the equivalent functionality:

- **Name, Logo, Home page URL, Terms of service URL, and Privacy statement URL** are now on the app's **Branding** page.
- **Object ID** and **Application (client) ID** are on the **Overview** page.
- The functionality controlled by the **Multi-tenant** toggle in the legacy experience has been replaced by **Supported account types** on the **Authentication** page. For more information, see [Quickstart: Modify the accounts supported by an application](#).
- **Logout URL** is now on the **Authentication** page.
- **Application type** is no longer a valid field. Instead, redirect URIs, which you can find on the **Authentication** page, determine which app types are supported.
- **App ID URI** is now called **Application ID URI** and you can find it on **Expose an API**. In the legacy experience, this property was autoregistered using the following format: `https://{{tenantdomain}}/{{appID}}`, for example, `https://microsoft.onmicrosoft.com/492439af-3282-44c3-b297-45463339544b`. In the new experience, it's autogenerated as `api://{{appID}}`, but it needs to be explicitly saved. In Azure AD B2C tenants, the `https://{{tenantdomain}}/{{appID}}` format is still used.

Reply URLs/redirect URIs

In the legacy experience, an app had a **Reply URLs** page. In the new experience, reply URLs can be found on an app's **Authentication** page. They're now referred to as **Redirect URIs**.

The format for redirect URIs has changed. They're required to be associated with an app type, either web or public. For security reasons, wildcards and `http://` schemes aren't supported, except for `http://localhost`.

Keys/Certificates & secrets

In the legacy experience, an app had **Keys** page. In the new experience, it has been renamed to **Certificates & secrets**.

Public keys are now referred to as **Certificates**. **Passwords** are now referred to as **Client secrets**.

Required permissions/API permissions

In the legacy experience, an app had a **Required permissions** page. In the new experience, it has been renamed to **API permissions**.

When you selected an API in the legacy experience, you could choose from a small list of Microsoft APIs. You could

also search through service principals in the tenant. In the new experience, you can choose from multiple tabs: **Microsoft APIs**, **APIs my organization uses**, or **My APIs**. The search bar on **APIs my organization** uses tab searches through service principals in the tenant.

NOTE

You won't see this tab if your application isn't associated with a tenant. For more information on how to request permissions, see [Quickstart: Configure a client application to access web APIs](#).

The legacy experience had a **Grant permissions** button at the top of the **Requested permissions** page. In the new experience, the **Grant consent** page has a **Grant admin consent** button on an app's **API permissions** section. There are also some differences in the ways the buttons function.

In the legacy experience, the logic varied depending on the signed in user and the permissions being requested. The logic was:

- If only user consent-able permissions were being requested and the signed in user wasn't an admin, the user could grant user consent for the requested permissions.
- If at least one permission that requires admin consent was requested and the signed in user wasn't an admin, the user got an error when attempting to grant consent.
- If the signed in user was an admin, admin consent was granted for all the requested permissions.

In the new experience, only an admin can grant consent. When an admin selects **Grant admin consent**, admin consent is granted to all the requested permissions.

Deleting an app registration

In the legacy experience, you could delete only single-tenant apps. The delete button was disabled for multi-tenant apps. In the new experience, you can delete apps in any state, but you must confirm the action. For more information, see [Quickstart: Remove an application registered with the Microsoft identity platform](#).

Application manifest

The legacy and new experiences use different versions for the format of the JSON in the manifest editor. For more information, see [Azure Active Directory app manifest](#).

New UI

The new experience adds UI controls for the following properties:

- The **Authentication** page has **Implicit grant flow** (`oauth2AllowImplicitFlow`). Unlike in the legacy experience, you can enable **Access tokens** or **ID tokens**, or both.
- The **Expose an API** page contains **Scopes defined by this API** (`oauth2Permissions`) and **Authorized client applications** (`preAuthorizedApplications`). For more information on how to configure an app to be a web API and expose permissions/scopes, see [Quickstart: Configure an application to expose web APIs](#).
- The **Branding** page contains the **Publisher domain**. The publisher domain is displayed to users on the application's **consent prompt**. For more information, see [How to: Configure an application's publisher domain](#).

Limitations

The new experience has the following limitations:

- The format of client secrets (app passwords) is different than that of the legacy experience and may break CLI.
- Changing the value for supported accounts is not supported in the UI. You need to use the app manifest unless

you're switching between Azure AD single-tenant and multi-tenant.

Transitioning from Application Registration Portal to the new App registrations experience in the Azure portal

11/12/2019 • 6 minutes to read • [Edit Online](#)

There are many improvements in the new [App registrations](#) experience in the Azure portal. If you're more familiar with the Application registration portal (apps.dev.microsoft.com) experience for registering or managing converged applications, referred to as the old experience, this training guide will get you started using the new experience.

What's not changing?

- Your applications and related configurations can be found as-is in the new experience. You do not need to register the applications again and users of your applications will not need to sign-in again.

NOTE

You must sign-in with the account you used to register applications to find them in the Azure portal. We recommend you check the signed in user in the Azure portal matches the user that was signed into the Application registration portal by comparing the email address from your profile.

In some cases, especially when you sign in using personal Microsoft accounts(e.g. Outlook, Live, Xbox, etc.) with an Azure AD email address, we found out that when you go to the Azure portal from the old experience, it signs you into a different account with the same email in your Azure AD tenant. If you still believe your applications are missing, sign out and sign in with the right account.

- Live SDK apps created using personal Microsoft accounts are not yet supported in the Azure portal and will continue to remain in the old experience in near future.

Key changes

- In the old experience, apps were by default registered as converged apps supporting all organizational accounts (multitenant) as well as personal Microsoft accounts. This could not be modified through the old experience, making it difficult to create apps that supported only organizational accounts (either multitenant or single tenant). The new experience allows you to register apps supporting all those options. [Learn more about app types](#).
- In the new experience, if your personal Microsoft account is also in an Azure AD tenant, you will see three tabs--all applications in the tenant, owned applications in the tenant as well as applications from your personal account. So, if you believe that apps registered with your personal Microsoft account are missing, check the **Applications from your personal account** tab.
- In the new experience, you can easily switch between tenants by navigating to your profile and choosing switch directory.

List of applications

- The new app list shows applications that were registered through the legacy app registrations experience in the Azure portal (apps that sign in Azure AD accounts only) as well as apps registered though the [Application registration portal](#) (apps that sign in both Azure AD and personal Microsoft accounts).

- The new app list has two additional columns: **Created on** column and **Certificates & secrets** column that shows the status (current, expiring soon, or expired) of credentials that have been registered on the app.

New app registration

In the old experience, to register a converged app you were only required to provide a Name. The apps that were created were registered as converged apps supporting all organizational directory (multitenant) as well as personal Microsoft accounts. This could not be modified through the old experience, making it difficult to create apps that supported only organizational accounts (either multitenant or single tenant). [Learn more about supported account types](#)

In the new experience, you must provide a Name for the app and choose the Supported account types. You can optionally provide a redirect URI. If you provide a redirect URI, you'll need to specify if it's web/public (native/mobile and desktop). For more info on how to register an app using the new app registrations experience, see [this quickstart](#).

App management page

The old experience had a single app management page for converged apps with the following sections: Properties, Application secrets, Platforms, Owners, Microsoft Graph Permissions, Profile, and Advanced Options.

The new experience in the Azure portal represents these features into separate pages. Here's where you can find the equivalent functionality:

- Properties - Name and Application ID is on the Overview page.
- Application Secrets is on the Certificates & secrets page
- Platforms configuration is on the Authentication page
- Microsoft Graph permissions is on the API permissions page along with other permissions
- Profile is on Branding page
- Advanced option - Live SDK support is on the Authentication page.

Application secrets/Certificates & secrets

In the new experience, **Application secrets** have been renamed to **Certificates & secrets**. In addition, **Public keys** are referred to as **Certificates** and **Passwords** are referred to as **Client secrets**. We chose to not bring this functionality along in the new experience for security reasons, hence, you can no longer generate a new key pair.

Platforms/Authentication: Reply URLs/redirect URIs

In the old experience, an app had Platforms section for Web, native, and Web API to configure Redirect URLs, Logout URL and Implicit flow.

In the new experience, Reply URLs can be found on an app's Authentication section. In addition, they are referred to as redirect URIs and the format for redirect URIs has changed. They are required to be associated with an app type (web or public client - mobile and desktop). [Learn more](#)

Web APIs are configured in Expose an API page.

NOTE

Try out the new Authentication settings experience where you can configure settings for your application based on the platform or device that you want to target. [Learn more](#)

Microsoft Graph permissions/API permissions

- When selecting an API in the old experience, you could choose from Microsoft Graph APIs only. In the new experience, you can choose from many Microsoft APIs including Microsoft Graph, APIs from your organization and your APIs, this is presented in three tabs: Microsoft APIs, APIs my organization uses, or My APIs. The search bar on APIs my organization uses tab searches through service principals in the tenant.

NOTE

You won't see this tab if your application isn't associated with a tenant. For more info on how to request permissions using the new experience, see [this quickstart](#).

- The old experience did not have a **Grant permissions** button. In the new experience, there's a Grant consent section with a **Grant admin consent** button on an app's API permissions section. Only an admin can grant consent and this button is enabled for admins only. When an admin selects the **Grant admin consent** button, admin consent is granted to all the requested permissions.

Profile

In the old experience, Profile had Logo, Home page URL, Terms of Service URL and Privacy Statement URL configuration. In the new experience, these can be found in Branding page.

Application manifest

In the new experience, Manifest page allows you to edit and update app's attributes. For more info, see [Application manifest](#).

New UI

There's new UI for properties that could previously only be set using the manifest editor or the API, or didn't exist.

- Implicit grant flow (oauth2AllowImplicitFlow) can be found on the Authentication page. Unlike the old experience, you can enable access tokens or ID tokens, or both.
- Scopes defined by this API (oauth2Permissions) and Authorized client applications (preAuthorizedApplications) can be configured through the Expose an API page. For more info on how to configure an app to be a web API and expose permissions/scopes, see [this quickstart](#).
- Publisher domain (which is displayed to users on the [application's consent prompt](#)) can be found on the Branding blade page. For more info on how to configure a publisher domain, see [this how-to](#).

Limitations

The new experience has the following limitations:

- The new experience does not yet support App registrations for Azure AD B2C tenants.
- The new experience does not yet support Live SDK apps created with personal Microsoft accounts.
- Changing the value for supported accounts is not supported in the UI. You need to use the app manifest unless you're switching between Azure AD single-tenant and multi-tenant.

NOTE

If you're a personal Microsoft account user in Azure AD tenant, and the tenant admin has restricted access to Azure portal, you may get an access denied. However, if you come through the shortcut by typing App registrations in the search bar or pinning it, you'll be able to access the new experience.

How to: Sign in any Azure Active Directory user using the multi-tenant application pattern

10/23/2019 • 13 minutes to read • [Edit Online](#)

If you offer a Software as a Service (SaaS) application to many organizations, you can configure your application to accept sign-ins from any Azure Active Directory (Azure AD) tenant. This configuration is called *making your application multi-tenant*. Users in any Azure AD tenant will be able to sign in to your application after consenting to use their account with your application.

If you have an existing application that has its own account system, or supports other kinds of sign-ins from other cloud providers, adding Azure AD sign-in from any tenant is simple. Just register your app, add sign-in code via OAuth2, OpenID Connect, or SAML, and put a "Sign in with Microsoft" button in your application.

NOTE

This article assumes you're already familiar with building a single tenant application for Azure AD. If you're not, start with one of the quickstarts on the [developer guide homepage](#).

There are four simple steps to convert your application into an Azure AD multi-tenant app:

1. [Update your application registration to be multi-tenant](#)
2. [Update your code to send requests to the /common endpoint](#)
3. [Update your code to handle multiple issuer values](#)
4. [Understand user and admin consent and make appropriate code changes](#)

Let's look at each step in detail. You can also jump straight to [this list of multi-tenant samples](#).

Update registration to be multi-tenant

By default, web app/API registrations in Azure AD are single tenant. You can make your registration multi-tenant by finding the **Supported account types** switch on the **Authentication** pane of your application registration in the [Azure portal](#) and setting it to **Accounts in any organizational directory**.

Before an application can be made multi-tenant, Azure AD requires the App ID URI of the application to be globally unique. The App ID URI is one of the ways an application is identified in protocol messages. For a single tenant application, it is sufficient for the App ID URI to be unique within that tenant. For a multi-tenant application, it must be globally unique so Azure AD can find the application across all tenants. Global uniqueness is enforced by requiring the App ID URI to have a host name that matches a verified domain of the Azure AD tenant.

By default, apps created via the Azure portal have a globally unique App ID URI set on app creation, but you can change this value. For example, if the name of your tenant was contoso.onmicrosoft.com then a valid App ID URI would be `https://contoso.onmicrosoft.com/myapp`. If your tenant had a verified domain of `contoso.com`, then a valid App ID URI would also be `https://contoso.com/myapp`. If the App ID URI doesn't follow this pattern, setting an application as multi-tenant fails.

NOTE

Native client registrations as well as [Microsoft identity platform applications](#) are multi-tenant by default. You don't need to take any action to make these application registrations multi-tenant.

Update your code to send requests to /common

In a single tenant application, sign-in requests are sent to the tenant's sign-in endpoint. For example, for contoso.onmicrosoft.com the endpoint would be: <https://login.microsoftonline.com/contoso.onmicrosoft.com>.

Requests sent to a tenant's endpoint can sign in users (or guests) in that tenant to applications in that tenant.

With a multi-tenant application, the application doesn't know up front what tenant the user is from, so you can't send requests to a tenant's endpoint. Instead, requests are sent to an endpoint that multiplexes across all Azure AD tenants: <https://login.microsoftonline.com/common>

When Microsoft identity platform receives a request on the /common endpoint, it signs the user in and, as a consequence, discovers which tenant the user is from. The /common endpoint works with all of the authentication protocols supported by the Azure AD: OpenID Connect, OAuth 2.0, SAML 2.0, and WS-Federation.

The sign-in response to the application then contains a token representing the user. The issuer value in the token tells an application what tenant the user is from. When a response returns from the /common endpoint, the issuer value in the token corresponds to the user's tenant.

IMPORTANT

The /common endpoint is not a tenant and is not an issuer, it's just a multiplexer. When using /common, the logic in your application to validate tokens needs to be updated to take this into account.

Update your code to handle multiple issuer values

Web applications and web APIs receive and validate tokens from Microsoft identity platform.

NOTE

While native client applications request and receive tokens from Microsoft identity platform, they do so to send them to APIs, where they are validated. Native applications do not validate tokens and must treat them as opaque.

Let's look at how an application validates tokens it receives from Microsoft identity platform. A single tenant application normally takes an endpoint value like:

```
https://login.microsoftonline.com/contoso.onmicrosoft.com
```

and uses it to construct a metadata URL (in this case, OpenID Connect) like:

```
https://login.microsoftonline.com/contoso.onmicrosoft.com/.well-known/openid-configuration
```

to download two critical pieces of information that are used to validate tokens: the tenant's signing keys and issuer value. Each Azure AD tenant has a unique issuer value of the form:

```
https://sts.windows.net/31537af4-6d77-4bb9-a681-d2394888ea26/
```

where the GUID value is the rename-safe version of the tenant ID of the tenant. If you select the preceding metadata link for contoso.onmicrosoft.com, you can see this issuer value in the document.

When a single tenant application validates a token, it checks the signature of the token against the signing keys from the metadata document. This test allows it to make sure the issuer value in the token matches the one that was found in the metadata document.

Because the /common endpoint doesn't correspond to a tenant and isn't an issuer, when you examine the issuer value in the metadata for /common it has a templated URL instead of an actual value:

```
https://sts.windows.net/{tenantid}/
```

Therefore, a multi-tenant application can't validate tokens just by matching the issuer value in the metadata with the `issuer` value in the token. A multi-tenant application needs logic to decide which issuer values are valid and which are not based on the tenant ID portion of the issuer value.

For example, if a multi-tenant application only allows sign-in from specific tenants who have signed up for their service, then it must check either the issuer value or the `tid` claim value in the token to make sure that tenant is in their list of subscribers. If a multi-tenant application only deals with individuals and doesn't make any access decisions based on tenants, then it can ignore the issuer value altogether.

In the [multi-tenant samples](#), issuer validation is disabled to enable any Azure AD tenant to sign in.

Understand user and admin consent

For a user to sign in to an application in Azure AD, the application must be represented in the user's tenant. This allows the organization to do things like apply unique policies when users from their tenant sign in to the application. For a single tenant application, this registration is simple; it's the one that happens when you register the application in the [Azure portal](#).

For a multi-tenant application, the initial registration for the application lives in the Azure AD tenant used by the developer. When a user from a different tenant signs in to the application for the first time, Azure AD asks them to consent to the permissions requested by the application. If they consent, then a representation of the application called a *service principal* is created in the user's tenant, and sign-in can continue. A delegation is also created in the directory that records the user's consent to the application. For details on the application's Application and ServicePrincipal objects, and how they relate to each other, see [Application objects and service principal objects](#).

This consent experience is affected by the permissions requested by the application. Microsoft identity platform supports two kinds of permissions, app-only and delegated.

- A delegated permission grants an application the ability to act as a signed in user for a subset of the things the user can do. For example, you can grant an application the delegated permission to read the signed in user's calendar.
- An app-only permission is granted directly to the identity of the application. For example, you can grant an application the app-only permission to read the list of users in a tenant, regardless of who is signed in to the application.

Some permissions can be consented to by a regular user, while others require a tenant administrator's consent.

Admin consent

App-only permissions always require a tenant administrator's consent. If your application requests an app-only permission and a user tries to sign in to the application, an error message is displayed saying the user isn't able to consent.

Certain delegated permissions also require a tenant administrator's consent. For example, the ability to write back to Azure AD as the signed in user requires a tenant administrator's consent. Like app-only permissions, if an ordinary user tries to sign in to an application that requests a delegated permission that requires administrator consent, your application receives an error. Whether a permission requires admin consent is determined by the developer that published the resource, and can be found in the documentation for the resource. The permissions documentation for the [Azure AD Graph API](#) and [Microsoft Graph API](#) indicate which permissions require admin

consent.

If your application uses permissions that require admin consent, you need to have a gesture such as a button or link where the admin can initiate the action. The request your application sends for this action is the usual OAuth2/OpenID Connect authorization request that also includes the `prompt=admin_consent` query string parameter. Once the admin has consented and the service principal is created in the customer's tenant, subsequent sign-in requests do not need the `prompt=admin_consent` parameter. Since the administrator has decided the requested permissions are acceptable, no other users in the tenant are prompted for consent from that point forward.

A tenant administrator can disable the ability for regular users to consent to applications. If this capability is disabled, admin consent is always required for the application to be used in the tenant. If you want to test your application with end-user consent disabled, you can find the configuration switch in the [Azure portal](#) in the **User settings** section under **Enterprise applications**.

The `prompt=admin_consent` parameter can also be used by applications that request permissions that do not require admin consent. An example of when this would be used is if the application requires an experience where the tenant admin "signs up" one time, and no other users are prompted for consent from that point on.

If an application requires admin consent and an admin signs in without the `prompt=admin_consent` parameter being sent, when the admin successfully consents to the application it will apply **only for their user account**. Regular users will still not be able to sign in or consent to the application. This feature is useful if you want to give the tenant administrator the ability to explore your application before allowing other users access.

NOTE

Some applications want an experience where regular users are able to consent initially, and later the application can involve the administrator and request permissions that require admin consent. There is no way to do this with a v1.0 application registration in Azure AD today; however, using the Microsoft identity platform (v2.0) endpoint allows applications to request permissions at runtime instead of at registration time, which enables this scenario. For more information, see [Microsoft identity platform endpoint](#).

Consent and multi-tier applications

Your application may have multiple tiers, each represented by its own registration in Azure AD. For example, a native application that calls a web API, or a web application that calls a web API. In both of these cases, the client (native app or web app) requests permissions to call the resource (web API). For the client to be successfully consented into a customer's tenant, all resources to which it requests permissions must already exist in the customer's tenant. If this condition isn't met, Azure AD returns an error that the resource must be added first.

Multiple tiers in a single tenant

This can be a problem if your logical application consists of two or more application registrations, for example a separate client and resource. How do you get the resource into the customer tenant first? Azure AD covers this case by enabling client and resource to be consented in a single step. The user sees the sum total of the permissions requested by both the client and resource on the consent page. To enable this behavior, the resource's application registration must include the client's App ID as a `knownClientApplications` in its [application manifest](#). For example:

```
knownClientApplications": ["94da0930-763f-45c7-8d26-04d5938baab2"]
```

This is demonstrated in a multi-tier native client calling web API sample in the [Related content](#) section at the end of this article. The following diagram provides an overview of consent for a multi-tier app registered in a single tenant.

Multiple tiers in multiple tenants

A similar case happens if the different tiers of an application are registered in different tenants. For example,

consider the case of building a native client application that calls the Office 365 Exchange Online API. To develop the native application, and later for the native application to run in a customer's tenant, the Exchange Online service principal must be present. In this case, the developer and customer must purchase Exchange Online for the service principal to be created in their tenants.

If it's an API built by an organization other than Microsoft, the developer of the API needs to provide a way for their customers to consent the application into their customers' tenants. The recommended design is for the third-party developer to build the API such that it can also function as a web client to implement sign-up. To do this:

1. Follow the earlier sections to ensure the API implements the multi-tenant application registration/code requirements.
2. In addition to exposing the API's scopes/roles, make sure the registration includes the "Sign in and read user profile" permission (provided by default).
3. Implement a sign-in/sign-up page in the web client and follow the [admin consent](#) guidance.
4. Once the user consents to the application, the service principal and consent delegation links are created in their tenant, and the native application can get tokens for the API.

The following diagram provides an overview of consent for a multi-tier app registered in different tenants.

Revoking consent

Users and administrators can revoke consent to your application at any time:

- Users revoke access to individual applications by removing them from their [Access Panel Applications](#) list.
- Administrators revoke access to applications by removing them using the [Enterprise applications](#) section of the [Azure portal](#).

If an administrator consents to an application for all users in a tenant, users cannot revoke access individually. Only the administrator can revoke access, and only for the whole application.

Multi-tenant applications and caching access tokens

Multi-tenant applications can also get access tokens to call APIs that are protected by Azure AD. A common error when using the Active Directory Authentication Library (ADAL) with a multi-tenant application is to initially request a token for a user using /common, receive a response, then request a subsequent token for that same user also using /common. Because the response from Azure AD comes from a tenant, not /common, ADAL caches the token as being from the tenant. The subsequent call to /common to get an access token for the user misses the cache entry, and the user is prompted to sign in again. To avoid missing the cache, make sure subsequent calls for an already signed in user are made to the tenant's endpoint.

Next steps

In this article, you learned how to build an application that can sign in a user from any Azure AD tenant. After enabling Single Sign-On (SSO) between your app and Azure AD, you can also update your application to access APIs exposed by Microsoft resources like Office 365. This lets you offer a personalized experience in your application, such as showing contextual information to the users, like their profile picture or their next calendar appointment. To learn more about making API calls to Azure AD and Office 365 services like Exchange, SharePoint, OneDrive, OneNote, and more, visit [Microsoft Graph API](#).

Related content

- [Multi-tenant application samples](#)
- [Branding guidelines for applications](#)
- [Application objects and service principal objects](#)

- Integrating applications with Azure Active Directory
- Overview of the Consent Framework
- Microsoft Graph API permission scopes
- Azure AD Graph API permission scopes

How to: Use Azure PowerShell to create a service principal with a certificate

10/23/2019 • 6 minutes to read • [Edit Online](#)

When you have an app or script that needs to access resources, you can set up an identity for the app and authenticate the app with its own credentials. This identity is known as a service principal. This approach enables you to:

- Assign permissions to the app identity that are different than your own permissions. Typically, these permissions are restricted to exactly what the app needs to do.
- Use a certificate for authentication when executing an unattended script.

IMPORTANT

Instead of creating a service principal, consider using managed identities for Azure resources for your application identity. If your code runs on a service that supports managed identities and accesses resources that support Azure Active Directory (Azure AD) authentication, managed identities are a better option for you. To learn more about managed identities for Azure resources, including which services currently support it, see [What is managed identities for Azure resources?](#).

This article shows you how to create a service principal that authenticates with a certificate. To set up a service principal with password, see [Create an Azure service principal with Azure PowerShell](#).

You must have the [latest version](#) of PowerShell for this article.

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

Required permissions

To complete this article, you must have sufficient permissions in both your Azure AD and Azure subscription. Specifically, you must be able to create an app in the Azure AD, and assign the service principal to a role.

The easiest way to check whether your account has adequate permissions is through the portal. See [Check required permission](#).

Assign the application to a role

To access resources in your subscription, you must assign the application to a role. Decide which role offers the right permissions for the application. To learn about the available roles, see [RBAC: Built in Roles](#).

You can set the scope at the level of the subscription, resource group, or resource. Permissions are inherited to lower levels of scope. For example, adding an application to the *Reader* role for a resource group means it can read the resource group and any resources it contains. To allow the application to execute actions like reboot, start and stop instances, select the *Contributor* role.

Create service principal with self-signed certificate

The following example covers a simple scenario. It uses [New-AzADServicePrincipal](#) to create a service principal with a self-signed certificate, and uses [New-AzureRmRoleAssignment](#) to assign the [Reader](#) role to the service principal. The role assignment is scoped to your currently selected Azure subscription. To select a different subscription, use [Set-AzContext](#).

NOTE

The New-SelfSignedCertificate cmdlet and the PKI module are currently not supported in PowerShell Core.

```
$cert = New-SelfSignedCertificate -CertStoreLocation "cert:\CurrentUser\My" `  
-Subject "CN=exampleappScriptCert" `  
-KeySpec KeyExchange  
$keyValue = [System.Convert]::ToBase64String($cert.GetRawCertData())  
  
$sp = New-AzADServicePrincipal -DisplayName exampleapp `  
-CertValue $keyValue `  
-EndDate $cert.NotAfter `  
-StartDate $cert.NotBefore  
Sleep 20  
New-AzRoleAssignment -RoleDefinitionName Reader -ServicePrincipalName $sp.ApplicationId
```

The example sleeps for 20 seconds to allow some time for the new service principal to propagate throughout Azure AD. If your script doesn't wait long enough, you'll see an error stating: "Principal {ID} does not exist in the directory {DIR-ID}." To resolve this error, wait a moment then run the [New-AzRoleAssignment](#) command again.

You can scope the role assignment to a specific resource group by using the **ResourceGroupName** parameter. You can scope to a specific resource by also using the **ResourceType** and **ResourceName** parameters.

If you **do not have Windows 10 or Windows Server 2016**, you need to download the [Self-signed certificate generator](#) from Microsoft Script Center. Extract its contents and import the cmdlet you need.

```
# Only run if you could not use New-SelfSignedCertificate  
Import-Module -Name c:\ExtractedModule\New-SelfSignedCertificateEx.ps1
```

In the script, substitute the following two lines to generate the certificate.

```
New-SelfSignedCertificateEx -StoreLocation CurrentUser `  
-Subject "CN=exampleapp" `  
-KeySpec "Exchange" `  
-FriendlyName "exampleapp"  
$cert = Get-ChildItem -path Cert:\CurrentUser\my | where {$PSitem.Subject -eq 'CN=exampleapp' }
```

Provide certificate through automated PowerShell script

Whenever you sign in as a service principal, you need to provide the tenant ID of the directory for your AD app. A tenant is an instance of Azure AD.

```

$TenantId = (Get-AzSubscription -SubscriptionName "Contoso Default").TenantId
$ApplicationId = (Get-AzADApplication -DisplayNameStartWith exampleapp).ApplicationId

$Thumbprint = (Get-ChildItem cert:\CurrentUser\My\ | Where-Object {$_.Subject -eq "CN=exampleappScriptCert"}).Thumbprint
Connect-AzAccount -ServicePrincipal `-
-CertificateThumbprint $Thumbprint `-
-ApplicationId $ApplicationId `-
-TenantId $TenantId

```

Create service principal with certificate from Certificate Authority

The following example uses a certificate issued from a Certificate Authority to create service principal. The assignment is scoped to the specified Azure subscription. It adds the service principal to the [Reader](#) role. If an error occurs during the role assignment, it retries the assignment.

```

Param (
    [Parameter(Mandatory=$true)]
    [String] $ApplicationDisplayName,
    [Parameter(Mandatory=$true)]
    [String] $SubscriptionId,
    [Parameter(Mandatory=$true)]
    [String] $CertPath,
    [Parameter(Mandatory=$true)]
    [String] $CertPlainPassword
)

Connect-AzAccount
Import-Module Az.Resources
Set-AzContext -Subscription $SubscriptionId

$CertPassword = ConvertTo-SecureString $CertPlainPassword -AsPlainText -Force

$PFXCert = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Certificate2 -ArgumentList
@($CertPath, $CertPassword)
$keyValue = [System.Convert]::ToBase64String($PFXCert.GetRawCertData())

$ServicePrincipal = New-AzADServicePrincipal -DisplayName $ApplicationDisplayName
New-AzADSPrincipals -ObjectId $ServicePrincipal.Id -CertValue $keyValue -StartDate $PFXCert.NotBefore -
EndDate $PFXCert.NotAfter
Get-AzADServicePrincipal -ObjectId $ServicePrincipal.Id

$NewRole = $null
$Retries = 0;
While ($NewRole -eq $null -and $Retries -le 6)
{
    # Sleep here for a few seconds to allow the service principal application to become active (should only
    # take a couple of seconds normally)
    Sleep 15
    New-AzRoleAssignment -RoleDefinitionName Reader -ServicePrincipalName $ServicePrincipal.ApplicationId | 
    Write-Verbose -ErrorAction SilentlyContinue
    $NewRole = Get-AzRoleAssignment -ObjectId $ServicePrincipal.Id -ErrorAction SilentlyContinue
    $Retries++;
}

$NewRole

```

Provide certificate through automated PowerShell script

Whenever you sign in as a service principal, you need to provide the tenant ID of the directory for your AD app. A

tenant is an instance of Azure AD.

```
Param (
    [Parameter(Mandatory=$true)]
    [String] $CertPath,
    [Parameter(Mandatory=$true)]
    [String] $CertPlainPassword,
    [Parameter(Mandatory=$true)]
    [String] $ApplicationId,
    [Parameter(Mandatory=$true)]
    [String] $TenantId
)

$CertPassword = ConvertTo-SecureString $CertPlainPassword -AsPlainText -Force
$PFXCert = New-Object `-
    -TypeName System.Security.Cryptography.X509Certificates.X509Certificate2 `-
    -ArgumentList @($CertPath, $CertPassword)
$Thumbprint = $PFXCert.Thumbprint

Connect-AzAccount -ServicePrincipal `-
    -CertificateThumbprint $Thumbprint `-
    -ApplicationId $ApplicationId `-
    -TenantId $TenantId
```

The application ID and tenant ID aren't sensitive, so you can embed them directly in your script. If you need to retrieve the tenant ID, use:

```
(Get-AzSubscription -SubscriptionName "Contoso Default").TenantId
```

If you need to retrieve the application ID, use:

```
(Get-AzADApplication -DisplayNameStartWith {display-name}).ApplicationId
```

Change credentials

To change the credentials for an AD app, either because of a security compromise or a credential expiration, use the [Remove-AzADAppCredential](#) and [New-AzADAppCredential](#) cmdlets.

To remove all the credentials for an application, use:

```
Get-AzADApplication -DisplayName exampleapp | Remove-AzADAppCredential
```

To add a certificate value, create a self-signed certificate as shown in this article. Then, use:

```
Get-AzADApplication -DisplayName exampleapp | New-AzADAppCredential `-
    -CertValue $keyValue `-
    -EndDate $cert.NotAfter `-
    -StartDate $cert.NotBefore
```

Debug

You may get the following errors when creating a service principal:

- "**Authentication_Unauthorized**" or "**No subscription found in the context.**" - You see this error when your account doesn't have the [required permissions](#) on the Azure AD to register an app. Typically, you see this error when only admin users in your Azure Active Directory can register apps, and your account isn't an admin. Ask your administrator to either assign you to an administrator role, or to enable users to register apps.
- Your account "**does not have authorization to perform action 'Microsoft.Authorization/roleAssignments/write' over scope '/subscriptions/{guid}'.**" - You see this error when your account doesn't have sufficient permissions to assign a role to an identity. Ask your subscription administrator to add you to User Access Administrator role.

Next steps

- To set up a service principal with password, see [Create an Azure service principal with Azure PowerShell](#).
- For a more detailed explanation of applications and service principals, see [Application Objects and Service Principal Objects](#).
- For more information about Azure AD authentication, see [Authentication Scenarios for Azure AD](#).

How to: Use the portal to create an Azure AD application and service principal that can access resources

11/4/2019 • 6 minutes to read • [Edit Online](#)

This article shows you how to create a new Azure Active Directory (Azure AD) application and service principal that can be used with the role-based access control. When you have code that needs to access or modify resources, you can create an identity for the app. This identity is known as a service principal. You can then assign the required permissions to the service principal. This article shows you how to use the portal to create the service principal. It focuses on a single-tenant application where the application is intended to run within only one organization. You typically use single-tenant applications for line-of-business applications that run within your organization.

IMPORTANT

Instead of creating a service principal, consider using managed identities for Azure resources for your application identity. If your code runs on a service that supports managed identities and accesses resources that support Azure AD authentication, managed identities are a better option for you. To learn more about managed identities for Azure resources, including which services currently support it, see [What is managed identities for Azure resources?](#).

Create an Azure Active Directory application

Let's jump straight into creating the identity. If you run into a problem, check the [required permissions](#) to make sure your account can create the identity.

1. Sign in to your Azure Account through the [Azure portal](#).
2. Select **Azure Active Directory**.
3. Select **App registrations**.
4. Select **New registration**.
5. Name the application. Select a supported account type, which determines who can use the application.

Under **Redirect URI**, select **Web** for the type of application you want to create. Enter the URI where the access token is sent to. You can't create credentials for a [Native application](#). You can't use that type for an automated application. After setting the values, select **Register**.

Register an application

⚠ If you are building an application for external users that will be distributed by Microsoft, you must register as a first party application to meet all security, privacy, and compliance policies. [Read our decision guide](#)

* Name

The user-facing display name for this application (this can be changed later).

example-app



Supported account types

Who can use this application or access this API?

Accounts in this organizational directory only (Microsoft)

Accounts in any organizational directory

Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)

[Help me choose...](#)

Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web



<https://contoso.org/exampleapp>



[By proceeding, you agree to the Microsoft Platform Policies](#)

[Register](#)

You've created your Azure AD application and service principal.

Assign the application to a role

To access resources in your subscription, you must assign the application to a role. Decide which role offers the right permissions for the application. To learn about the available roles, see [RBAC: Built in Roles](#).

You can set the scope at the level of the subscription, resource group, or resource. Permissions are inherited to lower levels of scope. For example, adding an application to the Reader role for a resource group means it can read the resource group and any resources it contains.

1. In the Azure portal, select the level of scope you wish to assign the application to. For example, to assign a role at the subscription scope, search for and select **Subscriptions**, or select **Subscriptions** on the **Home** page.

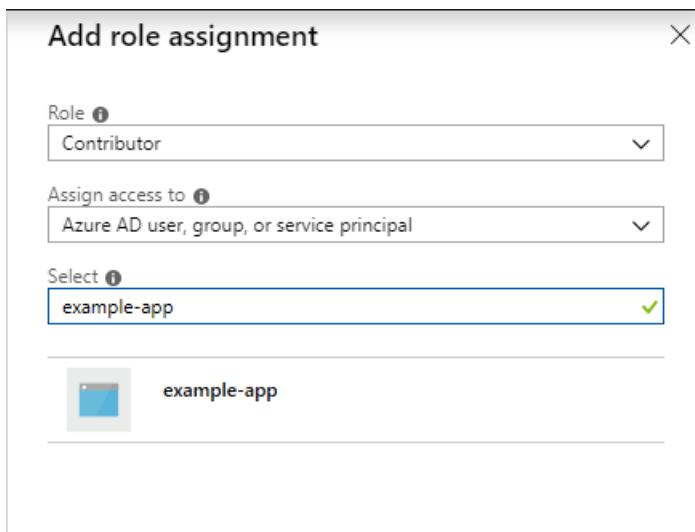
The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with 'Azure services' and 'Recent resources'. The main area is titled 'Subscriptions'. A red box highlights the 'Subscriptions' link under 'Services'. Below it, there are links for 'Event Grid Subscriptions', 'Resource groups', and 'Manage subscriptions in the Billing/Account Center'. Under 'Resources', there's a single entry: 'APEX C+L - Aquent Vendor Subscriptions'. Under 'Resource Groups', it says 'No results were found.' In the bottom right of the main area, there are buttons for 'Documentation' (with a 'All' dropdown), 'Scaling with multiple Azure subscriptions - Microsoft ...', 'Azure subscription limits and quotas | Microsoft Docs', and 'Create an additional Azure subscription | Microsoft Docs'. At the bottom, there's a search bar and buttons for 'Subscriptions', 'Resource groups', and 'All resources'.

2. Select the particular subscription to assign the application to.

The screenshot shows the 'Subscriptions' page in the Microsoft Azure portal. It has a header with 'Dashboard > Subscriptions'. Below that is a section titled 'Subscriptions' with a 'Microsoft' logo and a '+ Add' button. It says 'Showing subscriptions in Microsoft. Don't see a subscription? [Switch directories](#)'. There's a dropdown for 'My role' set to '8 selected' with an 'Apply' button. A checked checkbox says 'Show only subscriptions selected in the [global subscriptions filter](#)'. Below is a search bar. The main area lists 'SUBSCRIPTION' and 'SUBSCRIPTION ID'. A red box highlights the 'Internal testing subscription' row.

If you don't see the subscription you're looking for, select **global subscriptions filter**. Make sure the subscription you want is selected for the portal.

3. Select **Access control (IAM)**.
4. Select **Add role assignment**.
5. Select the role you wish to assign to the application. For example, to allow the application to execute actions like **reboot**, **start** and **stop** instances, select the **Contributor** role. Read more about the [available roles](#) By default, Azure AD applications aren't displayed in the available options. To find your application, search for the name and select it.



6. Select **Save** to finish assigning the role. You see your application in the list of users assigned to a role for that scope.

Your service principal is set up. You can start using it to run your scripts or apps. The next section shows how to get values that are needed when signing in programmatically.

Get values for signing in

When programmatically signing in, you need to pass the tenant ID with your authentication request. You also need the ID for your application and an authentication key. To get those values, use the following steps:

1. Select **Azure Active Directory**.
2. From **App registrations** in Azure AD, select your application.
3. Copy the Directory (tenant) ID and store it in your application code.

Home > Microsoft - App registrations > example-app

example-app

Search (Ctrl+ /)

Delete Endpoints

Welcome to the new and improved App registrations. Looking to learn how it's changed from

| | |
|------------|--|
| Overview | Display name : example-app |
| Quickstart | Application (client) ID : 8f3f32e1-1041-4761-9e04-4ec0600010c7 |
| Manage | Directory (tenant) ID : 72f93d7f-0007-48eb-a4e6-2d17e15111f7 |
| Branding | Object ID : 50600046-0001-4712-8172-017701010101 |

Copy to clipboard

4. Copy the **Application ID** and store it in your application code.

Home > Microsoft - App registrations > example-app

example-app

Search (Ctrl+ /)

Delete Endpoints

Welcome to the new and improved App registrations. Looking to learn how it's changed from

| | |
|------------|--|
| Overview | Display name : example-app |
| Quickstart | Application (client) ID : 8f3f32e1-1041-4761-9e04-4ec0600010c7 |
| Manage | Directory (tenant) ID : 72f93d7f-0007-48eb-a4e6-2d17e15111f7 |
| Branding | Object ID : 50600046-0001-4712-8172-017701010101 |

Copy to clipboard

Certificates and secrets

Daemon applications can use two forms of credentials to authenticate with Azure AD: certificates and application secrets. We recommend using a certificate, but you can also create a new application secret.

Upload a certificate

You can use an existing certificate if you have one. Optionally, you can create a self-signed certificate for testing purposes. Open PowerShell and run [New-SelfSignedCertificate](#) with the following parameters to create a self-signed certificate in the user certificate store on your computer:

```
$cert = New-SelfSignedCertificate -Subject "CN=DaemonConsoleCert" -CertStoreLocation "Cert:\CurrentUser\My" -KeyExportPolicy Exportable -KeySpec Signature
```

Export this certificate to a file using the [Manage User Certificate](#) MMC snap-in accessible from the Windows Control Panel.

To upload the certificate:

1. Select **Certificates & secrets**.
2. Select **Upload certificate** and select the certificate (an existing certificate or the self-signed certificate you exported).

Certificates

Certificates can be used as secrets to prove the application's identity when requesting a token. Also can be referred to as public keys.

Upload certificate

| THUMBPRINT | START DATE | EXPIRES |
|------------|------------|---------|
| | | |

No certificates have been added for this application.

3. Select **Add**.

After registering the certificate with your application in the application registration portal, you need to enable the client application code to use the certificate.

Create a new application secret

If you choose not to use a certificate, you can create a new application secret.

1. Select **Certificates & secrets**.
2. Select **Client secrets -> New client secret**.
3. Provide a description of the secret, and a duration. When done, select **Add**.

After saving the client secret, the value of the client secret is displayed. Copy this value because you aren't able to retrieve the key later. You provide the key value with the application ID to sign in as the application. Store the key value where your application can retrieve it.

Client secrets

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

New client secret

| DESCRIPTION | EXPIRES | VALUE |
|-------------|-----------|-------------------------------------|
| demo secret | 5/14/2020 | nWu9HVZ7Rnj.2y7XSkVyUngZ][x9Z:e [n] |

Configure access policies on resources

Keep in mind, you might need to configure addition permissions on resources that your application needs to access. For example, you must also [update a key vault's access policies](#) to give your application access to keys, secrets, or certificates.

1. In the [Azure portal](#), navigate to your key vault and select **Access policies**.
2. Select **Add access policy**, then select the key, secret, and certificate permissions you want to grant your application. Select the service principal you created previously.
3. Select **Add** to add the access policy, then **Save** to commit your changes.

The screenshot shows the 'Access policies' page for a key vault named 'example-key-vault'. The left sidebar has a red box around the 'Access policies' item. The top right has a red box around the 'Save' button. A tooltip above the 'Save' button says 'Please click 'Save' button to commit your changes.' Below the 'Save' button is a blue bar with the text '+ Add Access Policy...'. The main area shows a table of current access policies, with one row for 'example-app' under the 'APPLICATION' category. The 'Action' column for this row contains four buttons: 'List', 'List', 'List', and 'Delete'.

Required permissions

You must have sufficient permissions to register an application with your Azure AD tenant, and assign the application to a role in your Azure subscription.

Check Azure AD permissions

1. Select **Azure Active Directory**.
2. Note your role. If you have the **User** role, you must make sure that non-administrators can register applications.

The screenshot shows the 'Microsoft - Overview' page in Azure Active Directory. The left sidebar has a red box around the 'Enterprise applications' item. The main area shows a 'Sign-ins' section with a note: 'Only global administrators, security administrators, security readers, and report readers can view sign-ins. [More info](#)'. Below this note, the text 'Your role User' is highlighted with a red box.

3. In the left pane, select **User settings**.
4. Check the **App registrations** setting. This value can only be set by an administrator. If set to **Yes**, any user in

the Azure AD tenant can register an app.

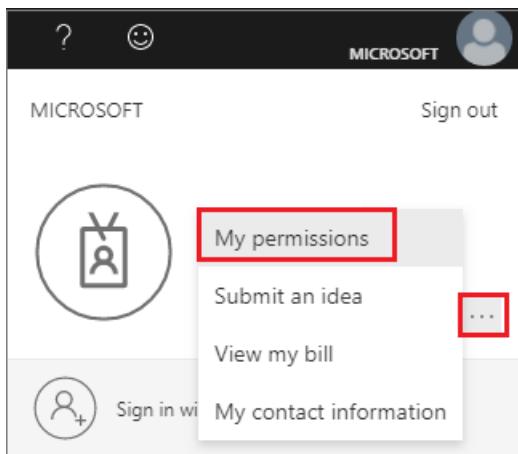
If the app registrations setting is set to **No**, only users with an administrator role may register these types of applications. See [available roles](#) and [role permissions](#) to learn about available administrator roles and the specific permissions in Azure AD that are given to each role. If your account is assigned to the User role, but the app registration setting is limited to admin users, ask your administrator to either assign you to one of the administrator roles that can create and manage all aspects of app registrations, or to enable users to register apps.

Check Azure subscription permissions

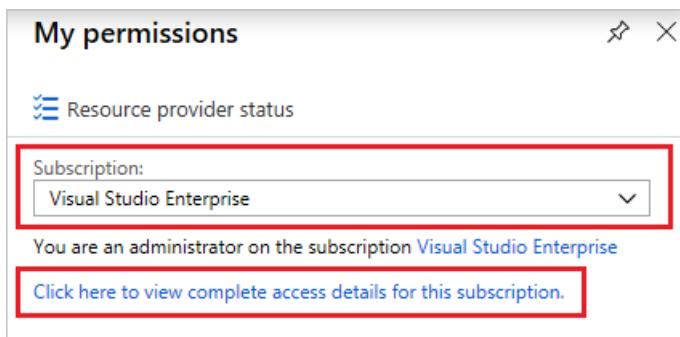
In your Azure subscription, your account must have `Microsoft.Authorization/*/Write` access to assign an AD app to a role. This action is granted through the [Owner](#) role or [User Access Administrator](#) role. If your account is assigned to the **Contributor** role, you don't have adequate permission. You receive an error when attempting to assign the service principal to a role.

To check your subscription permissions:

1. Select your account in the upper right corner, and select... -> **My permissions**.



2. From the drop-down list, select the subscription you want to create the service principal in. Then, select [Click here to view complete access details for this subscription](#).



3. Select **Role assignments** to view your assigned roles, and determine if you have adequate permissions to assign an AD app to a role. If not, ask your subscription administrator to add you to User Access Administrator role. In the following image, the user is assigned to the Owner role, which means that user has adequate permissions.

Add **Remove** **Roles** **Refresh** **Help**

Name **i** Type **i** Role **i** Scope **i**
Search by name or email All 2 selected All scopes

2 items (1 Users, 1 Service Principals)

| <input type="checkbox"/> NAME | TYPE | ROLE |
|--|------|--|
| OWNER
<input checked="" type="checkbox"/> Example User
example@contoso.org | User | Owner i , Service administrator |

Next steps

- To learn about specifying security policies, see [Azure Role-based Access Control](#).
- For a list of available actions that can be granted or denied to users, see [Azure Resource Manager Resource Provider operations](#).

How to: Restrict your app to a set of users

10/23/2019 • 4 minutes to read • [Edit Online](#)

Applications registered in an Azure Active Directory (Azure AD) tenant are, by default, available to all users of the tenant who authenticate successfully.

Similarly, in case of a [multi-tenant](#) app, all users in the Azure AD tenant where this app is provisioned will be able to access this application once they successfully authenticate in their respective tenant.

Tenant administrators and developers often have requirements where an app must be restricted to a certain set of users. Developers can accomplish the same by using popular authorization patterns like Role Based Access Control (RBAC), but this approach requires a significant amount of work on part of the developer.

Azure AD allows tenant administrators and developers to restrict an app to a specific set of users or security groups in the tenant.

Supported app configurations

The option to restrict an app to a specific set of users or security groups in a tenant works with the following types of applications:

- Applications configured for federated single sign-on with SAML-based authentication
- Application proxy applications that use Azure AD pre-authentication
- Applications built directly on the Azure AD application platform that use OAuth 2.0/OpenID Connect authentication after a user or admin has consented to that application.

NOTE

This feature is available for web app/web API and enterprise applications only. Apps that are registered as [native](#) cannot be restricted to a set of users or security groups in the tenant.

Update the app to enable user assignment

There are two ways to create an application with enabled user assignment. One requires the **Global Administrator** role, the second does not.

Enterprise applications (requires the Global Administrator role)

1. Go to the [Azure portal](#) and sign in as a **Global Administrator**.
2. On the top bar, select the signed-in account.
3. Under **Directory**, select the Azure AD tenant where the app will be registered.
4. In the navigation on the left, select **Azure Active Directory**. If Azure Active Directory is not available in the navigation pane, follow these steps:
 - a. Select **All services** at the top of the main left-hand navigation menu.
 - b. Type in **Azure Active Directory** in the filter search box, and then select the **Azure Active Directory** item from the result.
5. In the **Azure Active Directory** pane, select **Enterprise Applications** from the **Azure Active Directory** left-hand navigation menu.

6. Select **All Applications** to view a list of all your applications.

If you do not see the application you want show up here, use the various filters at the top of the **All applications** list to restrict the list or scroll down the list to locate your application.

7. Select the application you want to assign a user or security group to from the list.

8. On the application's **Overview** page, select **Properties** from the application's left-hand navigation menu.

9. Locate the setting **User assignment required?** and set it to **Yes**. When this option is set to **Yes**, users must first be assigned to this application before they can access it.

10. Select **Save** to save this configuration change.

App registration

1. Go to the [Azure portal](#).

2. On the top bar, select the signed-in account.

3. Under **Directory**, select the Azure AD tenant where the app will be registered.

4. In the navigation on the left, select **Azure Active Directory**.

5. In the **Azure Active Directory** pane, select **App Registrations** from the **Azure Active Directory** left-hand navigation menu.

6. Create or select the app you want to manage. You need to be **Owner** of this app registration.

7. On the application's **Overview** page, follow the **Managed application in local directory** link under the essentials in the top of the page. This will take you to the *managed Enterprise Application* of your app registration.

8. From the navigation blade on the left, select **Properties**.

9. Locate the setting **User assignment required?** and set it to **Yes**. When this option is set to **Yes**, users must first be assigned to this application before they can access it.

10. Select **Save** to save this configuration change.

Assign users and groups to the app

Once you've configured your app to enable user assignment, you can go ahead and assign users and groups to the app.

1. Select the **Users and groups** pane in the application's left-hand navigation menu.

2. At the top of the **Users and groups** list, select the **Add user** button to open the **Add Assignment** pane.

3. Select the **Users** selector from the **Add Assignment** pane.

A list of users and security groups will be shown along with a textbox to search and locate a certain user or group. This screen allows you to select multiple users and groups in one go.

4. Once you are done selecting the users and groups, press the **Select** button on bottom to move to the next part.

5. Press the **Assign** button on the bottom to finish the assignments of users and groups to the app.

6. Confirm that the users and groups you added are showing up in the updated **Users and groups** list.

How to: Add app roles in your application and receive them in the token

11/4/2019 • 4 minutes to read • [Edit Online](#)

Role-based access control (RBAC) is a popular mechanism to enforce authorization in applications. When using RBAC, an administrator grants permissions to roles, and not to individual users or groups. The administrator can then assign roles to different users and groups to control who has access to what content and functionality.

Using RBAC with Application Roles and Role Claims, developers can securely enforce authorization in their apps with little effort on their part.

Another approach is to use Azure AD Groups and Group Claims, as shown in [WebApp-GroupClaims-DotNet](#). Azure AD Groups and Application Roles are by no means mutually exclusive; they can be used in tandem to provide even finer grained access control.

Declare roles for an application

These application roles are defined in the [Azure portal](#) in the application's registration manifest. When a user signs into the application, Azure AD emits a `roles` claim for each role that the user has been granted individually to the user and from their group membership. Assignment of users and groups to roles can be done through the portal's UI, or programmatically using [Microsoft Graph](#).

Declare app roles using Azure portal

1. Sign in to the [Azure portal](#).
2. On the top bar, select your account, and then **Switch Directory**.
3. Once the **Directory + subscription** pane opens, choose the Active Directory tenant where you wish to register your application, from the **Favorites** or **All Directories** list.
4. Select **All services** in the left-hand nav, and choose **Azure Active Directory**.
5. In the **Azure Active Directory** pane, select **App registrations (Legacy)** to view a list of all your applications.

If you do not see the application you want show up here, use the various filters at the top of the **App registrations (Legacy)** list to restrict the list or scroll down the list to locate your application.

6. Select the application you want to define app roles in.
7. In the blade for your application, select **Manifest**.
8. Edit the app manifest by locating the `appRoles` setting and adding all your Application Roles.

NOTE

Each app role definition in this manifest must have a different valid GUID for the `id` property.

The `value` property of each app role definition should exactly match the strings that are used in the code in the application. The `value` property can't contain spaces. If it does, you'll receive an error when you save the manifest.

9. Save the manifest.

Examples

The following example shows the `appRoles` that you can assign to `users`.

NOTE

The `id` must be a unique GUID.

```
"appId": "8763f1c4-f988-489c-a51e-158e9ef97d6a",
"appRoles": [
  {
    "allowedMemberTypes": [
      "User"
    ],
    "displayName": "Writer",
    "id": "d1c2ade8-98f8-45fd-aa4a-6d06b947c66f",
    "isEnabled": true,
    "description": "Writers Have the ability to create tasks.",
    "value": "Writer"
  }
],
"availableToOtherTenants": false,
```

NOTE

The `displayName` cannot contain spaces.

You can define app roles to target `users`, `applications`, or both. When available to `applications`, app roles appear as application permissions in the **Required Permissions** blade. The following example shows an app role targeted towards an `Application`.

```
"appId": "8763f1c4-f988-489c-a51e-158e9ef97d6a",
"appRoles": [
  {
    "allowedMemberTypes": [
      "Application"
    ],
    "displayName": "ConsumerApps",
    "id": "47fbb575-859a-4941-89c9-0f7a6c30beac",
    "isEnabled": true,
    "description": "Consumer apps have access to the consumer data.",
    "value": "Consumer"
  }
],
"availableToOtherTenants": false,
```

The number of roles defined affects the limits that the application manifest has. They have been discussed in detail on the [manifest limits](#) page.

Assign users and groups to roles

Once you've added app roles in your application, you can assign users and groups to these roles.

1. In the **Azure Active Directory** pane, select **Enterprise applications** from the **Azure Active Directory** left-hand navigation menu.
2. Select **All applications** to view a list of all your applications.

If you do not see the application you want show up here, use the various filters at the top of the **All**

applications list to restrict the list or scroll down the list to locate your application.

3. Select the application in which you want to assign users or security group to roles.
4. Select the **Users and groups** pane in the application's left-hand navigation menu.
5. At the top of the **Users and groups** list, select the **Add user** button to open the **Add Assignment** pane.
6. Select the **Users and groups** selector from the **Add Assignment** pane.

A list of users and security groups will be shown along with a textbox to search and locate a certain user or group. This screen allows you to select multiple users and groups in one go.

7. Once you are done selecting the users and groups, press the **Select** button on bottom to move to the next part.
8. Choose the **Select Role** selector from the **Add assignment** pane. All the roles declared earlier in the app manifest will show up.
9. Choose a role and press the **Select** button.
10. Press the **Assign** button on the bottom to finish the assignments of users and groups to the app.
11. Confirm that the users and groups you added are showing up in the updated **Users and groups** list.

More information

- [Authorization in a web app using Azure AD application roles & role claims \(Sample\)](#)
- [Using Security Groups and Application Roles in your apps \(Video\)](#)
- [Azure Active Directory, now with Group Claims and Application Roles](#)
- [Azure Active Directory app manifest](#)
- [AAD Access tokens](#)
- [AAD `id_tokens`](#)

Branding guidelines for applications

10/23/2019 • 5 minutes to read • [Edit Online](#)

When developing applications with Azure Active Directory (Azure AD), you'll need to direct your customers when they want to use their work or school account (managed in Azure AD), or their personal account for sign-up and sign-in to your application.

In this article, you will:

- Learn about the two kinds of user accounts managed by Microsoft and how to refer to Azure AD accounts in your application
- Find out what you need to do add the Microsoft logo for use in your app
- Download the official **Sign in** or **Sign in with Microsoft** images to use in your app
- Learn about the branding and navigation do's and don'ts

Personal accounts vs. work or school accounts from Microsoft

Microsoft manages two kinds of user accounts:

- **Personal accounts** (formerly known as Windows Live ID). These accounts represent the relationship between *individual* users and Microsoft, and are used to access consumer devices and services from Microsoft. These accounts are intended for personal use.
- **Work or school accounts.** These accounts are managed by Microsoft on behalf of organizations that use Azure Active Directory. These accounts are used to sign in to Office 365 and other business services from Microsoft.

Microsoft work or school accounts are typically assigned to end users (employees, students, federal employees) by their organizations (company, school, government agency). These accounts master directly in the cloud (in the Azure AD platform) or synced to Azure AD from an on-premises directory, such as Windows Server Active Directory. Microsoft is the *custodian* of the work or school accounts, but the accounts are owned and controlled by the organization.

Referring to Azure AD accounts in your application

Microsoft doesn't expose end users to the Azure or the Active Directory brand names, and neither should you.

- Once users are signed in, use the organization's name and logo as much as possible. This is better than using generic terms like "your organization."
- When users aren't signed in, refer to their accounts as "Work or school accounts" and use the Microsoft logo to convey that Microsoft manages these accounts. Don't use terms like "enterprise account," "business account," or "corporate account," which create user confusion.

User account pictogram

In an earlier version of these guidelines, we recommended using a "blue badge" pictogram. Based on user and developer feedback, we now recommend the use of the Microsoft logo instead. The Microsoft logo will help users understand that they can reuse the account they use with Office 365 or other Microsoft business services to sign into your app.

Signing up and signing in with Azure AD

Your app may present separate paths for sign-up and sign-in and the following sections provide visual guidance for

both scenarios.

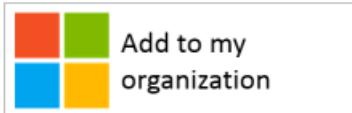
If your app supports end-user sign-up (for example, free to trial or freemium model): You can show a **sign-in** button that allows users to access your app with their work account or their personal account. Azure AD will show a consent prompt the first time they access your app.

If your app requires permissions that only admins can consent to, or if your app requires organizational licensing: Separate admin acquisition from user sign-in. The “**get this app**” button will redirect admins to sign in then ask them to grant consent on behalf of users in their organization, which has the added benefit of suppressing end-user consent prompts to your app.

Visual guidance for app acquisition

Your “get the app” link must redirect the user to the Azure AD grant access (authorize) page, to allow an organization’s administrator to authorize your app to have access to their organization’s data, which is hosted by Microsoft. Details on how to request access are discussed in the [Integrating Applications with Azure Active Directory](#) article.

After admins consent to your app, they can choose to add it to their users’ Office 365 app launcher experience (accessible from the waffle and from <https://portal.office.com/myapps>). If you want to advertise this capability, you can use terms like “Add this app to your organization” and show a button like the following example:



However, we recommend that you write explanatory text instead of relying on buttons. For example:

If you already use Office 365 or other business service from Microsoft, you can grant <your_app_name> access to your organization's data. This will allow your users to access <your_app_name> with their existing work accounts.

To download the official Microsoft logo for use in your app, right-click the one you want to use and then save it to your computer.

| ASSET | PNG FORMAT | SVG FORMAT |
|----------------|------------|------------|
| Microsoft logo | | |

Visual guidance for sign-in

Your app should display a sign-in button that redirects users to the sign-in endpoint that corresponds to the protocol you use to integrate with Azure AD. The following section provides details on what that button should look like.

Pictogram and “Sign in with Microsoft”

It’s the association of the Microsoft logo and the “Sign in with Microsoft” terms that uniquely represent Azure AD amongst other identity providers your app may support. If you don’t have enough space for “Sign in with Microsoft,” it’s ok to shorten it to “Sign in.” You can use a light or dark color scheme for the buttons.

The following diagram shows the Microsoft-recommended redlines when using the assets with your app. The redlines apply to “Sign in with Microsoft” or the shorter “Sign in” version.



Vertically aligned icon and text

12px 12px

12px

Sign in with Microsoft

41px

Font: **Segoe UI Regular 15px**

Font weight: **600**

Font color: **#FFFFFF**

Background: **#2F2F2F**



Vertically aligned icon and text

12px 12px

12px

Sign in with Microsoft

41px

Font: **Segoe UI Regular 15px**

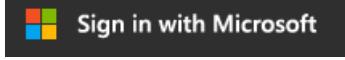
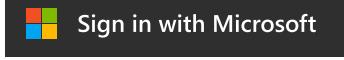
Font weight: **600**

Font color: **#5E5E5E**

Background: **#FFFFFF**

Border: **1px #8C8C8C**

To download the official images for use in your app, right-click the one you want to use and then save it to your computer.

| ASSET | PNG FORMAT | SVG FORMAT |
|--------------------------------------|---|---|
| Sign in with Microsoft (dark theme) |  |  |
| Sign in with Microsoft (light theme) |  |  |
| Sign in (dark theme) |  |  |
| Sign in (light theme) |  |  |

Branding Do's and Don'ts

DO use "work or school account" in combination with the "Sign in with Microsoft" button to provide additional explanation to help end users recognize whether they can use it. **DON'T** use other terms such as "enterprise account", "business account" or "corporate account".

DON'T use "Office 365 ID" or "Azure ID." Office 365 is also the name of a consumer offering from Microsoft, which doesn't use Azure AD for authentication.

DON'T alter the Microsoft logo.

DON'T expose end users to the Azure or Active Directory brands. It's ok however to use these terms with

developers, IT pros, and admins.

Navigation Do's and Don'ts

DO provide a way for users to sign out and switch to another user account. While most people have a single personal account from Microsoft/Facebook/Google/Twitter, people are often associated with more than one organization. Support for multiple signed-in users is coming soon.

How to: Configure an application's publisher domain

11/12/2019 • 4 minutes to read • [Edit Online](#)

An application's publisher domain is displayed to users on the [application's consent prompt](#) to let users know where their information is being sent. Multi-tenant applications that are registered after May 21, 2019 that don't have a publisher domain show up as **unverified**. Multi-tenant applications are applications that support accounts outside of a single organizational directory; for example, support all Azure AD accounts, or support all Azure AD accounts and personal Microsoft accounts.

New applications

When you register a new app, the publisher domain of your app may be set to a default value. The value depends on where the app is registered, particularly whether the app is registered in a tenant and whether the tenant has tenant verified domains.

If there are tenant-verified domains, the app's publisher domain will default to the primary verified domain of the tenant. If there are no tenant verified domains (which is the case when the application is not registered in a tenant), the app's publisher domain will be set to null.

The following table summarizes the default behavior of the publisher domain value.

| TENANT-VERIFIED DOMAINS | DEFAULT VALUE OF PUBLISHER DOMAIN |
|---|-----------------------------------|
| null | null |
| *.onmicrosoft.com | *.onmicrosoft.com |
| - *.onmicrosoft.com
- domain1.com
- domain2.com (primary) | domain2.com |

If a multi-tenant application's publisher domain isn't set, or if it's set to a domain that ends in .onmicrosoft.com, the app's consent prompt will show **unverified** in place of the publisher domain.

Grandfathered applications

If your app was registered before May 21, 2019, your application's consent prompt will not show **unverified** if you have not set a publisher domain. We recommend that you set the publisher domain value so that users can see this information on your app's consent prompt.

Configure publisher domain using the Azure portal

To set your app's publisher domain, follow these steps.

1. Sign in to the [Azure portal](#) using either a work or school account, or a personal Microsoft account.
2. If your account is present in more than one Azure AD tenant:
 - a. Select your profile from the menu on the top-right corner of the page, and then **Switch directory**.
 - b. Change your session to the Azure AD tenant where you want to create your application.
3. Navigate to [Azure Active Directory > App registrations](#) to find and select the app that you want to configure.

Once you've selected the app, you'll see the app's **Overview** page.

4. From the app's **Overview** page, select the **Branding** section.
5. Find the **Publisher domain** field and select one of the following options:
 - Select **Configure a domain** if you haven't configured a domain already.
 - Select **Update domain** if a domain has already been configured.

If your app is registered in a tenant, you'll see two tabs to select from: **Select a verified domain** and **Verify a new domain**.

If your app isn't registered in a tenant, you'll only see the option to verify a new domain for your application.

To verify a new domain for your app

1. Create a file named `microsoft-identity-association.json` and paste the following JSON code snippet.

```
{  
  "associatedApplications": [  
    {  
      "applicationId": "{YOUR-APP-ID-HERE}"  
    }  
  ]  
}
```

2. Replace the placeholder `{YOUR-APP-ID-HERE}` with the application (client) ID that corresponds to your app.
3. Host the file at: `https://{{YOUR-DOMAIN-HERE}}.com/.well-known/microsoft-identity-association.json`. Replace the placeholder `{YOUR-DOMAIN-HERE}` to match the verified domain.
4. Click the **Verify and save domain** button.

To select a verified domain

- If your tenant has verified domains, select one of the domains from the **Select a verified domain** dropdown.

NOTE

The expected 'Content-Type' header that should be returned is `application/json`. You may get an error as mentioned below if you use anything else like `application/json; charset=utf-8`

```
"Verification of publisher domain failed. Error getting JSON file from https://{{well-known/microsoft-identity-association}}. The server returned an unexpected content type header value. "
```

Implications on the app consent prompt

Configuring the publisher domain has an impact on what users see on the app consent prompt. To fully understand the components of the consent prompt, see [Understanding the application consent experiences](#).

The following table describes the behavior for applications created before May 21, 2019.

| Publisher domain | Consent prompt shows: |
|-------------------|---|
| null |  Contoso Test App
App info |
| *.onmicrosoft.com |  Contoso Test App
App info |
| domain2.com |  Contoso Test App
domain2.com |

The behavior for new applications created after May 21, 2019 will depend on the publisher domain and the type of application. The following table describes the changes you should expect to see with the different combinations of configurations.

| Publisher domain | For multi-tenant apps the consent prompt shows: | For AAD only single tenant apps consent prompt shows: |
|-------------------|---|---|
| null |  Contoso Test App
unverified |  Contoso Test App
App info |
| *.onmicrosoft.com |  Contoso Test App
unverified |  Contoso Test App
App info |
| domain2.com |  Contoso Test App
domain2.com |  Contoso Test App
domain2.com |

Implications on redirect URIs

Applications that sign in users with any work or school account, or personal Microsoft accounts ([multi-tenant](#)) are subject to few restrictions when specifying redirect URIs.

Single root domain restriction

When the publisher domain value for multi-tenant apps is set to null, apps are restricted to share a single root domain for the redirect URIs. For example, the following combination of values isn't allowed because the root domain, contoso.com, doesn't match fabrikam.com.

```
"https://contoso.com",  
"https://fabrikam.com",
```

Subdomain restrictions

Subdomains are allowed, but you must explicitly register the root domain. For example, while the following URIs share a single root domain, the combination isn't allowed.

```
"https://app1.contoso.com",  
"https://app2.contoso.com",
```

However, if the developer explicitly adds the root domain, the combination is allowed.

```
"https://contoso.com",  
"https://app1.contoso.com",  
"https://app2.contoso.com",
```

Exceptions

The following cases aren't subject to the single root domain restriction:

- Single tenant apps, or apps that target accounts in a single directory
- Use of localhost as redirect URIs
- Redirect URIs with custom schemes (non-HTTP or HTTPS)

Configure publisher domain programmatically

Currently, there is no REST API or PowerShell support to configure publisher domain programmatically.

How to: Configure terms of service and privacy statement for an app

7/1/2019 • 2 minutes to read • [Edit Online](#)

Developers who build and manage apps that integrate with Azure Active Directory (Azure AD) and Microsoft accounts should include links to the app's terms of service and privacy statement. The terms of service and privacy statement are surfaced to users through the user consent experience. They help your users know that they can trust your app. The terms of service and privacy statement are especially critical for user-facing multi-tenant apps--apps that are used by multiple directories or are available to any Microsoft account.

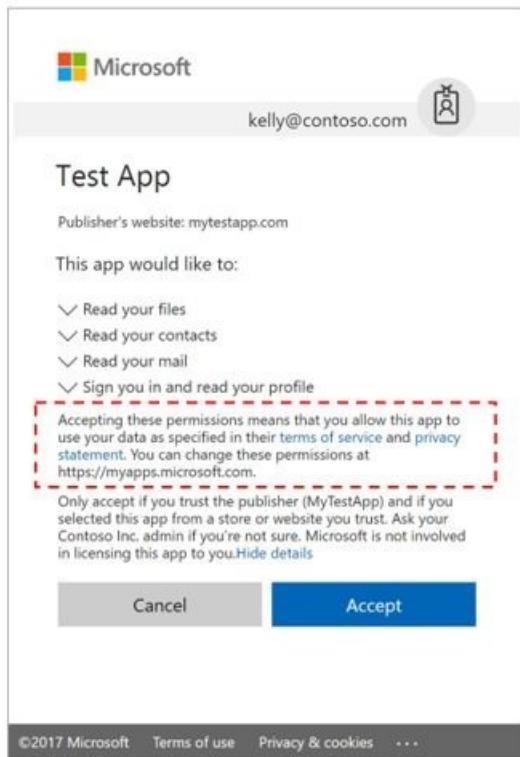
You are responsible for creating the terms of service and privacy statement documents for your app, and for providing the URLs to these documents. For multi-tenant apps that fail to provide these links, the user consent experience for your app will show an alert, which may discourage users from consenting to your app.

NOTE

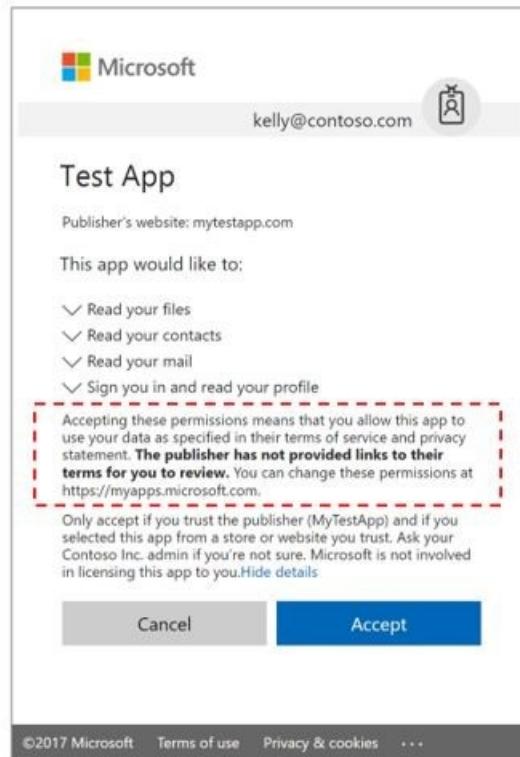
- Single-tenant apps will not show an alert.
- If one or both of the two links are missing, your app will show an alert.

User consent experience

The following examples show the user consent experience when the terms of service and privacy statement are configured and when these links are not configured.



Privacy statement and terms of service have been provided



Privacy statement and terms of service have not been provided

Formatting links to the terms of service and privacy statement

documents

Before you add links to your app's terms of service and privacy statement documents, make sure the URLs follow these guidelines.

| GUIDELINE | DESCRIPTION |
|---------------|--------------------------------------|
| Format | Valid URL |
| Valid schemas | HTTP and HTTPS
We recommend HTTPS |
| Max length | 2048 characters |

Examples: <https://myapp.com/terms-of-service> and <https://myapp.com/privacy-statement>

Adding links to the terms of service and privacy statement

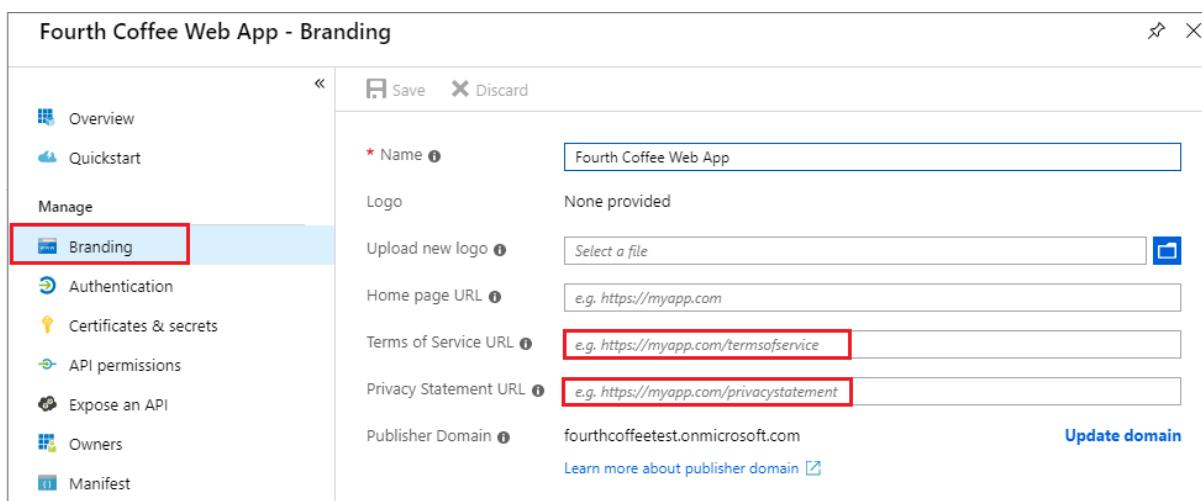
When the terms of service and privacy statement are ready, you can add links to these documents in your app using one of these methods:

- [Through the Azure portal](#)
- [Using the app object JSON](#)
- [Using the MSGraph beta REST API](#)

Using the Azure portal

Follow these steps in the Azure portal.

1. Sign in to the [Azure portal](#).
2. Navigate to the **App Registrations** section and select your app.
3. Open the **Branding** pane.
4. Fill out the **Terms of Service URL** and **Privacy Statement URL** fields.
5. Save your changes.



Using the app object JSON

If you prefer to modify the app object JSON directly, you can use the manifest editor in the Azure portal or Application Registration Portal to include links to your app's terms of service and privacy statement.

```
"informationalUrls": {  
    "termsOfService": "<your_terms_of_service_url>",  
    "privacy": "<your_privacy_statement_url>"  
}
```

Using the MSGraph beta REST API

To programmatically update all your apps, you can use the MSGraph beta REST API to update all your apps to include links to the terms of service and privacy statement documents.

```
PATCH https://graph.microsoft.com/beta/applications/{application id}  
{  
    "appId": "{your application id}",  
    "info": {  
        "termsOfServiceUrl": "<your_terms_of_service_url>",  
        "supportUrl": null,  
        "privacyStatementUrl": "<your_privacy_statement_url>",  
        "marketingUrl": null,  
        "logoUrl": null  
    }  
}
```

NOTE

- Be careful not to overwrite any pre-existing values you have assigned to any of these fields: `supportUrl`, `marketingUrl`, and `logoUrl`
- The MSGraph beta REST API will only work when you sign in with an Azure AD account. Personal Microsoft accounts are not supported.

Get a token from the token cache using MSAL.NET

10/23/2019 • 2 minutes to read • [Edit Online](#)

When you acquire an access token using Microsoft Authentication Library for .NET (MSAL.NET), the token is cached. When the application needs a token, it should first call the `AcquireTokenSilent` method to verify if an acceptable token is in the cache. In many cases, it's possible to acquire another token with more scopes based on a token in the cache. It's also possible to refresh a token when it's getting close to expiration (as the token cache also contains a refresh token).

The recommended pattern is to call the `AcquireTokenSilent` method first. If `AcquireTokenSilent` fails, then acquire a token using other methods.

In the following example, the application first attempts to acquire a token from the token cache. If a `MsalUiRequiredException` exception is thrown, the application acquires a token interactively.

```
AuthenticationResult result = null;
var accounts = await app.GetAccountsAsync();

try
{
    result = await app.AcquireTokenSilent(scopes, accounts.FirstOrDefault())
        .ExecuteAsync();
}
catch (MsalUiRequiredException ex)
{
    // A MsalUiRequiredException happened on AcquireTokenSilent.
    // This indicates you need to call AcquireTokenInteractive to acquire a token
    System.Diagnostics.Debug.WriteLine($"MsalUiRequiredException: {ex.Message}");

    try
    {
        result = await app.AcquireTokenInteractive(scopes)
            .ExecuteAsync();
    }
    catch (MsalException msalex)
    {
        ResultText.Text = $"Error Acquiring Token:{System.Environment.NewLine}{msalex}";
    }
}
catch (Exception ex)
{
    ResultText.Text = $"Error Acquiring Token Silently:{System.Environment.NewLine}{ex}";
    return;
}

if (result != null)
{
    string accessToken = result.AccessToken;
    // Use the token
}
```

Token cache serialization in MSAL.NET

10/23/2019 • 7 minutes to read • [Edit Online](#)

After a [token is acquired](#), it is cached by Microsoft Authentication Library (MSAL). Application code should try to get a token from the cache before acquiring a token by another method. This article discusses default and custom serialization of the token cache in MSAL.NET.

This article is for MSAL.NET 3.x. If you're interested in MSAL.NET 2.x, see [Token cache serialization in MSAL.NET 2.x](#).

Default serialization for mobile platforms

In MSAL.NET, an in-memory token cache is provided by default. Serialization is provided by default for platforms where secure storage is available for a user as part of the platform. This is the case for Universal Windows Platform (UWP), Xamarin.iOS, and Xamarin.Android.

NOTE

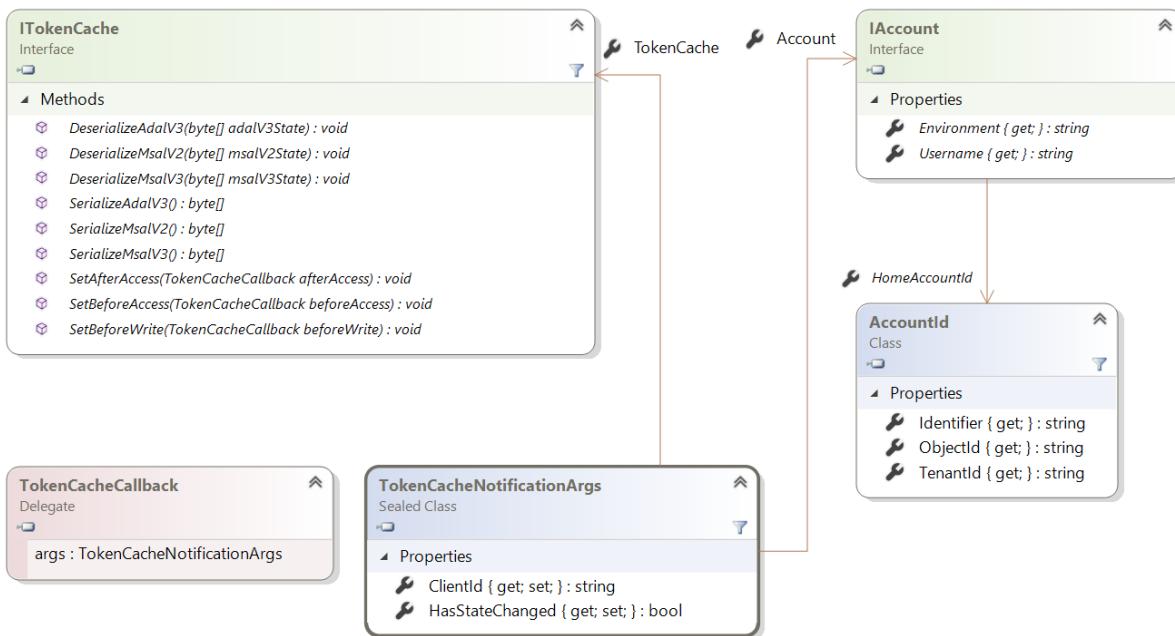
When you migrate a Xamarin.Android project from MSAL.NET 1.x to MSAL.NET 3.x, you might want to add `android:allowBackup="false"` to your project to avoid old cached tokens from coming back when Visual Studio deployments trigger a restore of local storage. See [Issue #659](#).

Custom serialization for Windows desktop apps and web apps/web APIs

Remember, custom serialization isn't available on mobile platforms (UWP, Xamarin.iOS, and Xamarin.Android). MSAL already defines a secure and performant serialization mechanism for these platforms. .NET desktop and .NET Core applications, however, have varied architectures and MSAL can't implement a general-purpose serialization mechanism. For example, web sites may choose to store tokens in a Redis cache, or desktop apps store tokens in an encrypted file. So serialization isn't provided out-of-the-box. To have a persistent token cache application in .NET desktop or .NET Core, you need to customize the serialization.

The following classes and interfaces are used in token cache serialization:

- `ITokenCache`, which defines events to subscribe to token cache serialization requests as well as methods to serialize or de-serialize the cache at various formats (ADAL v3.0, MSAL 2.x, and MSAL 3.x = ADAL v5.0).
- `TokenCacheCallback` is a callback passed to the events so that you can handle the serialization. They'll be called with arguments of type `TokenCacheNotificationArgs`.
- `TokenCacheNotificationArgs` only provides the `ClientId` of the application and a reference to the user for which the token is available.



IMPORTANT

MSAL.NET creates token caches for you and provides you with the `IToken` cache when you call an application's `UserTokenCache` and `AppTokenCache` properties. You are not supposed to implement the interface yourself. Your responsibility, when you implement a custom token cache serialization, is to:

- React to `BeforeAccess` and `AfterAccess` "events" (or their Async flavors). The `BeforeAccess` delegate is responsible to deserialize the cache, whereas the `AfterAccess` one is responsible for serializing the cache.
- Part of these events store or load blobs, which are passed through the event argument to whatever storage you want.

The strategies are different depending on if you're writing a token cache serialization for a [public client application](#) (desktop), or a [confidential client application](#) (web app / web API, daemon app).

Token cache for a public client

Since MSAL.NET v2.x you have several options for serializing the token cache of a public client. You can serialize the cache only to the MSAL.NET format (the unified format cache is common across MSAL and the platforms). You can also support the [legacy](#) token cache serialization of ADAL V3.

Customizing the token cache serialization to share the single sign-on state between ADAL.NET 3.x, ADAL.NET 5.x, and MSAL.NET is explained in part of the following sample: [active-directory-dotnet-v1-to-v2](#).

NOTE

The MSAL.NET 1.1.4-preview token cache format is no longer supported in MSAL 2.x. If you have applications leveraging MSAL.NET 1.x, your users will have to re-sign-in. Alternately, the migration from ADAL 4.x (and 3.x) is supported.

Simple token cache serialization (MSAL only)

Below is an example of a naive implementation of custom serialization of a token cache for desktop applications. Here, the user token cache is a file in the same folder as the application.

After you build the application, you enable the serialization by calling the `TokenCacheHelper.EnableSerialization()` method and passing the application `UserTokenCache`.

```

app = PublicClientApplicationBuilder.Create(ClientId)
    .Build();
TokenCacheHelper.EnableSerialization(app.UserTokenCache);

```

The `TokenCacheHelper` helper class is defined as:

```

static class TokenCacheHelper
{
    public static void EnableSerialization(ITokenCache tokenCache)
    {
        tokenCache.SetBeforeAccess(BeforeAccessNotification);
        tokenCache.SetAfterAccess(AfterAccessNotification);
    }

    ///<summary>
    /// Path to the token cache
    ///</summary>
    public static readonly string CacheFilePath = System.Reflection.Assembly.GetExecutingAssembly().Location +
    ".msalcache.bin3";

    private static readonly object FileLock = new object();

    private static void BeforeAccessNotification(TokenCacheNotificationArgs args)
    {
        lock (FileLock)
        {
            args.TokenCache.DeserializeMsalV3(File.Exists(CacheFilePath)
                ? ProtectedData.Unprotect(File.ReadAllBytes(CacheFilePath),
                    null,
                    DataProtectionScope.CurrentUser)
                : null);
        }
    }

    private static void AfterAccessNotification(TokenCacheNotificationArgs args)
    {
        // if the access operation resulted in a cache update
        if (args.HasStateChanged)
        {
            lock (FileLock)
            {
                // reflect changes in the persistent store
                File.WriteAllBytes(CacheFilePath,
                    ProtectedData.Protect(args.TokenCache.SerializeMsalV3(),
                        null,
                        DataProtectionScope.CurrentUser));
            }
        }
    }
}

```

A preview of a product quality token cache file based serializer for public client applications (for desktop applications running on Windows, Mac and Linux) is available from the [Microsoft.Identity.Client.Extensions.Msal](#) open-source library. You can include it in your applications from the following nuget package:

[Microsoft.Identity.Client.Extensions.Msal](#).

Dual token cache serialization (MSAL unified cache and ADAL v3)

If you want to implement token cache serialization both with the unified cache format (common to ADAL.NET 4.x, MSAL.NET 2.x, and other MSALs of the same generation or older, on the same platform), take a look at the following code:

```

string appLocation = Path.GetDirectoryName(Assembly.GetEntryAssembly().Location);
string cacheFolder = Path.GetFullPath(appLocation) + @"\..\..\..";
string adalV3cacheFileName = Path.Combine(cacheFolder, "cacheAdalV3.bin");
string unifiedCacheFileName = Path.Combine(cacheFolder, "unifiedCache.bin");

IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
    .Build();
FilesBasedTokenCacheHelper.EnableSerialization(app.UserTokenCache,
                                              unifiedCacheFileName,
                                              adalV3cacheFileName);

```

This time the helper class as defined as:

```

using System;
using System.IO;
using System.Security.Cryptography;
using Microsoft.Identity.Client;

namespace CommonCacheMsalV3
{
    /// <summary>
    /// Simple persistent cache implementation of the dual cache serialization (ADAL V3 legacy
    /// and unified cache format) for a desktop applications (from MSAL 2.x)
    /// </summary>
    static class FilesBasedTokenCacheHelper
    {
        /// <summary>
        /// Enables the serialization of the token cache
        /// </summary>
        /// <param name="adalV3CacheFileName">File name where the cache is serialized with the
        /// ADAL V3 token cache format. Can
        /// be <c>null</c> if you don't want to implement the legacy ADAL V3 token cache
        /// serialization in your MSAL 2.x+ application</param>
        /// <param name="unifiedCacheFileName">File name where the cache is serialized
        /// with the Unified cache format, common to
        /// ADAL V4 and MSAL V2 and above, and also across ADAL/MSAL on the same platform.
        /// Should not be <c>null</c></param>
        /// <returns></returns>
        public static void EnableSerialization(ITokenCache tokenCache, string unifiedCacheFileName, string
adalV3CacheFileName)
        {
            UnifiedCacheFileName = unifiedCacheFileName;
            AdalV3CacheFileName = adalV3CacheFileName;

            tokenCache.SetBeforeAccess(BeforeAccessNotification);
            tokenCache.SetAfterAccess(AfterAccessNotification);
        }

        /// <summary>
        /// File path where the token cache is serialized with the unified cache format
        /// (ADAL.NET V4, MSAL.NET V3)
        /// </summary>
        public static string UnifiedCacheFileName { get; private set; }

        /// <summary>
        /// File path where the token cache is serialized with the legacy ADAL V3 format
        /// </summary>
        public static string AdalV3CacheFileName { get; private set; }

        private static readonly object FileLock = new object();

        public static void BeforeAccessNotification(TokenCacheNotificationArgs args)
        {
            lock (FileLock)

```

```

{
    args.TokenCache.DeserializeAdalV3(ReadFromFileIfExists(AdalV3CacheFileName));
    try
    {
        args.TokenCache.DeserializeMsalV3(ReadFromFileIfExists(UnifiedCacheFileName));
    }
    catch(Exception ex)
    {
        // Compatibility with the MSAL v2 cache if you used one
        args.TokenCache.DeserializeMsalV2(ReadFromFileIfExists(UnifiedCacheFileName));
    }
}
}

public static void AfterAccessNotification(TokenCacheNotificationArgs args)
{
    // if the access operation resulted in a cache update
    if (args.HasStateChanged)
    {
        lock (FileLock)
        {
            WriteToFileIfNotNull(UnifiedCacheFileName, args.TokenCache.SerializeMsalV3());
            if (!string.IsNullOrWhiteSpace(AdalV3CacheFileName))
            {
                WriteToFileIfNotNull(AdalV3CacheFileName, args.TokenCache.SerializeAdalV3());
            }
        }
    }
}

/// <summary>
/// Read the content of a file if it exists
/// </summary>
/// <param name="path">File path</param>
/// <returns>Content of the file (in bytes)</returns>
private static byte[] ReadFromFileIfExists(string path)
{
    byte[] protectedBytes = (!string.IsNullOrEmpty(path) && File.Exists(path))
        ? File.ReadAllBytes(path) : null;
    byte[] unprotectedBytes = encrypt ?
        ((protectedBytes != null) ? ProtectedData.Unprotect(protectedBytes, null,
DataProtectionScope.CurrentUser) : null)
        : protectedBytes;
    return unprotectedBytes;
}

/// <summary>
/// Writes a blob of bytes to a file. If the blob is <c>null</c>, deletes the file
/// </summary>
/// <param name="path">path to the file to write</param>
/// <param name="blob">Blob of bytes to write</param>
private static void WriteToFileIfNotNull(string path, byte[] blob)
{
    if (blob != null)
    {
        byte[] protectedBytes = encrypt
            ? ProtectedData.Protect(blob, null, DataProtectionScope.CurrentUser)
            : blob;
        File.WriteAllBytes(path, protectedBytes);
    }
    else
    {
        File.Delete(path);
    }
}

// Change if you want to test with an un-encrypted blob (this is a json format)
private static bool encrypt = true;
}

```

}

Token cache for a web app (confidential client application)

In web apps or web APIs the cache could leverage the session, a Redis cache, or a database.

In web apps or web APIs, keep one token cache per account. For web apps, the token cache should be keyed by the account ID. For web APIs, the account should be keyed by the hash of the token used to call the API. MSAL.NET provides custom token cache serialization in .NET Framework and .NET Core subplatforms. Events are fired when the cache is accessed, apps can choose whether to serialize or deserialize the cache. On confidential client applications that handle users (web apps that sign in users and call web APIs, and web APIs calling downstream web APIs), there can be many users and the users are processed in parallel. For security and performance reasons, our recommendation is to serialize one cache per user. Serialization events compute a cache key based on the identity of the processed user and serialize/deserialize a token cache for that user.

Examples of how to use token caches for web apps and web APIs are available in the [ASP.NET Core web app tutorial](#) in the phase [2-2 Token Cache](#). For implementations have a look at the folder [TokenCacheProviders](#) in the [microsoft-authentication-extensions-for-dotnet](#) library (in the [Microsoft.Identity.Client.Extensions.Web](#) folder).

Next steps

The following samples illustrate token cache serialization.

| SAMPLE | PLATFORM | DESCRIPTION |
|--|-------------------|---|
| active-directory-dotnet-desktop-msgraph-v2 | Desktop (WPF) | <p>Windows Desktop .NET (WPF) application calling the Microsoft Graph API.</p> <p>The diagram illustrates the flow of a token acquisition process. A local desktop application (labeled 'Desktop App (WPF) TodoListClient') sends a request to 'Acquire token interactively' to 'Azure Active Directory'. This leads to the Microsoft Graph API, which is represented as a cloud icon with a network of nodes. The flow is indicated by arrows labeled '1 Acquire token interactively' and '2 /me'.</p> |
| active-directory-dotnet-v1-to-v2 | Desktop (Console) | <p>Set of Visual Studio solutions illustrating the migration of Azure AD v1.0 applications (using ADAL.NET) to Azure AD v2.0 applications, also named converged applications (using MSAL.NET), in particular Token Cache Migration</p> |

Clear the token cache using MSAL.NET

10/23/2019 • 2 minutes to read • [Edit Online](#)

When you [acquire an access token](#) using Microsoft Authentication Library for .NET (MSAL.NET), the token is cached. When the application needs a token, it should first call the `AcquireTokenSilent` method to verify if an acceptable token is in the cache.

Clearing the cache is achieved by removing the accounts from the cache. This does not remove the session cookie which is in the browser, though. The following example instantiates a public client application, gets the accounts for the application, and removes the accounts.

```
private readonly IPublicClientApplication _app;
private static readonly string ClientId = ConfigurationManager.AppSettings["ida:ClientId"];
private static readonly string Authority = string.Format(CultureInfo.InvariantCulture, AadInstance, Tenant);

_app = PublicClientApplicationBuilder.Create(ClientId)
    .WithAuthority(Authority)
    .Build();

var accounts = (await _app.GetAccountsAsync()).ToList();

// clear the cache
while (accounts.Any())
{
    await _app.RemoveAsync(accounts.First());
    accounts = (await _app.GetAccountsAsync()).ToList();
}
```

To learn more about acquiring and caching tokens, read [acquire an access token](#).

Instantiate a public client application with configuration options using MSAL.NET

10/23/2019 • 2 minutes to read • [Edit Online](#)

This article describes how to instantiate a [public client application](#) using Microsoft Authentication Library for .NET (MSAL.NET). The application is instantiated with configuration options defined in a settings file.

Before initializing an application, you first need to [register](#) it so that your app can be integrated with the Microsoft identity platform. After registration, you may need the following information (which can be found in the Azure portal):

- The client ID (a string representing a GUID)
- The identity provider URL (named the instance) and the sign-in audience for your application. These two parameters are collectively known as the authority.
- The tenant ID if you are writing a line of business application solely for your organization (also named single-tenant application).
- For web apps, and sometimes for public client apps (in particular when your app needs to use a broker), you'll have also set the redirectUri where the identity provider will contact back your application with the security tokens.

A .NET Core console application could have the following *appsettings.json* configuration file:

```
{  
  "Authentication": {  
    "AzureCloudInstance": "AzurePublic",  
    "AadAuthorityAudience": "AzureAdMultipleOrgs",  
    "ClientId": "ebe2ab4d-12b3-4446-8480-5c3828d04c50"  
  },  
  
  "WebAPI": {  
    "MicrosoftGraphBaseEndpoint": "https://graph.microsoft.com"  
  }  
}
```

The following code reads this file using the .NET configuration framework:

```

public class SampleConfiguration
{
    /// <summary>
    /// Authentication options
    /// </summary>
    public PublicClientApplicationOptions PublicClientApplicationOptions { get; set; }

    /// <summary>
    /// Base URL for Microsoft Graph (it varies depending on whether the application is ran
    /// in Microsoft Azure public clouds or national / sovereign clouds
    /// </summary>
    public string MicrosoftGraphBaseEndpoint { get; set; }

    /// <summary>
    /// Reads the configuration from a json file
    /// </summary>
    /// <param name="path">Path to the configuration json file</param>
    /// <returns>SampleConfiguration as read from the json file</returns>
    public static SampleConfiguration ReadFromJsonFile(string path)
    {
        // .NET configuration
        IConfigurationRoot Configuration;
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile(path);
        Configuration = builder.Build();

        // Read the auth and graph endpoint config
        SampleConfiguration config = new SampleConfiguration()
        {
            PublicClientApplicationOptions = new PublicClientApplicationOptions()
        };
        Configuration.Bind("Authentication", config.PublicClientApplicationOptions);
        config.MicrosoftGraphBaseEndpoint = Configuration.GetValue<string>
        ("WebAPI:MicrosoftGraphBaseEndpoint");
        return config;
    }
}

```

The following code creates your application, using the configuration from the settings file:

```

SampleConfiguration config = SampleConfiguration.ReadFromJsonFile("appsettings.json");
var app = PublicClientApplicationBuilder.CreateWithApplicationOptions(config.PublicClientApplicationOptions)
    .Build();

```

Instantiate a confidential client application with configuration options using MSAL.NET

10/23/2019 • 2 minutes to read • [Edit Online](#)

This article describes how to instantiate a [confidential client application](#) using Microsoft Authentication Library for .NET (MSAL.NET). The application is instantiated with configuration options defined in a settings file.

Before initializing an application, you first need to [register](#) it so that your app can be integrated with the Microsoft identity platform. After registration, you may need the following information (which can be found in the Azure portal):

- The client ID (a string representing a GUID)
- The identity provider URL (named the instance) and the sign-in audience for your application. These two parameters are collectively known as the authority.
- The tenant ID if you are writing a line of business application solely for your organization (also named single-tenant application).
- The application secret (client secret string) or certificate (of type X509Certificate2) if it's a confidential client app.
- For web apps, and sometimes for public client apps (in particular when your app needs to use a broker), you'll have also set the redirectUri where the identity provider will contact back your application with the security tokens.

Configure the application from the config file

The name of the properties of the options in MSAL.NET match the name of the properties of the `AzureADOptions` in ASP.NET Core, so you don't need to write any glue code.

An ASP.NET Core application configuration is described in an `appsettings.json` file:

```
{
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "Domain": "[Enter the domain of your tenant, e.g. contoso.onmicrosoft.com]",
    "TenantId": "[Enter 'common', or 'organizations' or the Tenant Id (Obtained from the Azure portal. Select 'Endpoints' from the 'App registrations' blade and use the GUID in any of the URLs), e.g. da41245a5-11b3-996c-00a8-4d99re19f292]",
    "ClientId": "[Enter the Client Id (Application ID obtained from the Azure portal), e.g. ba74781c2-53c2-442a-97c2-3d60re42f403]",
    "CallbackPath": "/signin-oidc",
    "SignedOutCallbackPath": "/signout-callback-oidc",

    "ClientSecret": "[Copy the client secret added to the app from the Azure portal]"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Starting in MSAL.NET v3.x, you can configure your confidential client application from the config file.

In the class where you want configure and instantiate your application, you need to declare a

`ConfidentialClientApplicationOptions` object. Bind the configuration read from the source (including the `appconfig.json` file) to the instance of the application options, using the `IConfigurationRoot.Bind()` method from the [Microsoft.Extensions.Configuration.Binder nuget package](#):

```
using Microsoft.Identity.Client;

private ConfidentialClientApplicationOptions _applicationOptions;
_applicationOptions = new ConfidentialClientApplicationOptions();
configuration.Bind("AzureAD", _applicationOptions);
```

This enables the content of the "AzureAD" section of the `appsettings.json` file to be bound to the corresponding properties of the `ConfidentialClientApplicationOptions` object. Next, build a `ConfidentialClientApplication` object:

```
IConfidentialClientApplication app;
app = ConfidentialClientApplicationBuilder.CreateWithApplicationOptions(_applicationOptions)
    .Build();
```

Add runtime configuration

In a confidential client application, you usually have a cache per user. Therefore you will need to get the cache associated with the user and inform the application builder that you want to use it. In the same way, you might have a dynamically computed redirect URI. In this case the code is the following:

```
IConfidentialClientApplication app;
var request = httpContext.Request;
var currentUri = UriHelper.BuildAbsolute(request.Scheme, request.Host, request.PathBase,
    _azureAdOptions.CallbackPath ?? string.Empty);
app = ConfidentialClientApplicationBuilder.CreateWithApplicationOptions(_applicationOptions)
    .WithRedirectUri(currentUri)
    .Build();
TokenCache userTokenCache = _tokenCacheProvider.SerializeCache(app.UserTokenCache, httpContext,
    claimsPrincipal);
```

User gets consent for several resources using MSAL.NET

10/23/2019 • 2 minutes to read • [Edit Online](#)

The Microsoft identity platform endpoint does not allow you to get a token for several resources at once. When using the Microsoft Authentication Library for .NET (MSAL.NET), the scopes parameter in the acquire token method should only contain scopes for a single resource. However, you can pre-consent to several resources upfront by specifying additional scopes using the `.WithExtraScopeToConsent` builder method.

NOTE

Getting consent for several resources works for Microsoft identity platform, but not for Azure AD B2C. Azure AD B2C supports only admin consent, not user consent.

For example, if you have two resources that have 2 scopes each:

- `https://mytenant.onmicrosoft.com/customerapi` (with 2 scopes `customer.read` and `customer.write`)
- `https://mytenant.onmicrosoft.com/vendorapi` (with 2 scopes `vendor.read` and `vendor.write`)

You should use the `.WithExtraScopeToConsent` modifier which has the `extraScopesToConsent` parameter as shown in the following example:

```
string[] scopesForCustomerApi = new string[]
{
    "https://mytenant.onmicrosoft.com/customerapi/customer.read",
    "https://mytenant.onmicrosoft.com/customerapi/customer.write"
};
string[] scopesForVendorApi = new string[]
{
    "https://mytenant.onmicrosoft.com/vendorapi/vendor.read",
    "https://mytenant.onmicrosoft.com/vendorapi/vendor.write"
};

var accounts = await app.GetAccountsAsync();
var result = await app.AcquireTokenInteractive(scopesForCustomerApi)
    .WithAccount(accounts.FirstOrDefault())
    .WithExtraScopeToConsent(scopesForVendorApi)
    .ExecuteAsync();
```

This will get you an access token for the first web API. Then, when you need to access the second web API you can silently acquire the token from the token cache:

```
AcquireTokenSilent(scopesForVendorApi, accounts.FirstOrDefault()).ExecuteAsync();
```

Providing your own HttpClient and proxy using MSAL.NET

10/23/2019 • 2 minutes to read • [Edit Online](#)

When [initializing a public client application](#), you can use the `.WithHttpClientFactory` method to provide your own `HttpClient`. Providing your own `HttpClient` enables advanced scenarios such fine-grained control of an HTTP proxy, customizing user agent headers, or forcing MSAL to use a specific `HttpClient` (for example in ASP.NET Core web apps/APIs).

Initialize with `HttpClientFactory`

The following example shows to create an `HttpClientFactory` and then initialize a public client application with it:

```
IMsalHttpClientFactory httpClientFactory = new MyHttpClientFactory();

var pca = PublicClientApplicationBuilder.Create(MsalTestConstants.ClientId)
    .WithHttpClientFactory(httpClientFactory)
    .Build();
```

HttpClient and Xamarin iOS

When using Xamarin iOS, it is recommended to create an `HttpClient` that explicitly uses the `NSURLSession`-based handler for iOS 7 and newer. MSAL.NET automatically creates an `HttpClient` that uses `NSURLSessionHandler` for iOS 7 and newer. For more information, read the [Xamarin iOS documentation for `HttpClient`](#).

Avoid page reloads when acquiring and renewing tokens silently using MSAL.js

10/23/2019 • 2 minutes to read • [Edit Online](#)

Microsoft Authentication Library for JavaScript (MSAL.js) uses hidden `iframe` elements to acquire and renew tokens silently in the background. Azure AD returns the token back to the registered `redirect_uri` specified in the token request (by default this is the app's root page). Since the response is a 302, it results in the HTML corresponding to the `redirect_uri` getting loaded in the `iframe`. Usually the app's `redirect_uri` is the root page and this causes it to reload.

In other cases, if navigating to the app's root page requires authentication, it might lead to nested `iframe` elements or `X-Frame-Options: deny` error.

Since MSAL.js cannot dismiss the 302 issued by Azure AD and is required to process the returned token, it cannot prevent the `redirect_uri` from getting loaded in the `iframe`.

To avoid the entire app reloading again or other errors caused due to this, please follow these workarounds.

Specify different HTML for the iframe

Set the `redirect_uri` property on config to a simple page, that does not require authentication. You have to make sure that it matches with the `redirect_uri` registered in Azure portal. This will not affect user's login experience as MSAL saves the start page when user begins the login process and redirects back to the exact location after login is completed.

Initialization in your main app file

If your app is structured such that there is one central Javascript file that defines the app's initialization, routing, and other stuff, you can conditionally load your app modules based on whether the app is loading in an `iframe` or not. For example:

In AngularJS: app.js

```

// Check that the window is an iframe and not popup
if (window !== window.parent && !window.opener) {
angular.module('todoApp', ['ui.router', 'MsalAngular'])
    .config(['$httpProvider', 'msalAuthenticationServiceProvider','$locationProvider', function ($httpProvider,
msalProvider,$locationProvider) {
        msalProvider.init(
            // msal configuration
        );
        $locationProvider.html5Mode(false).hashPrefix('');
    }]);
}
else {
    angular.module('todoApp', ['ui.router', 'MsalAngular'])
        .config(['$stateProvider', '$httpProvider', 'msalAuthenticationServiceProvider', '$locationProvider',
function ($stateProvider, $httpProvider, msalProvider, $locationProvider) {
        $stateProvider.state("Home", {
            url: '/Home',
            controller: "homeCtrl",
            templateUrl: "/App/Views/Home.html",
        }).state("TodoList", {
            url: '/TodoList',
            controller: "todoListCtrl",
            templateUrl: "/App/Views/TodoList.html",
            requireLogin: true
        });
        $locationProvider.html5Mode(false).hashPrefix('');
        msalProvider.init(
            // msal configuration
        );
    }]);
}

```

In Angular: app.module.ts

```

// Imports...
@NgModule({
  declarations: [
    AppComponent,
    MsalComponent,
    MainMenuComponent,
    AccountMenuComponent,
    OsNavComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    ServiceWorkerModule.register('ngsw-worker.js', { enabled: environment.production }),
    MsalModule.forRoot(environment.MsalConfig),
    SuiModule,
    PagesModule
  ],
  providers: [
    HttpServiceHelper,
    {provide: HTTP_INTERCEPTORS, useClass: MsalInterceptor, multi: true},
    AuthService
  ],
  entryComponents: [
    AppComponent,
    MsalComponent
  ]
})
export class AppModule {
  constructor() {
    console.log('APP Module Constructor!');
  }

  ngDoBootstrap(ref: ApplicationRef) {
    if (window !== window.parent && !window.opener)
    {
      console.log("Bootstrap: MSAL");
      ref.bootstrap(MsalComponent);
    }
    else
    {
      //this.router.resetConfig(RouterModule);
      console.log("Bootstrap: App");
      ref.bootstrap(AppComponent);
    }
  }
}

```

MsalComponent:

```

import { Component } from '@angular/core';
import { MsalService } from '@azure/msal-angular';

// This component is used only to avoid Angular reload
// when doing acquireTokenSilent()

@Component({
  selector: 'app-root',
  template: '',
})

export class MsalComponent {
  constructor(private Msal: MsalService) {
  }
}

```

Next steps

Learn more about building a single-page application (SPA) using MSAL.js.

Pass custom state in authentication requests using MSAL.js

10/30/2019 • 2 minutes to read • [Edit Online](#)

The `state` parameter, as defined by OAuth 2.0, is included in an authentication request and is also returned in the token response to prevent cross-site request forgery attacks. By default, Microsoft Authentication Library for JavaScript (MSAL.js) passes a randomly generated unique `state` parameter value in the authentication requests.

The `state` parameter can also be used to encode information of the app's state before redirect. You can pass the user's state in the app, such as the page or view they were on, as input to this parameter. The MSAL.js library allows you to pass your custom state as state parameter in the `Request` object:

```
// Request type
export type AuthenticationParameters = {
    scopes?: Array<string>;
    extraScopesToConsent?: Array<string>;
    prompt?: string;
    extraQueryParameters?: QPDict;
    claimsRequest?: string;
    authority?: string;
    state?: string;
    correlationId?: string;
    account?: Account;
    sid?: string;
    loginHint?: string;
};
```

For example:

```
let loginRequest = {
    scopes: ["user.read", "user.write"],
    state: "page_url"
}

myMSALObj.loginPopup(loginRequest);
```

The passed in state is appended to the unique GUID set by MSAL.js when sending the request. When the response is returned, MSAL.js checks for a state match and then returns the custom passed in state in the `Response` object as `accountState`.

```
export type AuthResponse = {
    uniqueId: string;
    tenantId: string;
    tokenType: string;
    idToken: IdToken;
    accessToken: string;
    scopes: Array<string>;
    expiresOn: Date;
    account: Account;
    accountState: string;
};
```

To learn more, read about [building a single-page application \(SPA\)](#) using MSAL.js.

Prompt behavior in MSAL.js interactive requests

10/23/2019 • 2 minutes to read • [Edit Online](#)

When a user has established an active Azure AD session with multiple user accounts, the Azure AD sign in page will by default prompt the user to select an account before proceeding to sign in. Users will not see an account selection experience if there is only a single authenticated session with Azure AD.

The MSAL.js library (starting in v0.2.4) does not send a prompt parameter during the interactive requests (`loginRedirect`, `loginPopup`, `acquireTokenRedirect` and `acquireTokenPopup`), and thereby does not enforce any prompt behavior. For silent token requests using the `acquireTokenSilent` method, MSAL.js passes a prompt parameter set to `none`.

Based on your application scenario, you can control the prompt behavior for the interactive requests by setting the prompt parameter in the request parameters passed to the methods. For example, if you want to invoke the account selection experience:

```
var request = {
  scopes: ["user.read"],
  prompt: 'select_account',
}

userAgentApplication.loginRedirect(request);
```

The following prompt values can be passed when authenticating with Azure AD:

login: This value will force the user to enter credentials on the authentication request.

select_account: This value will provide the user with an account selection experience listing all the accounts in session.

consent: This value will invoke the OAuth consent dialogue that allows users to grant permissions to the app.

none: This value will ensure that the user does not see any interactive prompt. It is recommended not to pass this value to interactive methods in MSAL.js as it can have unexpected behaviors. Instead, use the `acquireTokenSilent` method to achieve silent calls.

Next steps

Read more about the `prompt` parameter in the [OAuth 2.0 implicit grant](#) protocol which MSAL.js library uses.

Microsoft Authentication Library for iOS and macOS differences

10/30/2019 • 2 minutes to read • [Edit Online](#)

This article explains the differences in functionality between the Microsoft Authentication Library (MSAL) for iOS and macOS.

NOTE

On the Mac, MSAL only supports macOS apps.

General differences

MSAL for macOS is a subset of the functionality available for iOS.

MSAL for macOS doesn't support:

- different browser types such as `ASWebAuthenticationSession`, `SFAuthenticationSession`, `SFSafariViewController`.
- brokered authentication through Microsoft Authenticator app is not supported for macOS.

Keychain sharing between apps from the same publisher is more limited on macOS 10.14 and earlier. Use [access control lists](#) to specify the paths to the apps that should share the keychain. User may see additional keychain prompts.

On macOS 10.15+, MSAL's behavior is the same between iOS and macOS. MSAL uses [keychain access groups](#) for keychain sharing.

Conditional access authentication differences

For Conditional Access scenarios, there will be fewer user prompts when you use MSAL for iOS. This is because iOS uses the broker app (Microsoft Authenticator) which negates the need to prompt the user in some cases.

Project setup differences

macOS

- When you set up your project on macOS, ensure that your application is signed with a valid development or production certificate. MSAL still works in the unsigned mode, but it will behave differently with regards to cache persistence. The app should only be run unsigned for debugging purposes. If you distribute the app unsigned, it will:
 1. On 10.14 and earlier, MSAL will prompt the user for a keychain password every time they restart the app.
 2. On 10.15+, MSAL will prompt user for credentials for every token acquisition.
- macOS apps don't need to implement the AppDelegate call.

iOS

- There are additional steps to set up your project to support authentication broker flow. The steps are called out in the tutorial.
- iOS projects need to register custom schemes in the info.plist. This isn't required on macOS.

Configure keychain

10/23/2019 • 2 minutes to read • [Edit Online](#)

When the [Microsoft Authentication Library for iOS and macOS](#) (MSAL) signs in a user, or refreshes a token, it tries to cache tokens in the keychain. Caching tokens in the keychain allows MSAL to provide silent single sign-on (SSO) between multiple apps that are distributed by the same Apple developer. SSO is achieved via the keychain access groups functionality. For more information, see Apple's [Keychain Items documentation](#).

This article covers how to configure app entitlements so that MSAL can write cached tokens to iOS and macOS keychain.

Default keychain access group

iOS

MSAL on iOS uses the `com.microsoft.adalcache` access group by default. This is the shared access group used by both MSAL and Azure AD Authentication Library (ADAL) SDKs and ensures the best single sign-on (SSO) experience between multiple apps from the same publisher.

On iOS, add the `com.microsoft.adalcache` keychain group to your app's entitlement in XCode under **Project settings > Capabilities > Keychain sharing**

macOS

MSAL on macOS uses `com.microsoft.identity.universalstorage` access group by default.

Due to macOS keychain limitations, MSAL's `access group` doesn't directly translate to the keychain access group attribute (see [kSecAttrAccessGroup](#)) on macOS 10.14 and earlier. However, it behaves similarly from a SSO perspective by ensuring that multiple applications distributed by the same Apple developer can have silent SSO.

On macOS 10.15 onwards (macOS Catalina), MSAL uses keychain access group attribute to achieve silent SSO, similarly to iOS.

Custom keychain access group

If you'd like to use a different keychain access group, you can pass your custom group when creating `MSALPublicClientApplicationConfig` before creating `MSALPublicClientApplication`, like this:

Objective-C:

```
MSALPublicClientApplicationConfig *config = [[MSALPublicClientApplicationConfig alloc]
initWithClientId:@"your-client-id"

redirectUri:@"your-redirect-uri"
           authority:nil];

config.cacheConfig.keychainSharingGroup = @"custom-group";

MSALPublicClientApplication *application = [[MSALPublicClientApplication alloc] initWithConfiguration:config
error:nil];

// Now call acquiretoken.
// Tokens will be saved into the "custom-group" access group
// and only shared with other applications declaring the same access group
```

Swift:

```
let config = MSALPublicClientApplicationConfig(clientId: "your-client-id",
                                                redirectUri: "your-redirect-uri",
                                                authority: nil)
config.cacheConfig.keychainSharingGroup = "custom-group"

do {
    let application = try MSALPublicClientApplication(configuration: config)
    // continue on with application
} catch let error as NSError {
    // handle error here
}
```

Disable keychain sharing

If you don't want to share SSO state between multiple apps, or use any keychain access group, disable keychain sharing by passing the application bundle ID as your keychainGroup:

Objective-C:

```
config.cacheConfig.keychainSharingGroup = [[NSBundle mainBundle] bundleIdentifier];
```

Swift:

```
if let bundleIdentifier = Bundle.main.bundleIdentifier {
    config.cacheConfig.keychainSharingGroup = bundleIdentifier
}
```

Handle -34018 error (failed to set item into keychain)

Error -34018 normally means that the keychain hasn't been configured correctly. Ensure the keychain access group that has been configured in MSAL matches the one configured in entitlements.

Ensure your application is properly signed

On macOS, applications can execute without being signed by developer. While most of MSAL's functionality will continue to work, SSO through keychain access requires application to be signed. If you're experiencing multiple keychain prompts, make sure your application's signature is valid.

Next steps

Learn more about keychain access groups in Apple's [Sharing Access to Keychain Items Among a Collection of Apps](#) article.

How to: Customize browsers and WebViews for iOS/macOS

10/23/2019 • 3 minutes to read • [Edit Online](#)

A web browser is required for interactive authentication. On iOS, the Microsoft Authentication Library (MSAL) uses the system web browser by default (which might appear on top of your app) to do interactive authentication to sign in users. Using the system browser has the advantage of sharing the Single Sign ON (SSO) state with other applications and with web applications.

You can change the experience by customizing the configuration to other options for displaying web content, such as:

For iOS only:

- [ASWebAuthenticationSession](#)
- [SFAuthenticationSession](#)
- [SFSafariViewController](#)

For iOS and macOS:

- [WKWebView](#).

MSAL for macOS only supports [WKWebView](#).

System browsers

For iOS, [ASWebAuthenticationSession](#), [SFAuthenticationSession](#), and [SFSafariViewController](#) are considered system browsers. In general, system browsers share cookies and other website data with the Safari browser application.

By default, MSAL will dynamically detect iOS version and select the recommended system browser available on that version. On iOS 12+ it will be [ASWebAuthenticationSession](#).

| VERSION | WEB BROWSER |
|---------|----------------------------|
| iOS 12+ | ASWebAuthenticationSession |
| iOS 11 | SFAuthenticationSession |
| iOS 10 | SFSafariViewController |

Developers can also select a different system browser for MSAL apps:

- [SFAuthenticationSession](#) is the iOS 11 version of [ASWebAuthenticationSession](#).
- [SFSafariViewController](#) is more general purpose and provides an interface for browsing the web and can be used for login purposes as well. In iOS 9 and 10, cookies and other website data are shared with Safari--but not in iOS 11 and later.

In-app browser

[WKWebView](#) is an in-app browser that displays web content. It doesn't share cookies or web site data with other

WKWebView instances, or with the Safari browser. WKWebView is a cross-platform browser that is available for both iOS and macOS.

Cookie sharing and Single sign-on (SSO) implications

The browser you use impacts the SSO experience because of how they share cookies. The following tables summarize the SSO experiences per browser.

| TECHNOLOGY | BROWSER TYPE | IOS AVAILABILITY | MACOS AVAILABILITY | SHARES COOKIES AND OTHER DATA | MSAL AVAILABILITY | SSO |
|--|--------------|------------------|--------------------|-------------------------------|-------------------|---------------------|
| ASWebAuthenticationSession | System | iOS12 and up | macOS 10.15 and up | Yes | iOS only | w/ Safari instances |
| SFAuthenticationSession | System | iOS11 and up | N/A | Yes | iOS only | w/ Safari instances |
| SFSafariViewController | System | iOS11 and up | N/A | No | iOS only | No** |
| SFSafariView Controller | System | iOS10 | N/A | Yes | iOS only | w/ Safari instances |
| WKWebView | In-app | iOS8 and up | macOS 10.10 and up | No | iOS and macOS | No** |

** For SSO to work, tokens need to be shared between apps. This requires a token cache, or broker application, such as Microsoft Authenticator for iOS.

Change the default browser for the request

You can use an in-app browser, or a specific system browser depending on your UX requirements, by changing the following property in `MSALWebviewParameters`:

```
@property (nonatomic) MSALWebviewType webviewType;
```

Change per interactive request

Each request can be configured to override the default browser by changing the `MSALInteractiveTokenParameters.webviewParameters.webviewType` property before passing it to the `acquireTokenWithParameters:completionBlock:` API.

Additionally, MSAL supports passing in a custom `WKWebView` by setting the `MSALInteractiveTokenParameters.webviewParameters.customWebView` property.

For example:

Objective-C

```

UIViewController *myParentController = ...;
WKWebView *myCustomWebView = ...;
MSALWebviewParameters *webViewParameters = [[MSALWebviewParameters alloc]
initWithParentViewController:myParentController];
webViewParameters.webviewType = MSALWebviewTypeWKWebView;
webViewParameters.customWebview = myCustomWebView;
MSALInteractiveTokenParameters *interactiveParameters = [[MSALInteractiveTokenParameters alloc]
initWithScopes:@[@"myscope"] webviewParameters:webViewParameters];

[app acquireTokenWithParameters:interactiveParameters completionBlock:completionBlock];

```

Swift

```

let myParentController: UIViewController = ...
let myCustomWebView: WKWebView = ...
let webViewParameters = MSALWebviewParameters(parentViewController: myParentController)
webViewParameters.webviewType = MSALWebviewType.wkWebView
webViewParameters.customWebview = myCustomWebView
let interactiveParameters = MSALInteractiveTokenParameters(scopes: ["myscope"], webviewParameters:
webViewParameters)

app.acquireToken(with: interactiveParameters, completionBlock: completionBlock)

```

If you use a custom webview, notifications are used to indicate the status of the web content being displayed, such as:

```

/*! Fired at the start of a resource load in the webview. The URL of the load, if available, will be in the
@"url" key in the userInfo dictionary */
extern NSString *MSALWebAuthDidStartLoadNotification;

/*! Fired when a resource finishes loading in the webview. */
extern NSString *MSALWebAuthDidFinishLoadNotification;

/*! Fired when web authentication fails due to reasons originating from the network. Look at the @"error" key
in the userInfo dictionary for more details.*/
extern NSString *MSALWebAuthDidFailNotification;

/*! Fired when authentication finishes */
extern NSString *MSALWebAuthDidCompleteNotification;

/*! Fired before ADAL invokes the broker app */
extern NSString *MSALWebAuthWillSwitchToBrokerApp;

```

Options

All MSAL supported web browser types are declared in the [MSALWebviewType](#) enum

```
typedef NS_ENUM(NSInteger, MSALWebviewType)
{
#if TARGET_OS_IPHONE
    // For iOS 11 and up, uses AuthenticationSession (ASWebAuthenticationSession
    // or SFAuthenticationSession).
    // For older versions, with AuthenticationSession not being available, uses
    // SafariViewController.
    MSALWebviewTypeDefault,
    // Use SFAuthenticationSession/ASWebAuthenticationSession
    MSALWebviewTypeAuthenticationSession,
    // Use SFSafariViewController for all versions.
    MSALWebviewTypeSafariViewController,
#endif
    // Use WKWebView
    MSALWebviewTypeWKWebView,
};

};
```

Next steps

Learn more about [Authentication flows and application scenarios](#)

How to: Request custom claims using MSAL for iOS and macOS

10/23/2019 • 2 minutes to read • [Edit Online](#)

OpenID Connect allows you to optionally request the return of individual claims from the UserInfo Endpoint and/or in the ID Token. A claims request is represented as a JSON object that contains a list of requested claims. See [OpenID Connect Core 1.0](#) for more details.

The Microsoft Authentication Library (MSAL) for iOS and macOS allows requesting specific claims in both interactive and silent token acquisition scenarios. It does so through the `claimsRequest` parameter.

There are multiple scenarios where this is needed. For example:

- Requesting claims outside of the standard set for your application.
- Requesting specific combinations of the standard claims that cannot be specified using scopes for your application. For example, if an access token gets rejected because of missing claims, the application can request the missing claims using MSAL.

NOTE

MSAL bypasses the access token cache whenever a claims request is specified. It's important to only provide `claimsRequest` parameter when additional claims are needed (as opposed to always providing same `claimsRequest` parameter in each MSAL API call).

`claimsRequest` can be specified in `MSALSilentTokenParameters` and `MSALInteractiveTokenParameters`:

```
/*
MSALTokenParameters is the base abstract class for all types of token parameters (silent and interactive).
*/
@interface MSALTokenParameters : NSObject

/*
The claims parameter that needs to be sent to authorization or token endpoint.
If claims parameter is passed in silent flow, access token will be skipped and refresh token will be tried.
*/
@property (nonatomic, nullable) MSALClaimsRequest *claimsRequest;

@end
```

`MSALClaimsRequest` can be constructed from an `NSString` representation of JSON Claims request.

Objective-C:

```
NSError *claimsError = nil;
MSALClaimsRequest *request = [[MSALClaimsRequest alloc] initWithJsonString:@"{\"id_token\":{\"auth_time\":
{\"essential\":true},\"acr\":{\"values\":[\"urn:mace:incommon:iap:silver\"]}}}" error:&claimsError];
```

Swift:

```

var requestError: NSError? = nil
let request = MSALClaimsRequest(jsonString: "{\"id_token\":{\"auth_time\":{\"essential\":true},\"acr\":\"urn:mace:incommon:iap:silver\"}}",
                                error: &requestError)

```

It can also be modified by requesting additional specific claims:

Objective-C:

```

MSALIndividualClaimRequest *individualClaimRequest = [[MSALIndividualClaimRequest alloc]
initWithName:@"custom_claim"];
individualClaimRequest.additionalInfo = [MSALIndividualClaimRequestAdditionalInfo new];
individualClaimRequest.additionalInfo.essential = @1;
individualClaimRequest.additionalInfo.value = @"myvalue";
[request requestClaim:individualClaimRequest forTarget:MSALClaimsRequestTargetIdToken error:&claimsError];

```

Swift:

```

let individualClaimRequest = MSALIndividualClaimRequest(name: "custom-claim")
let additionalInfo = MSALIndividualClaimRequestAdditionalInfo()
additionalInfo.essential = 1
additionalInfo.value = "myvalue"
individualClaimRequest.additionalInfo = additionalInfo
do {
    try request.requestClaim(individualClaimRequest, for: .idToken)
} catch let error as NSError {
    // handle error here
}

```

`MSALClaimsRequest` should be then set in the token parameters and provided to one of MSAL token acquisitions APIs:

Objective-C:

```

MSALPublicClientApplication *application = ...
MSALWebviewParameters *webParameters = ...

MSALInteractiveTokenParameters *parameters = [[MSALInteractiveTokenParameters alloc]
initWithScopes:@[@"user.read"]

webviewParameters:webParameters];
parameters.claimsRequest = request;

[application acquireTokenWithParameters:parameters completionBlock:completionBlock];

```

Swift:

```

let application: MSALPublicClientApplication!
let webParameters: MSALWebviewParameters!

let parameters = MSALInteractiveTokenParameters(scopes: ["user.read"], webviewParameters: webParameters)
parameters.claimsRequest = request

application.acquireToken(with: parameters) { (result: MSALResult?, error: Error?) in
    ...
}

```

Next steps

Learn more about [Authentication flows and application scenarios](#)

Using redirect URLs with the Microsoft authentication library for iOS and macOS

10/23/2019 • 3 minutes to read • [Edit Online](#)

When a user authenticates, Azure Active Directory (Azure AD) sends the token to the app by using the redirect URI registered with the Azure AD application.

The Microsoft Authentication library (MSAL) requires that the redirect URI be registered with the Azure AD app in a specific format. MSAL uses a default redirect URI, if you don't specify one. The format is

`msauth.[Your_Bundle_Id]://auth`.

The default redirect URI format works for most apps and scenarios, including brokered authentication and system web view. Use the default format whenever possible.

However, you may need to change the redirect URI for advanced scenarios, as described below.

Scenarios that require a different redirect URI

Cross-app single sign on (SSO)

For the Microsoft Identity platform to share tokens across apps, each app needs to have the same client ID or application ID. This is the unique identifier provided when you registered your app in the portal (not the application bundle ID that you register per app with Apple).

The redirect URIs need to be different for each iOS app. This allows the Microsoft identity service to uniquely identify different apps that share an application ID. Each application can have multiple redirect URIs registered in the Azure portal. Each app in your suite will have a different redirect URI. For example:

Given the following application registration in the Azure portal:

```
Client ID: ABCDE-12345 (this is a single client ID)
RedirectUris: msauth.com.contoso.app1://auth, msauth.com.contoso.app2://auth, msauth.com.contoso.app3://auth
```

App1 uses redirect `msauth.com.contoso.app1://auth` App2 uses `msauth.com.contoso.app2://auth` App3 uses
`msauth.com.contoso.app3://auth`

Migrating from ADAL to MSAL

When migrating code that used the Azure AD Authentication Library (ADAL) to MSAL, you may already have a redirect URI configured for your app. You can continue using the same redirect URI as long as your ADAL app was configured to support brokered scenarios and your redirect URI satisfies the MSAL redirect URI format requirements.

MSAL redirect URI format requirements

- The MSAL redirect URI must be in the form `<scheme>://host`

Where `<scheme>` is a unique string that identifies your app. It's primarily based on the Bundle Identifier of your application to guarantee uniqueness. For example, if your app's Bundle ID is `com.contoso.myapp`, your redirect URI would be in the form: `msauth.com.contoso.myapp://auth`.

If you're migrating from ADAL, your redirect URI will likely have this format: `<scheme>://[Your_Bundle_Id]`,

where `scheme` is a unique string. This format will continue to work when you use MSAL.

- `<scheme>` must be registered in your app's Info.plist under `CFBundleURLTypes > CFBundleURLSchemes`. In this example, Info.plist has been opened as source code:

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>msauth.[BUNDLE_ID]</string>
        </array>
    </dict>
</array>
```

MSAL will verify if your redirect URI registers correctly, and return an error if it's not.

- If you want to use universal links as a redirect URI, the `<scheme>` must be `https` and doesn't need to be declared in `CFBundleURLSchemes`. Instead, configure the app and domain per Apple's instructions at [Universal Links for Developers](#) and call the `handleMSALResponse:sourceApplication:` method of `MSALPublicClientApplication` when your application is opened through a universal link.

Use a custom redirect URI

To use a custom redirect URI, pass the `redirectUri` parameter to `MSALPublicClientApplicationConfig` and pass that object to `MSALPublicClientApplication` when you initialize the object. If the redirect URI is invalid, the initializer will return `nil` and set the `redirectURIError` with additional information. For example:

Objective-C:

```
MSALPublicClientApplicationConfig *config =
[[MSALPublicClientApplicationConfig alloc] initWithClientId:@"your-client-id"
                                              redirectUri:@"your-redirect-uri"
                                              authority:authority];
NSError *redirectURIError;
MSALPublicClientApplication *application =
[[MSALPublicClientApplication alloc] initWithConfiguration:config error:&redirectURIError];
```

Swift:

```
let config = MSALPublicClientApplicationConfig(clientId: "your-client-id",
                                                redirectUri: "your-redirect-uri",
                                                authority: authority)
do {
    let application = try MSALPublicClientApplication(configuration: config)
    // continue on with application
} catch let error as NSError {
    // handle error here
}
```

Handle the URL opened event

Your application should call MSAL when it receives any response through URL schemes or universal links. Call the `handleMSALResponse:sourceApplication:` method of `MSALPublicClientApplication` when your application is opened.

Here's an example for custom schemes:

Objective-C:

```
- (BOOL)application:(UIApplication *)app
    openURL:(NSURL *)url
    options:(NSDictionary<UIApplicationOpenURLOptionsKey,id> *)options
{
    return [MSALPublicClientApplication handleMSALResponse:url
sourceApplication:options[UIApplicationOpenURLOptionsSourceApplicationKey]];
}
```

Swift:

```
func application(_ app: UIApplication, open url: URL, options: [UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
    return MSALPublicClientApplication.handleMSALResponse(url, sourceApplication:
options[UIApplication.OpenURLOptionsKey.sourceApplication] as? String)
}
```

Next steps

Learn more about [Authentication flows and application scenarios](#)

Custom token cache serialization in MSAL for Java (MSAL4J)

11/10/2019 • 2 minutes to read • [Edit Online](#)

To have a persistent token cache application, you will need to customize the serialization. The Java classes and interfaces involved in token cache serialization are the following:

- [ITokenCache](#): Interface representing security token cache.
- [ITokenCacheAccessAspect](#): Interface representing operation of executing code before and after access. You would @Override *beforeCacheAccess* and *afterCacheAccess* with the logic responsible for serializing and deserializing the cache.
- [ITokenCacheContext](#): Interface representing context in which the token cache is accessed.

Below is a naive implementation of custom serialization of token cache serialization/deserialization. Do not copy and paste this into a production environment.

```
static class TokenPersistence implements ITokenCacheAccessAspect {
    String data;

    TokenPersistence(String data) {
        this.data = data;
    }

    @Override
    public void beforeCacheAccess(ITokenCacheAccessContext iTokenCacheAccessContext) {
        iTokenCacheAccessContext.tokenCache().deserialize(data);
    }

    @Override
    public void afterCacheAccess(ITokenCacheAccessContext iTokenCacheAccessContext) {
        data = iTokenCacheAccessContext.tokenCache().serialize();
    }
}
```

```
// Loads cache from file
String dataToInitCache = readResource(this.getClass(), "/cache_data/serialized_cache.json");

ITokenCacheAccessAspect persistenceAspect = new TokenPersistence(dataToInitCache);

// By setting *TokenPersistence* on the PublicClientApplication, MSAL will call *beforeCacheAccess()* before
// accessing the cache and *afterCacheAccess()* after accessing the cache.
PublicClientApplication app =
    PublicClientApplication.builder("my_client_id").setTokenCacheAccessAspect(persistenceAspect).build();
```

Learn more

Learn about [Get and remove accounts from the token cache using MSAL for Java](#).

Get and remove accounts from the token cache using MSAL for Java (MSAL4j)

11/10/2019 • 2 minutes to read • [Edit Online](#)

MSAL4J provides an in-memory token cache by default. The in-memory token cache lasts the duration of the application instance.

See which accounts are in the cache

You can check what accounts are in the cache by calling `PublicClientApplication.getAccounts()` as shown in the following example:

```
PublicClientApplication pca = new PublicClientApplication.Builder(  
    labResponse.getAppId()).  
    authority(TestConstants.ORGANIZATIONS_AUTHORITY).  
    build();  
  
Set<IAccount> accounts = pca.getAccounts().join();
```

Remove accounts from the cache

To remove an account from the cache, find the account that needs to be removed and then call

`PublicClientApplicatioin.removeAccount()` as shown in the following example:

```
Set<IAccount> accounts = pca.getAccounts().join();  
  
IAccount accountToBeRemoved = accounts.stream().filter(  
    x -> x.username().equalsIgnoreCase(  
        UPN_OF_USER_TO_BE_REMOVED)).findFirst().orElse(null);  
  
pca.removeAccount(accountToBeRemoved).join();
```

Learn more

If you are using MSAL for Java, learn about [Custom token cache serialization in MSAL for Java](#).

Adding an Azure Active Directory by using Connected Services in Visual Studio

11/12/2019 • 2 minutes to read • [Edit Online](#)

By using Azure Active Directory (Azure AD), you can support Single Sign-On (SSO) for ASP.NET MVC web applications, or Active Directory Authentication in Web API services. With Azure AD Authentication, your users can use their accounts from Azure Active Directory to connect to your web applications. The advantages of Azure AD Authentication with Web API include enhanced data security when exposing an API from a web application. With Azure AD, you do not have to manage a separate authentication system with its own account and user management.

This article and its companion articles provide details of using the Visual Studio Connected Service feature for Active Directory. The capability is available in Visual Studio 2015 and later.

At present, the Active Directory connected service does not support ASP.NET Core applications.

Prerequisites

- Azure account: if you don't have an Azure account, you can [sign up for a free trial](#) or [activate your Visual Studio subscriber benefits](#).
- **Visual Studio 2015** or later. [Download Visual Studio now](#).

Connect to Azure Active Directory using the Connected Services dialog

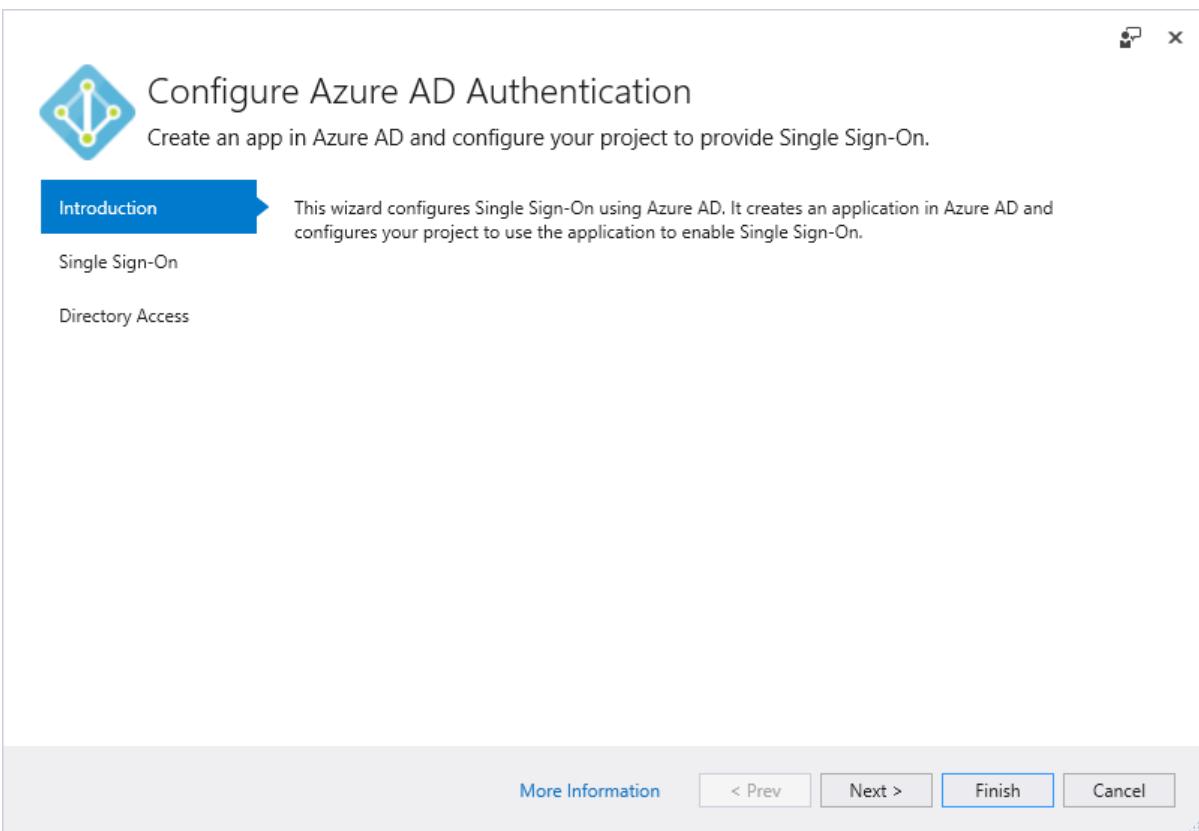
1. In Visual Studio, create or open an ASP.NET MVC project, or an ASP.NET Web API project. You can use the MVC, Web API, Single-Page Application, Azure API App, Azure Mobile App, and Azure Mobile Service templates.
2. Select the **Project > Add Connected Service...** menu command, or double-click the **Connected Services** node found under the project in Solution Explorer.
3. On the **Connected Services** page, select **Authentication with Azure Active Directory**.

Connected Services

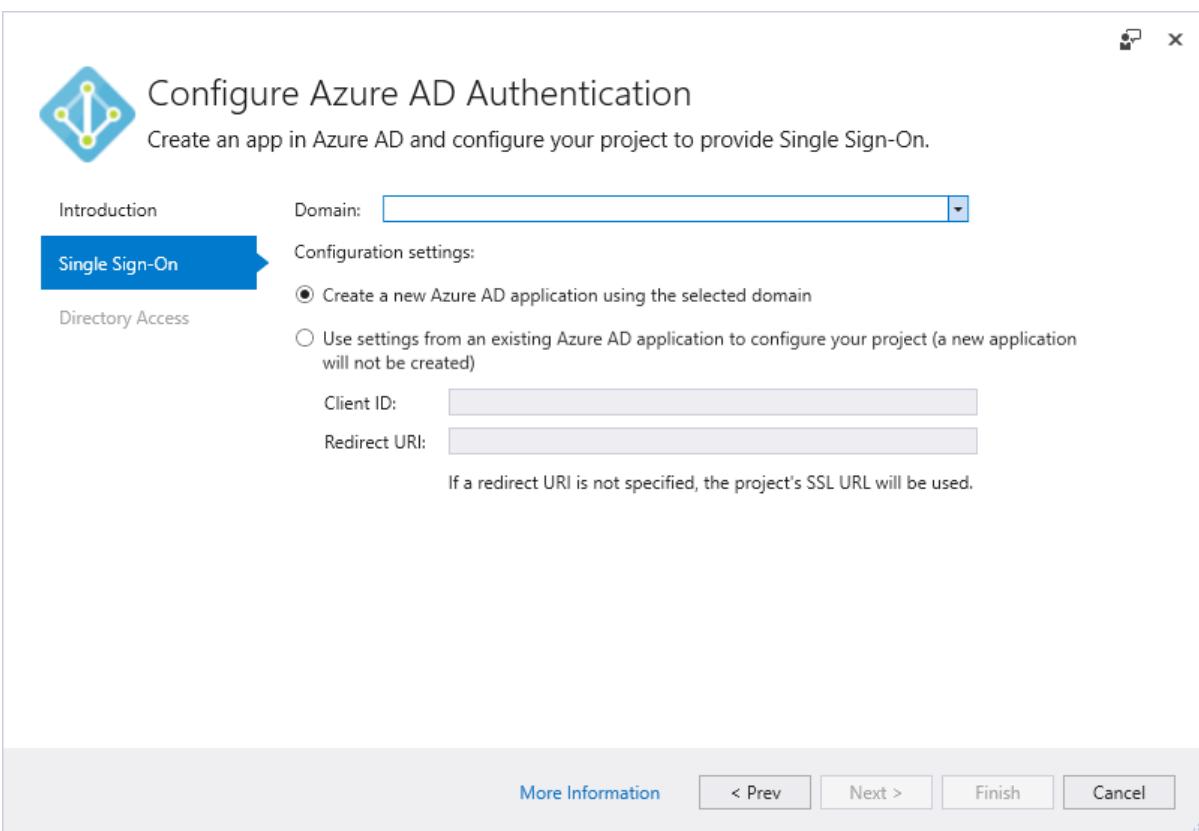
Add code and dependencies for one of these services to your application

-  **Cloud Storage with Azure Storage**
Store and access data with Azure Storage services like Blobs, Queues, and Tables.
-  **Access Office 365 Services with Microsoft Graph**
Use the Microsoft Graph API to integrate your applications with Office 365 services.
-  **Authentication with Azure Active Directory**
Configure Single Sign-On in your application using Azure AD.

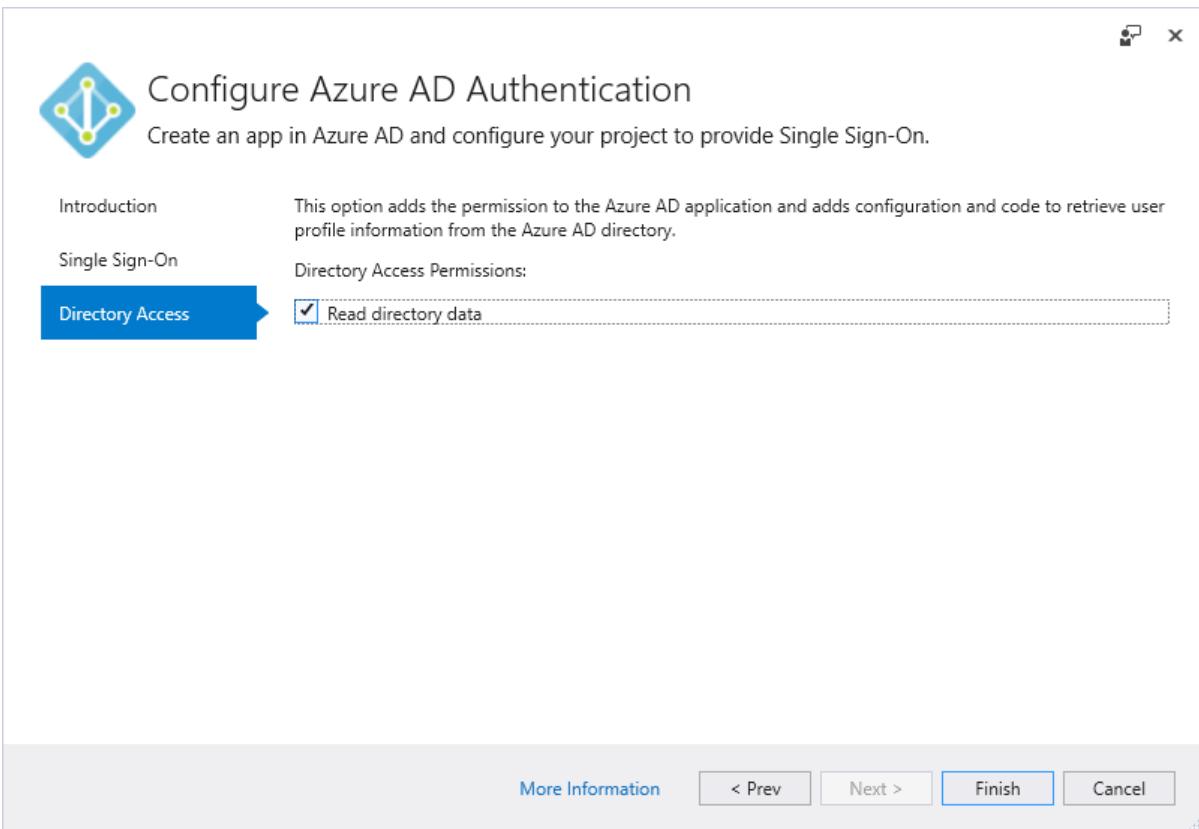
4. On the **Introduction** page, select **Next**. If you see errors on this page, refer to [Diagnosing errors with the Azure Active Directory Connected Service](#).



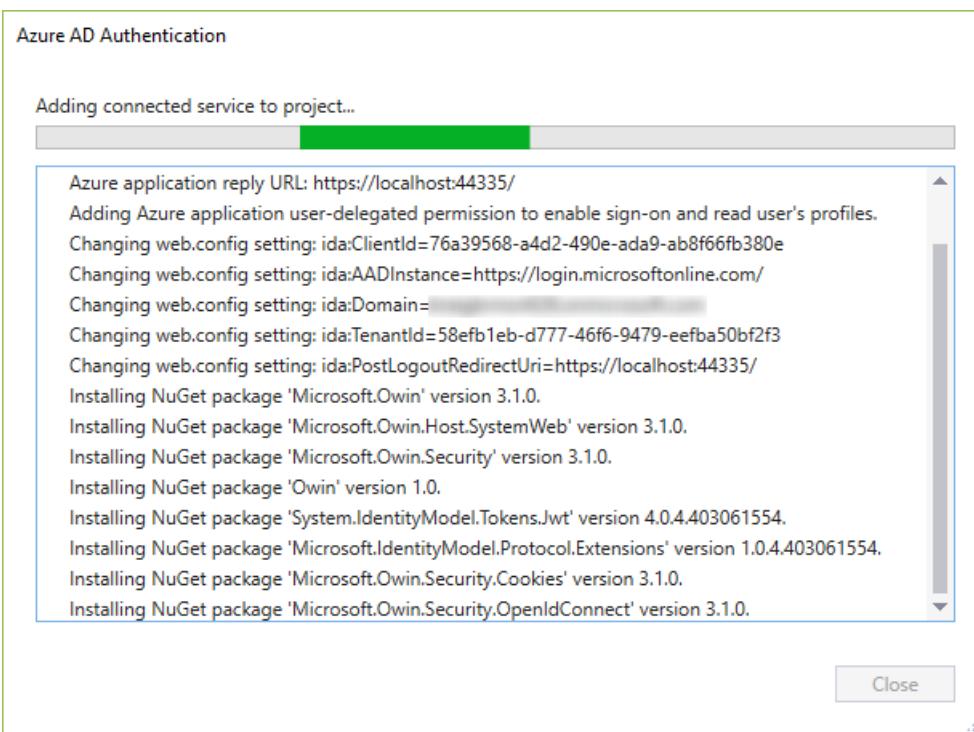
5. On the **Single-Sign On** page, select a domain from the **Domain** drop-down list. The list contains all domains accessible by the accounts listed in the Account Settings dialog of Visual Studio (**File > Account Settings...**). As an alternative, you can enter a domain name if you don't find the one you're looking for, such as `mydomain.onmicrosoft.com`. You can choose the option to create an Azure Active Directory app or use the settings from an existing Azure Active Directory app. Select **Next** when done.



6. On the **Directory Access** page, select the **Read directory data** option as desired. Developers typically include this option.



7. Select **Finish** to start modifications to your project to enable Azure AD authentication. Visual Studio shows progress during this time:



8. When the process is complete, Visual Studio opens your browser to one of the following articles, as appropriate to your project type:

- [Get started with .NET MVC projects](#)
- [Get started with WebAPI projects](#)

9. You can also see the Active Directory domain on the [Azure portal](#).

How your project is modified

When you add the connected service the wizard, Visual Studio adds Azure Active Directory and associated references to your project. Configuration files and code files in your project are also modified to add support for Azure AD. The specific modifications that Visual Studio makes depend on the project type. See the following articles for details:

- [What happened to my .NET MVC project?](#)
- [What happened to my Web API project?](#)

Next steps

- [Authentication scenarios for Azure Active Directory](#)
- [Add sign-in with Microsoft to an ASP.NET web app](#)

Getting Started with Azure Active Directory (ASP.NET MVC Projects)

10/23/2019 • 2 minutes to read • [Edit Online](#)

This article provides additional guidance after you've added Active Directory to an ASP.NET MVC project through the **Project > Connected Services** command of Visual Studio. If you've not already added the service to your project, you can do so at any time.

See [What happened to my MVC project?](#) for the changes made to your project when adding the connected service.

Requiring authentication to access controllers

All controllers in your project were adorned with the `[Authorize]` attribute. This attribute requires the user to be authenticated before accessing these controllers. To allow the controller to be accessed anonymously, remove this attribute from the controller. If you want to set the permissions at a more granular level, apply the attribute to each method that requires authorization instead of applying it to the controller class.

Adding SignIn / SignOut Controls

To add the SignIn/SignOut controls to your view, you can use the `_LoginPartial.cshtml` partial view to add the functionality to one of your views. Here is an example of the functionality added to the standard `_Layout.cshtml` view. (Note the last element in the div with class `navbar-collapse`):

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @Html.Partial("_LoginPartial")
            </div>
        </div>
        <div class="container body-content">
            @RenderBody()
            <hr />
            <footer>
                <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
            </footer>
        </div>
        @Scripts.Render("~/bundles/jquery")
        @Scripts.Render("~/bundles/bootstrap")
        @RenderSection("scripts", required: false)
    </body>
</html>

```

Next steps

- [Authentication scenarios for Azure Active Directory](#)
- [Add sign-in with Microsoft to an ASP.NET web app](#)

What happened to my MVC project (Visual Studio Azure Active Directory connected service)?

8/7/2019 • 3 minutes to read • [Edit Online](#)

This article identifies the exact changes made to an ASP.NET MVC project when adding the [Azure Active Directory connected service using Visual Studio](#).

For information on working with the connected service, see [Getting Started](#).

Added references

Affects the project file *.NET references) and `packages.config` (NuGet references).

| TYPE | REFERENCE |
|-------------|---|
| .NET; NuGet | Microsoft.IdentityModel.Protocol.Extensions |
| .NET; NuGet | Microsoft.Owin |
| .NET; NuGet | Microsoft.Owin.Host.SystemWeb |
| .NET; NuGet | Microsoft.Owin.Security |
| .NET; NuGet | Microsoft.Owin.Security.Cookies |
| .NET; NuGet | Microsoft.Owin.Security.OpenIdConnect |
| .NET; NuGet | Owin |
| .NET | System.IdentityModel |
| .NET; NuGet | System.IdentityModel.Tokens.Jwt |
| .NET | System.Runtime.Serialization |

Additional references if you selected the **Read directory data** option:

| TYPE | REFERENCE |
|-------------|---|
| .NET; NuGet | EntityFramework |
| .NET | EntityFramework.SqlServer (Visual Studio 2015 only) |
| .NET; NuGet | Microsoft.Azure.ActiveDirectory.GraphClient |
| .NET; NuGet | Microsoft.Data.Edm |
| .NET; NuGet | Microsoft.Data.OData |

| TYPE | REFERENCE |
|-------------|---|
| .NET; NuGet | Microsoft.Data.Services.Client |
| .NET; NuGet | Microsoft.IdentityModel.Clients.ActiveDirectory |
| .NET | Microsoft.IdentityModel.Clients.ActiveDirectory.WindowsForms
(Visual Studio 2015 only) |
| .NET; NuGet | System.Spatial |

The following references are removed (ASP.NET 4 projects only, as in Visual Studio 2015):

| TYPE | REFERENCE |
|-------------|---|
| .NET; NuGet | Microsoft.AspNet.Identity.Core |
| .NET; NuGet | Microsoft.AspNet.Identity.EntityFramework |
| .NET; NuGet | Microsoft.AspNet.Identity.Owin |

Project file changes

- Set the property `IISExpressSSLPort` to a distinct number.
- Set the property `WebProject_DirectoryAccessLevelKey` to 0, or 1 if you selected the **Read directory data** option.
- Set the property `IISUrl` to `https://localhost:<port>/` where `<port>` matches the `IISExpressSSLPort` value.

web.config or app.config changes

- Added the following configuration entries:

```

<appSettings>
    <add key="ida:ClientId" value="<ClientId from the new Azure AD app>" />
    <add key="ida:AADInstance" value="https://login.microsoftonline.com/" />
    <add key="ida:Domain" value="<your selected Azure domain>" />
    <add key="ida:TenantId" value="<the Id of your selected Azure AD tenant>" />
    <add key="ida:PostLogoutRedirectUri" value="<project start page, such as https://localhost:44335>" />
</appSettings>

```

- Added `<dependentAssembly>` elements under the `<runtime><assemblyBinding>` node for `System.IdentityModel.Tokens.Jwt` and `Microsoft.IdentityModel.Protocol.Extensions`.

Additional changes if you selected the **Read directory data** option:

- Added the following configuration entry under `<appSettings>`:

```

<add key="ida:ClientSecret" value="<Azure AD app's new client secret>" />

```

- Added the following elements under `<configuration>`; values for the project-mdf-file and project-catalog-id will vary:

```

<configSections>
    <!-- For more information on Entity Framework configuration, visit https://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
    EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    requirePermission="false" />
</configSections>

<connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=
    (localdb)\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\<project-mdf-file>.mdf;Initial Catalog=<project-
    catalog-id>;Integrated Security=True" providerName="System.Data.SqlClient" />
</connectionStrings>

<entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
    EntityFramework">
        <parameters>
            <parameter value="mssqllocaldb" />
        </parameters>
    </defaultConnectionFactory>
    <providers>
        <provider invariantName="System.Data.SqlClient"
        type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
</entityFramework>

```

- Added `<dependentAssembly>` elements under the `<runtime><assemblyBinding>` node for `Microsoft.Data.Services.Client`, `Microsoft.Data.Edm`, and `Microsoft.Data.OData`.

Code changes and additions

- Added the `[Authorize]` attribute to `Controllers/HomeController.cs` and any other existing controllers.
- Added an authentication startup class, `App_Start/Startup.Auth.cs`, containing startup logic for Azure AD authentication. If you selected the **Read directory data** option, this file also contains code to receive an OAuth code and exchange it for an access token.
- Added a controller class, `Controllers/AccountController.cs`, containing `SignIn` and `SignOut` methods.
- Added a partial view, `Views/Shared/_LoginPartial.cshtml`, containing an action link for `SignIn` and `SignOut`.
- Added a partial view, `Views/Account/SignoutCallback.cshtml`, containing HTML for sign-out UI.
- Updated the `Startup.Configuration` method to include a call to `ConfigureAuth(app)` if the class already existed; otherwise added a `Startup` class that includes calls the method.
- Added `Connected Services/AzureAD/ConnectedService.json` (Visual Studio 2017) or `Service References/Azure AD/ConnectedService.json` (Visual Studio 2015), containing information that Visual Studio uses to track the addition of the connected service.
- If you selected the **Read directory data** option, added `Models/ADALTokenCache.cs` and `Models/ApplicationDbContext.cs` to support token caching. Also added an additional controller and view to illustrate accessing user profile information using Azure graph APIs: `Controllers/UserProfileController.cs`, `Views/UserProfile/Index.cshtml`, and `Views/UserProfile/Relogin.cshtml`

File backup (Visual Studio 2015)

When adding the connected service, Visual Studio 2015 backs up changed and removed files. All affected files are saved in the folder `Backup/AzureAD`. Visual Studio 2017 and later does not create backups.

- `Startup.cs`
- `App_Start\IdentityConfig.cs`
- `App_Start\Startup.Auth.cs`
- `Controllers\AccountController.cs`
- `Controllers\ManageController.cs`
- `Models\IdentityModels.cs`
- `Models\ManageViewModels.cs`
- `Views\Shared_LoginPartial.cshtml`

Changes on Azure

- Created an Azure AD Application in the domain that you selected when adding the connected service.
- Updated the app to include the **Read directory data** permission if that option was selected.

[Learn more about Azure Active Directory.](#)

Next steps

- [Authentication scenarios for Azure Active Directory](#)
- [Add sign-in with Microsoft to an ASP.NET web app](#)

Get Started with Azure Active Directory (WebAPI projects)

8/7/2019 • 2 minutes to read • [Edit Online](#)

This article provides additional guidance after you've added Active Directory to an ASP.NET WebAPI project through the **Project > Connected Services** command of Visual Studio. If you've not already added the service to your project, you can do so at any time.

See [What happened to my WebAPI project?](#) for the changes made to your project when adding the connected service.

Requiring authentication to access controllers

All controllers in your project were adorned with the `[Authorize]` attribute. This attribute requires the user to be authenticated before accessing the APIs defined by these controllers. To allow the controller to be accessed anonymously, remove this attribute from the controller. If you want to set the permissions at a more granular level, apply the attribute to each method that requires authorization instead of applying it to the controller class.

Next steps

- [Authentication scenarios for Azure Active Directory](#)
- [Add sign-in with Microsoft to an ASP.NET web app](#)

What happened to my WebAPI project (Visual Studio Azure Active Directory connected service)

8/7/2019 • 2 minutes to read • [Edit Online](#)

This article identifies the exact changes made to ASP.NET WebAPI, ASP.NET Single-Page Application, and ASP.NET Azure API projects when adding the [Azure Active Directory connected service using Visual Studio](#). Also applies to the ASP.NET Azure Mobile Service projects in Visual Studio 2015.

For information on working with the connected service, see [Getting Started](#).

Added references

Affects the project file *.NET references) and `packages.config` (NuGet references).

| TYPE | REFERENCE |
|-------------|---|
| .NET; NuGet | Microsoft.Owin |
| .NET; NuGet | Microsoft.Owin.Host.SystemWeb |
| .NET; NuGet | Microsoft.Owin.Security |
| .NET; NuGet | Microsoft.Owin.Security.ActiveDirectory |
| .NET; NuGet | Microsoft.Owin.Security.Jwt |
| .NET; NuGet | Microsoft.Owin.Security.OAuth |
| .NET; NuGet | Owin |
| .NET; NuGet | System.IdentityModel.Tokens.Jwt |

Additional references if you selected the **Read directory data** option:

| TYPE | REFERENCE |
|-------------|---|
| .NET; NuGet | EntityFramework |
| .NET | EntityFramework.SqlServer (Visual Studio 2015 only) |
| .NET; NuGet | Microsoft.Azure.ActiveDirectory.GraphClient |
| .NET; NuGet | Microsoft.Data.Edm |
| .NET; NuGet | Microsoft.Data.OData |
| .NET; NuGet | Microsoft.Data.Services.Client |
| .NET; NuGet | Microsoft.IdentityModel.Clients.ActiveDirectory |

| TYPE | REFERENCE |
|-------------|---|
| .NET | Microsoft.IdentityModel.Clients.ActiveDirectory.WindowsForms
(Visual Studio 2015 only) |
| .NET; NuGet | System.Spatial |

The following references are removed (ASP.NET 4 projects only, as in Visual Studio 2015):

| TYPE | REFERENCE |
|-------------|---|
| .NET; NuGet | Microsoft.AspNet.Identity.Core |
| .NET; NuGet | Microsoft.AspNet.Identity.EntityFramework |
| .NET; NuGet | Microsoft.AspNet.Identity.Owin |

Project file changes

- Set the property `IISExpressSSLPort` to a distinct number.
- Set the property `WebProject_DirectoryAccessLevelKey` to 0, or 1 if you selected the **Read directory data** option.
- Set the property `IISUrl` to `https://localhost:<port>/` where `<port>` matches the `IISExpressSSLPort` value.

web.config or app.config changes

- Added the following configuration entries:

```

<appSettings>
    <add key="ida:ClientId" value="<ClientId from the new Azure AD app>" />
    <add key="ida:Tenant" value="<your selected Azure domain>" />
    <add key="ida:Audience" value="<your selected domain + / + project name>" />
</appSettings>

```

- Visual Studio 2017 only: Also added the following entry under `<appSettings>`

```

<add key="ida:MetadataAddress" value="<domain URL + /federationmetadata/2007-06/federationmetadata.xml>" />

```

- Added `<dependentAssembly>` elements under the `<runtime><assemblyBinding>` node for `System.IdentityModel.Tokens.Jwt`.

- If you selected the **Read directory data** option, added the following configuration entry under `<appSettings>`:

```

<add key="ida:Password" value="<Your Azure AD app's new password>" />

```

Code changes and additions

- Added the `[Authorize]` attribute to `Controllers/ValueController.cs` and any other existing controllers.
- Added an authentication startup class, `App_Start/Startup.Auth.cs`, containing startup logic for Azure AD authentication, or modified it accordingly. If you selected the **Read directory data** option, this file also

contains code to receive an OAuth code and exchange it for an access token.

- (Visual Studio 2015 with ASP.NET 4 app only) Removed `App_Start/IdentityConfig.cs` and added `Controllers/AccountController.cs`, `Models/IdentityModel.cs`, and `Providers/ApplicationAuthProvider.cs`.
- Added `Connected Services/AzureAD/ConnectedService.json` (Visual Studio 2017) or `Service References/Azure AD/ConnectedService.json` (Visual Studio 2015), containing information that Visual Studio uses to track the addition of the connected service.

File backup (Visual Studio 2015)

When adding the connected service, Visual Studio 2015 backs up changed and removed files. All affected files are saved in the folder `Backup/AzureAD`. Visual Studio 2017 does not create backups.

- `Startup.cs`
- `App_Start\IdentityConfig.cs`
- `App_Start\Startup.Auth.cs`
- `Controllers\AccountController.cs`
- `Controllers\ManageController.cs`
- `Models\IdentityModels.cs`
- `Models\ApplicationOAuthProvider.cs`

Changes on Azure

- Created an Azure AD Application in the domain that you selected when adding the connected service.
- Updated the app to include the **Read directory data** permission if that option was selected.

[Learn more about Azure Active Directory.](#)

Next steps

- [Authentication scenarios for Azure Active Directory](#)
- [Add sign-in with Microsoft to an ASP.NET web app](#)

Diagnosing errors with the Azure Active Directory Connected Service

11/12/2019 • 2 minutes to read • [Edit Online](#)

While detecting previous authentication code, the Azure Active Director connect server detected an incompatible authentication type.

To correctly detect previous authentication code in a project, the project must be built. If you see this error and you don't have a previous authentication code in your project, rebuild and try again.

Project types

The connected service checks the type of project you're developing so it can inject the right authentication logic into the project. If there's any controller that derives from `ApiController` in the project, the project is considered a WebAPI project. If there are only controllers that derive from `MVC.Controller` in the project, the project is considered an MVC project. The connected service doesn't support any other project type.

Compatible authentication code

The connected service also checks for authentication settings that have been previously configured or are compatible with the service. If all settings are present, it's considered a re-entrant case, and the connected service opens display the settings. If only some of the settings are present, it's considered an error case.

In an MVC project, the connected service checks for any of the following settings, which result from previous use of the service:

```
<add key="ida:ClientId" value="" />
<add key="ida:Tenant" value="" />
<add key="ida:AADInstance" value="" />
<add key="ida:PostLogoutRedirectUri" value="" />
```

Also, the connected service checks for any of the following settings in a Web API project, which result from previous use of the service:

```
<add key="ida:ClientId" value="" />
<add key="ida:Tenant" value="" />
<add key="ida:Audience" value="" />
```

Incompatible authentication code

Finally, the connected service attempts to detect versions of authentication code that have been configured with previous versions of Visual Studio. If you received this error, it means your project contains an incompatible authentication type. The connected service detects the following types of authentication from previous versions of Visual Studio:

- Windows Authentication
- Individual User Accounts
- Organizational Accounts

To detect Windows Authentication in an MVC project, the connected looks for the `authentication` element in your `web.config` file.

```
<configuration>
  <system.web>
    <authentication mode="Windows" />
  </system.web>
</configuration>
```

To detect Windows Authentication in a Web API project, the connected service looks for the `IISExpressWindowsAuthentication` element in your project's `.csproj` file:

```
<Project>
  <PropertyGroup>
    <IISExpressWindowsAuthentication>enabled</IISExpressWindowsAuthentication>
  </PropertyGroup>
</Project>
```

To detect Individual User Accounts authentication, the connected service looks for the package element in your `packages.config` file.

```
<packages>
  <package id="Microsoft.AspNet.Identity.EntityFramework" version="2.1.0" targetFramework="net45" />
</packages>
```

To detect an old form of Organizational Account authentication, the connected service looks for the following element in `web.config`:

```
<configuration>
  <appSettings>
    <add key="ida:Realm" value="***" />
  </appSettings>
</configuration>
```

To change the authentication type, remove the incompatible authentication type and try adding the connected service again.

For more information, see [Authentication Scenarios for Azure AD](#).

Microsoft identity platform authentication libraries

10/31/2019 • 3 minutes to read • [Edit Online](#)

The [Microsoft identity platform endpoint](#) supports the industry-standard OAuth 2.0 and OpenID Connect 1.0 protocols. The Microsoft Authentication Library (MSAL) is designed to work with the Microsoft identity platform endpoint. You can also use open-source libraries that support OAuth 2.0 and OpenID Connect 1.0.

We recommend that you use libraries written by protocol domain experts who follow a Security Development Lifecycle (SDL) methodology. Such methodologies include [the one that Microsoft follows](#). If you hand code for the protocols, you should follow a methodology such as Microsoft SDL. Pay close attention to the security considerations in the standards specifications for each protocol.

NOTE

Are you looking for the Azure Active Directory Authentication Library (ADAL)? Check out the [ADAL library guide](#).

Types of libraries

The Microsoft identity platform endpoint works with two types of libraries:

- **Client libraries:** Native clients and servers use client libraries to acquire access tokens for calling a resource such as Microsoft Graph.
- **Server middleware libraries:** Web apps use server middleware libraries for user sign-in. Web APIs use server middleware libraries to validate tokens that are sent by native clients or by other servers.

Library support

Libraries come in two support categories:

- **Microsoft-supported:** Microsoft provides fixes for these libraries and has done SDL due diligence on these libraries.
- **Compatible:** Microsoft has tested these libraries in basic scenarios and has confirmed that they work with the Microsoft identity platform endpoint. Microsoft doesn't provide fixes for these libraries and hasn't done a review of these libraries. Issues and feature requests should be directed to the library's open-source project.

For a list of libraries that work with the Microsoft identity platform endpoint, see the following sections.

Microsoft-supported client libraries

Use client authentication libraries to acquire a token for calling a protected web API.

PLATFORM	LIBRARY	DOWNLOAD	SOURCE CODE	SAMPLE	REFERENCE	CONCEPTUAL DOC	ROADMAP
	MSAL.js	NPM	GitHub	Single-page app	Reference	Conceptual docs	Roadmap

Platform	Library	Download	Source Code	Sample	Reference	Conceptual Doc	Roadmap
	MSAL Angular JS	NPM	GitHub				
	MSAL Angular (Preview)	NPM	GitHub				
  	MSAL.NET	NuGet	GitHub	Desktop app	MSAL.NET	Conceptual docs	Roadmap
	MSAL Python (Preview)	PyPI	GitHub	Samples	ReadTheDocs	Wiki	Roadmap
	MSAL Java (Preview)	Maven	GitHub	Samples	Reference	Wiki	Roadmap
iOS & macOS	MSAL iOS and macOS	GitHub	GitHub	iOS app, macOS app	Reference	Conceptual docs	
	MSAL Android	Central repository	GitHub	Android app	JavaDocs	Conceptual docs	Roadmap

Microsoft-supported server middleware libraries

Use middleware libraries to help protect web applications and web APIs. Web apps or web APIs written with ASP.NET or ASP.NET Core use the middleware libraries.

Platform	Library	Download	Source Code	Sample	Reference
 	ASP.NET Security	NuGet	GitHub	MVC app	ASP.NET API reference
	IdentityModel Extensions for .NET		GitHub	MVC app	Reference

PLATFORM	LIBRARY	DOWNLOAD	SOURCE CODE	SAMPLE	REFERENCE
	Azure AD Passport	NPM	GitHub	Web app	

Microsoft-supported libraries by OS / language

In term of supported operating systems vs languages, the mapping is the following:

	WINDOWS	LINUX	MACOS	IOS	ANDROID
	MSAL.js	MSAL.js	MSAL.js	MSAL.js	MSAL.js
	ASP.NET, ASP.NET Core, MSAL.Net (.NET FW, Core, UWP)	ASP.NET Core, MSAL.Net (.NET Core)	ASP.NET Core, MSAL.Net (MacOS)	MSAL.Net (Xamarin.iOS)	MSAL.Net (Xamarin.Android)
Swift Objective-C			MSAL for iOS and macOS	MSAL for iOS and macOS	
	msal4j	msal4j	msal4j		MSAL Android
 Python	MSAL Python	MSAL Python	MSAL Python		
 Node.JS	Passport.node	Passport.node	Passport.node		

See also [Scenarios by supported platforms and languages](#)

Compatible client libraries

PLATFORM	LIBRARY NAME	TESTED VERSION	SOURCE CODE	SAMPLE
	Hello.js	Version 1.13.5	Hello.js	SPA
	Scribe Java	Version 3.2.0	ScribeJava	

Platform	Library Name	Tested Version	Source Code	Sample
	Gluu OpenID Connect library	Version 3.0.2	Gluu OpenID Connect library	
	Requests-OAuthlib	Version 1.2.0	Requests-OAuthlib	
	openid-client	Version 2.4.5	openid-client	
	The PHP League oauth2-client	Version 1.4.2	oauth2-client	
	OmniAuth	omniauth: 1.3.1 omniauth-oauth2: 1.4.0	OmniAuth OmniAuth OAuth2	
iOS, macOS, & Android	React Native App Auth	Version 4.2.0	React Native App Auth	

For any standards-compliant library, you can use the Microsoft identity platform endpoint. It's important to know where to go for support:

- For issues and new feature requests in library code, contact the library owner.
- For issues and new feature requests in the service-side protocol implementation, contact Microsoft.
- [File a feature request](#) for additional features you want to see in the protocol.
- [Create a support request](#) if you find an issue where the Microsoft identity platform endpoint isn't compliant with OAuth 2.0 or OpenID Connect 1.0.

Related content

For more information about the Microsoft identity platform endpoint, see the [Microsoft identity platform overview](#).

Azure Active Directory app manifest

10/15/2019 • 11 minutes to read • [Edit Online](#)

The application manifest contains a definition of all the attributes of an application object in the Microsoft identity platform. It also serves as a mechanism for updating the application object. For more info on the Application entity and its schema, see the [Graph API Application entity documentation](#).

You can configure an app's attributes through the Azure portal or programmatically using [REST API](#) or [PowerShell](#). However, there are some scenarios where you'll need to edit the app manifest to configure an app's attribute. These scenarios include:

- If you registered the app as Azure AD multi-tenant and personal Microsoft accounts, you can't change the supported Microsoft accounts in the UI. Instead, you must use the application manifest editor to change the supported account type.
- If you need to define permissions and roles that your app supports, you must modify the application manifest.

Configure the app manifest

To configure the application manifest:

1. Sign in to the [Azure portal](#).
2. Select the **Azure Active Directory** service, and then select **App registrations**.
3. Select the app you want to configure.
4. From the app's **Overview** page, select the **Manifest** section. A web-based manifest editor opens, allowing you to edit the manifest within the portal.
 Optionally, you can select **Download** to edit the manifest locally, and then use **Upload** to reapply it to your application.

Manifest reference

NOTE

If you can't see the **Example value** column after the **Description**, maximize your browser window and scroll/swipe until you see the **Example value** column.

KEY	VALUE TYPE	DESCRIPTION	EXAMPLE VALUE
<code>accessTokenAcceptedVersion</code>	Nullable Int32	<p>Specifies the access token version expected by the resource. This changes the version and format of the JWT produced independent of the endpoint or client used to request the access token.</p> <p>The endpoint used, v1.0 or v2.0, is chosen by the client and only impacts the version of id_tokens. Resources need to explicitly configure <code>accessTokenAcceptedVersion</code> to indicate the supported access token format.</p> <p>Possible values for <code>accessTokenAcceptedVersion</code> are 1, 2, or null. If the value is null, this defaults to 1, which corresponds to the v1.0 endpoint.</p> <p>If <code>signInAudience</code> is <code>AzureADandPersonalMicrosoftAccount</code>, the value must be 2</p>	2
<code>addIns</code>	Collection	Defines custom behavior that a consuming service can use to call an app in specific contexts. For example, applications that can render file streams may set the addIns property for its "FileHandler" functionality. This will let services like Office 365 call the application in the context of a document the user is working on.	{ "id":"968A844F-7A47-430C-9163-07AE7C31D407" "type": "FileHandler", "properties": [{"key": "version", "value": "2"}] }

KEY	VALUE TYPE	DESCRIPTION	EXAMPLE VALUE
<code>allowPublicClient</code>	Boolean	Specifies the fallback application type. Azure AD infers the application type from the replyUrlsWithType by default. There are certain scenarios where Azure AD cannot determine the client app type (e.g. ROPC flow where HTTP request happens without a URL redirection). In those cases Azure AD will interpret the application type based on the value of this property. If this value is set to true the fallback application type is set as public client, such as an installed app running on a mobile device. The default value is false which means the fallback application type is confidential client such as web app.	<code>false</code>
<code>availableToOtherTenants</code>	Boolean	true if the application is shared with other tenants; otherwise, false. <i>Note: This is available only in App registrations (Legacy) experience. Replaced by <code>signInAudience</code> in the App registrations experience.</i>	
<code>appId</code>	String	Specifies the unique identifier for the app that is assigned to an app by Azure AD.	<code>"601790de-b632-4f57-9523-ee7cb6ceba95"</code>
<code>appRoles</code>	Collection	Specifies the collection of roles that an app may declare. These roles can be assigned to users, groups, or service principals. For more examples and info, see Add app roles in your application and receive them in the token	[{ "allowedMemberTypes": [>User"], "description":"Read-only access to device information", "displayName":"Read Only", "id":guid, "isEnabled":true, "value":"ReadOnly" }]
<code>displayName</code>	String	The display name for the app. <i>Note: This is available only in App registrations (Legacy) experience. Replaced by <code>name</code> in the App registrations experience.</i>	<code>"MyRegisteredApp"</code>
<code>errorUrl</code>	String	Unsupported.	
<code>groupMembershipClaims</code>	String	Configures the <code>groups</code> claim issued in a user or OAuth 2.0 access token that the app expects. To set this attribute, use one of the following valid string values: - <code>"None"</code> - <code>"SecurityGroup"</code> (for security groups and Azure AD roles) - <code>"All"</code> (this will get all of the security groups, distribution groups, and Azure AD directory roles that the signed-in user is a member of)	<code>"SecurityGroup"</code>
<code>homepage</code>	String	The URL to the application's homepage. <i>Note: This is available only in App registrations (Legacy) experience. Replaced by <code>signInUrl</code> in the App registrations experience.</i>	<code>"https://MyRegisteredApp"</code>
<code>objectId</code>	String	The unique identifier for the app in the directory. <i>Note: This is available only in App registrations (Legacy) experience. Replaced by <code>id</code> in the App registrations experience.</i>	<code>"f7f9acf-ae0c-4d6c-b489-0a81dc1652dd"</code>

KEY	VALUE TYPE	DESCRIPTION	EXAMPLE VALUE
<code>optionalClaims</code>	String	The optional claims returned in the token by the security token service for this specific app. At this time, apps that support both personal accounts and Azure AD (registered through the app registration portal) cannot use optional claims. However, apps registered for just Azure AD using the v2.0 endpoint can get the optional claims they requested in the manifest. For more info, see optional claims .	<code>null</code>
<code>id</code>	String	The unique identifier for the app in the directory. This ID is not the identifier used to identify the app in any protocol transaction. It's used for the referencing the object in directory queries.	<code>"f7f9acf-ae0c-4d6c-b489-0a81dc1652dd"</code>
<code>identifierUris</code>	String Array	User-defined URI(s) that uniquely identify a Web app within its Azure AD tenant, or within a verified custom domain if the app is multi-tenant.	<code>["https://MyRegisteredApp"]</code>
<code>informationalUrls</code>	String	Specifies the links to the app's terms of service and privacy statement. The terms of service and privacy statement are surfaced to users through the user consent experience. For more info, see How to: Add Terms of service and privacy statement for registered Azure AD apps .	<code>{ "marketing": "https://MyRegisteredApp/mark", "privacy": "https://MyRegisteredApp/privac", "support": "https://MyRegisteredApp/suppor", "termsOfService": "https://MyRegisteredApp" }</code>
<code>keyCredentials</code>	Collection	Holds references to app-assigned credentials, string-based shared secrets and X.509 certificates. These credentials are used when requesting access tokens (when the app is acting as a client rather than as resource).	<code>[{ "customKeyIdentifier": null, "endDate": "2018-09-13T00:00:00Z", "keyId": "guid>", "startDate": "2017-09-12T00:00:00Z", "type": "AsymmetricX509Cert", "usage": "Verify", "value": null }]</code>
<code>knownClientApplications</code>	String Array	Used for bundling consent if you have a solution that contains two parts: a client app and a custom web API app. If you enter the appId of the client app into this value, the user will only have to consent once to the client app. Azure AD will know that consenting to the client means implicitly consenting to the web API and will automatically provision service principals for both the client and web API at the same time. Both the client and the web API app must be registered in the same tenant.	<code>["f7f9acf-ae0c-4d6c-b489-0a81dc1652dd"]</code>
<code>logoUrl</code>	String	Read only value that points to the CDN URL to logo that was uploaded in the portal.	<code>"https://MyRegisteredAppLogo"</code>
<code>logoutUrl</code>	String	The URL to log out of the app.	<code>"https://MyRegisteredAppLogout"</code>
<code>name</code>	String	The display name for the app.	<code>"MyRegisteredApp"</code>
<code>oauth2AllowImplicitFlow</code>	Boolean	Specifies whether this web app can request OAuth2.0 implicit flow access tokens. The default is false. This flag is used for browser-based apps, like Javascript single-page apps. To learn more, enter <code>OAuth 2.0 implicit grant flow</code> in the table of contents and see the topics about implicit flow.	<code>false</code>

KEY	VALUE TYPE	DESCRIPTION	EXAMPLE VALUE
<code>oauth2AllowIdTokenImplicitFlow</code>	Boolean	Specifies whether this web app can request OAuth2.0 implicit flow ID tokens. The default is false. This flag is used for browser-based apps, like Javascript single-page apps.	<code>false</code>
<code>oauth2Permissions</code>	Collection	Specifies the collection of OAuth 2.0 permission scopes that the web API (resource) app exposes to client apps. These permission scopes may be granted to client apps during consent.	[{ "adminConsentDescription": "Allow the app to access resources on behalf of the signed-in user.", "adminConsentDisplayName": "Access resource1", "id": "<guid>", "isEnabled": true, "type": "User", "userConsentDescription": "Allow the app to access resource1 on your behalf.", "userConsentDisplayName": "Access resources", "value": "user_impersonation" }]
<code>oauth2RequiredPostResponse</code>	Boolean	Specifies whether, as part of OAuth 2.0 token requests, Azure AD will allow POST requests, as opposed to GET requests. The default is false, which specifies that only GET requests will be allowed.	<code>false</code>
<code>parentalControlSettings</code>	String	<code>countriesBlockedForMinors</code> specifies the countries in which the app is blocked for minors. <code>legalAgeGroupRule</code> specifies the legal age group rule that applies to users of the app. Can be set to <code>Allow</code> , <code>RequireConsentForPrivacyServices</code> , <code>RequireConsentForMinors</code> , <code>RequireConsentForKids</code> , or <code>BlockMinors</code> .	{ "countriesBlockedForMinors": [], "legalAgeGroupRule": "Allow" }
<code>passwordCredentials</code>	Collection	See the description for the <code>keyCredentials</code> property.	[{ "customKeyIdentifier": null, "endDate": "2018-10-19T17:59:59.6521653Z", "keyId": "<guid>", "startDate": "2016-10-19T17:59:59.6521653Z", "value": null }]
<code>preAuthorizedApplications</code>	Collection	Lists applications and requested permissions for implicit consent. Requires an admin to have provided consent to the application. preAuthorizedApplications do not require the user to consent to the requested permissions. Permissions listed in preAuthorizedApplications do not require user consent. However, any additional requested permissions not listed in preAuthorizedApplications require user consent.	[{ "appId": "abcdefg2-000a-1111-a0e5-812ed8dd72e8", "permissionIds": ["8748f7db-21fe-4c83-8ab5-53033933c8f1"] }]
<code>publicClient</code>	Boolean	Specifies whether this application is a public client (such as an installed application running on a mobile device). <i>Note: This is available only in App registrations (Legacy) experience.</i> Replaced by <code>allowPublicClient</code> in the App registrations experience.	
<code>publisherDomain</code>	String	The verified publisher domain for the application. Read-only.	https://www.contoso.com

KEY	VALUETYPE	DESCRIPTION	EXAMPLE VALUE
<code>replyUrls</code>	String array	This multi-value property holds the list of registered redirect_uri values that Azure AD will accept as destinations when returning tokens. <i>Note: This is available only in App registrations (Legacy) experience.</i> Replaced by <code>replyUrlsWithType</code> in the App registrations experience.	
<code>replyUrlsWithType</code>	Collection	This multi-value property holds the list of registered redirect_uri values that Azure AD will accept as destinations when returning tokens. Each URI value should contain an associated app type value. Supported type values are: <ul style="list-style-type: none">• <code>Web</code>• <code>InstalledClient</code> Learn more about replyUrl restrictions and limitations .	<pre>"replyUrlsWithType": [{ "url": "https://localhost:4400/services", "type": "InstalledClient" }]</pre>
<code>requiredResourceAccess</code>	Collection	With dynamic consent, <code>requiredResourceAccess</code> drives the admin consent experience and the user consent experience for users who are using static consent. However, this does not drive the user consent experience for the general case. <code>resourceAppId</code> is the unique identifier for the resource that the app requires access to. This value should be equal to the appId declared on the target resource app. <code>resourceAccess</code> is an array that lists the OAuth2.0 permission scopes and app roles that the app requires from the specified resource. Contains the <code>id</code> and <code>type</code> values of the specified resources.	<pre>[{ "resourceAppId": "00000002-0000-0000-0000-000000000000", "resourceAccess": [{ "id": "311a71cc-e848-46a1-bdf8-97ff7156d8ec", "type": "Scope" }] }]</pre>
<code>samlMetadataUrl</code>	String	The URL to the SAML metadata for the app.	https://MyRegisteredAppSAMLMetadata
<code>signInUrl</code>	String	Specifies the URL to the app's home page.	https://MyRegisteredApp
<code>signInAudience</code>	String	Specifies what Microsoft accounts are supported for the current application. Supported values are: <ul style="list-style-type: none">• AzureADMyOrg - Users with a Microsoft work or school account in my organization's Azure AD tenant (i.e. single tenant)• AzureADMultipleOrgs - Users with a Microsoft work or school account in any organization's Azure AD tenant (i.e. multi-tenant)• AzureADandPersonalMicrosoftAccount - Users with a personal Microsoft account, or a work or school account in any organization's Azure AD tenant	AzureADandPersonalMicrosoftAccount
<code>tags</code>	String Array	Custom strings that can be used to categorize and identify the application.	<pre>["ProductionApp"]</pre>

Common issues

Manifest limits

An application manifest has multiple attributes that are referred to as collections; for example, approles, keycredentials, knownClientApplications, identifierUris, redirectUris, requiredResourceAccess, and oauth2Permissions. Within the complete application manifest for any application, the total number of entries in all the collections combined has been capped at 1200. If you already have 100 redirect URIs specified in the application manifest, then you're only left with 1100 remaining entries to use across all other collections combined that make up the manifest.

NOTE

In case you try to add more than 1200 entries in the application manifest, you may see an error "Failed to update application xxxxxx. Error details: The size of the manifest has exceeded its limit. Please reduce the number of values and retry your request."

Unsupported attributes

The application manifest represents the schema of the underlying application model in Azure AD. As the underlying schema evolves, the manifest editor will be updated to reflect the new schema from time to time. As a result, you may notice new attributes showing up in the application manifest. In rare occasions, you may notice a syntactic or semantic change in the existing attributes or you may find an attribute that existed previously are not supported anymore. For example, you will see new attributes in the [App registrations](#) which are known with a different name in the App registrations (Legacy) experience.

APP REGISTRATIONS (LEGACY)	APP REGISTRATIONS
availableToOtherTenants	signInAudience
displayName	name
errorUrl	-
homepage	signInUrl
objectId	Id
publicClient	allowPublicClient
replyUrls	replyUrlsWithType

For descriptions for these attributes, see the [manifest reference](#) section.

When you try to upload a previously downloaded manifest, you may see one of the following errors. This is likely because the manifest editor now supports a newer version of the schema, which doesn't match with the one you're trying to upload.

- "Failed to update xxxxxx application. Error detail: Invalid object identifier 'undefined'. []."
- "Failed to update xxxxxx application. Error detail: One or more property values specified are invalid. []."
- "Failed to update xxxxxx application. Error detail: Not allowed to set availableToOtherTenants in this api version for update. []."
- "Failed to update xxxxxx application. Error detail: Updates to 'replyUrls' property is not allowed for this application. Use 'replyUrlsWithType' property instead. []."
- "Failed to update xxxxxx application. Error detail: A value without a type name was found and no expected type is available. When the model is specified, each value in the payload must have a type which can be either specified in the payload, explicitly by the caller or implicitly inferred from the parent value. []"

When you see one of these errors, we recommend the following:

1. Edit the attributes individually in the manifest editor instead of uploading a previously downloaded manifest. Use the [manifest reference](#) table to understand the syntax and semantics of old and new attributes so that you can successfully edit the attributes you're interested in.
2. If your workflow requires you to save the manifests in your source repository for use later, we suggest rebasing the saved manifests in your repository with the one you see in the [App registrations](#) experience.

Next steps

- For more info on the relationship between an app's application and service principal object(s), see [Application and service principal objects in Azure AD](#).
- See the [Microsoft identity platform developer glossary](#) for definitions of some of the core Microsoft identity platform developer concepts.

Use the following comments section to provide feedback that helps refine and shape our content.

Android Microsoft Authentication Library (MSAL) configuration file

10/23/2019 • 6 minutes to read • [Edit Online](#)

MSAL ships with a [default configuration JSON file](#) that you customize to define the behavior of your public client app for things such as the default authority, which authorities you'll use, and so on.

This article will help you understand the various settings in the configuration file and how to specify the configuration file to use in your MSAL-based app.

Configuration settings

General settings

PROPERTY	DATA TYPE	REQUIRED	NOTES
<code>client_id</code>	String	Yes	Your app's Client ID from the Application registration page
<code>redirect_uri</code>	String	Yes	Your app's Redirect URI from the Application registration page
<code>authorities</code>	List<Authority>	No	The list of authorities your app needs
<code>authorization_user_agent</code>	AuthorizationAgent (enum)	No	Possible values: <code>DEFAULT</code> , <code>BROWSER</code> , <code>WEBVIEW</code>
<code>http</code>	HttpConfiguration	No	Configure <code>HttpURLConnection</code> , <code>connect_timeout</code> and <code>read_timeout</code>
<code>logging</code>	LoggingConfiguration	No	Specifies the level of logging detail. Optional configurations include: <code>pii_enabled</code> , which takes a boolean value, and <code>log_level</code> , which takes <code>ERROR</code> , <code>WARNING</code> , <code>INFO</code> , or <code>VERBOSE</code> .

`client_id`

The client ID or app ID that was created when you registered your application.

`redirect_uri`

The redirect URI you registered when you registered your application. If the redirect URI is to a broker app, refer to [Redirect URI for public client apps](#) to ensure you're using the correct redirect URI format for your broker app.

`authorities`

The list of authorities that are known and trusted by you. In addition to the authorities listed here, MSAL also queries Microsoft to get a list of clouds and authorities known to Microsoft. In this list of authorities, specify the type of the

authority and any additional optional parameters such as `"audience"`, which should align with the audience of your app based on your app's registration. The following is an example list of authorities:

```
// Example AzureAD and Personal Microsoft Account
{
    "type": "AAD",
    "audience": {
        "type": "AzureADandPersonalMicrosoftAccount"
    },
    "default": true // Indicates that this is the default to use if not provided as part of the acquireToken or
    acquireTokenSilent call
},
// Example AzureAD My Organization
{
    "type": "AAD",
    "audience": {
        "type": "AzureADMyOrg",
        "tenantId": "contoso.com" // Provide your specific tenant ID here
    }
},
// Example AzureAD Multiple Organizations
{
    "type": "AAD",
    "audience": {
        "type": "AzureADMultipleOrgs"
    }
},
//Example PersonalMicrosoftAccount
{
    "type": "AAD",
    "audience": {
        "type": "PersonalMicrosoftAccount"
    }
}
```

Map AAD authority & audience to Microsoft identity platform endpoints

TYPE	AUDIENCE	TENANT ID	AUTHORITY_URL	RESULTING ENDPOINT	NOTES
AAD	AzureADandPersonalMicrosoftAccount			https://login.microsoftonline.com/common	<code>common</code> is a tenant alias for where the account is. Such as a specific Azure Active Directory tenant or the Microsoft account system.
AAD	AzureADMyOrg	contoso.com		https://login.microsoftonline.com/contoso.com	Only accounts present in contoso.com can acquire a token. Any verified domain, or the tenant GUID, may be used as the tenant ID.

Type	Audience	Tenant ID	Authority URL	Resulting Endpoint	Notes
AAD	AzureADMultipleOrgs			https://login.microsoftonline.com/organizations	Only Azure Active Directory accounts can be used with this endpoint. Microsoft accounts can be members of organizations. To acquire a token using a Microsoft account for a resource in an organization, specify the organizational tenant from which you want the token.
AAD	PersonalMicrosoftAccount			https://login.microsoftonline.com/consumers	Only Microsoft accounts can use this endpoint.
B2C			See Resulting Endpoint	https://login.microsoftonline.com/tfp/contoso.onmicrosoft.com/B2C_1_SISOPolicy/	Only accounts present in the contoso.onmicrosoft.com tenant can acquire a token. In this example, the B2C policy is part of the Authority URL path.

Note

Authority validation cannot be enabled and disabled in MSAL. Authorities are either known to you as the developer as specified via configuration or known to Microsoft via metadata. If MSAL receives a request for a token to an unknown authority, an `MsalClientException` of type `UnknownAuthority` results.

Authority properties

Property	Data Type	Required	Notes
<code>type</code>	String	Yes	Mirrors the audience or account type your app targets. Possible values: <code>AAD</code> , <code>B2C</code>
<code>audience</code>	Object	No	Only applies when <code>type= AAD</code> . Specifies the identity your app targets. Use the value from your app registration

PROPERTY	DATA TYPE	REQUIRED	NOTES
<code>authority_url</code>	String	Yes	Required only when type= <code>B2C</code> . Specifies the authority URL or policy your app should use
<code>default</code>	boolean	Yes	A single <code>"default":true</code> is required when one or more authorities is specified.

Audience Properties

PROPERTY	DATA TYPE	REQUIRED	NOTES
<code>type</code>	String	Yes	Specifies the audience your app wants to target. Possible values: <code>AzureADandPersonalMicrosoftAccount</code> , <code>PersonalMicrosoftAccount</code> , <code>AzureADMultipleOrgs</code> , <code>AzureADMyOrg</code>
<code>tenant_id</code>	String	Yes	Required only when <code>"type": "AzureADMyOrg"</code> . Optional for other <code>type</code> values. This can be a tenant domain such as <code>contoso.com</code> , or a tenant ID such as <code>72f988bf-86f1-41af-91ab-2d7cd011db46</code>)

`authorization_user_agent`

Indicates whether to use an embedded webview, or the default browser on the device, when signing in an account or authorizing access to a resource.

Possible values:

- `DEFAULT` : Prefers the system browser. Uses the embedded web view if a browser isn't available on the device.
- `WEBVIEW` : Use the embedded web view.
- `BROWSER` : Uses the default browser on the device.

`multiple_clouds_supported`

For clients that support multiple national clouds, specify `true`. The Microsoft identity platform will then automatically redirect to the correct national cloud during authorization and token redemption. You can determine the national cloud of the signed-in account by examining the authority associated with the `AuthenticationResult`. Note that the `AuthenticationResult` doesn't provide the national cloud-specific endpoint address of the resource for which you request a token.

`broker_redirect_uri_registered`

A boolean that indicates whether you're using a Microsoft Identity broker compatible in-broker redirect URI. Set to `false` if you don't want to use the broker within your app.

If you're using the AAD Authority with Audience set to `"MicrosoftPersonalAccount"`, the broker won't be used.

http

Configure global settings for HTTP timeouts, such as:

PROPERTY	DATA TYPE	REQUIRED	NOTES
<code>connect_timeout</code>	int	No	Time in milliseconds
<code>read_timeout</code>	int	No	Time in milliseconds

logging

The following global settings are for logging:

PROPERTY	DATA TYPE	REQUIRED	NOTES
<code>pii_enabled</code>	boolean	No	Whether to emit personal data
<code>log_level</code>	boolean	No	Which log messages to output
<code>logcat_enabled</code>	boolean	No	Whether to output to log cat in addition to the logging interface

account_mode

Specifies how many accounts can be used within your app at a time. The possible values are:

- `MULTIPLE` (Default)
- `SINGLE`

Constructing a `PublicClientApplication` using an account mode that doesn't match this setting will result in an exception.

For more information about the differences between single and multiple accounts, see [Single and multiple account apps](#).

browser_safelist

An allow-list of browsers that are compatible with MSAL. These browsers correctly handle redirects to custom intents. You can add to this list. The default is provided in the default configuration shown below.

The default MSAL configuration file

The default MSAL configuration that ships with MSAL is shown below. You can see the latest version on [GitHub](#).

This configuration is supplemented by values that you provide. The values you provide override the defaults.

```
{
  "authorities": [
    {
      "type": "AAD",
      "audience": {
        "type": "AzureADandPersonalMicrosoftAccount"
      },
      "default": true
    }
  ],
  "authorization_user_agent": "DEFAULT",
  "multiple_clouds_supported": false,
  "broker_redirect_uri_registered": false,
  "http": {
```

```
"connect_timeout": 10000,
"read_timeout": 30000
},
"logging": {
    "pii_enabled": false,
    "log_level": "WARNING",
    "logcat_enabled": false
},
"shared_device_mode_supported": false,
"account_mode": "MULTIPLE",
"browser_safelist": [
    {
        "browser_package_name": "com.android.chrome",
        "browser_signature_hashes": [
            "7fmduHKTdHHrlMvldlEqAI1Sfi1tl35bxj10XN5Ve8c4lU6URVu4xtSHc3BVZxS6WWJnxMDhIfQN0N0K2NDJg=="
        ],
        "browser_use_customTab" : true,
        "browser_version_lower_bound": "45"
    },
    {
        "browser_package_name": "com.android.chrome",
        "browser_signature_hashes": [
            "7fmduHKTdHHrlMvldlEqAI1Sfi1tl35bxj10XN5Ve8c4lU6URVu4xtSHc3BVZxS6WWJnxMDhIfQN0N0K2NDJg=="
        ],
        "browser_use_customTab" : false
    },
    {
        "browser_package_name": "org.mozilla.firefox",
        "browser_signature_hashes": [
            "2gCe6pR_AO_Q2Vu8Iep-4AsiKNnUHQxu0FaDHO_qa178GByKybdT_BuE8_dYk99G5Uvx_gdONXA002EaXidpVQ=="
        ],
        "browser_use_customTab" : false
    },
    {
        "browser_package_name": "org.mozilla.firefox",
        "browser_signature_hashes": [
            "2gCe6pR_AO_Q2Vu8Iep-4AsiKNnUHQxu0FaDHO_qa178GByKybdT_BuE8_dYk99G5Uvx_gdONXA002EaXidpVQ=="
        ],
        "browser_use_customTab" : true,
        "browser_version_lower_bound": "57"
    },
    {
        "browser_package_name": "com.sec.android.app.sbrowser",
        "browser_signature_hashes": [
            "ABi2fbt8vkzj7SJ8aD5jc4xJFTDFntdkMrYXL3itsvqY1QIw-dZozdop5rgKNxjbrQAd5nntAGpgh9w8401Xgg=="
        ],
        "browser_use_customTab" : true,
        "browser_version_lower_bound": "4.0"
    },
    {
        "browser_package_name": "com.sec.android.app.sbrowser",
        "browser_signature_hashes": [
            "ABi2fbt8vkzj7SJ8aD5jc4xJFTDFntdkMrYXL3itsvqY1QIw-dZozdop5rgKNxjbrQAd5nntAGpgh9w8401Xgg=="
        ],
        "browser_use_customTab" : false
    },
    {
        "browser_package_name": "com.cloudmosa.puffinFree",
        "browser_signature_hashes": [
            "1WqG8SoK2WvE4NTYgr2550TRjhjhxT-7DWxu6C_o6Gr0LK6xzG67Hq7GCGDjkAFRCOChlo2XUUgllRAYu3Mn8Ag=="
        ],
        "browser_use_customTab" : false
    },
    {
        "browser_package_name": "com.duckduckgo.mobile.android",
        "browser_signature_hashes": [
            "S5Av4cfEycCvIvKPpKGjyCuAE5gZ8y60-knFfGkAEIZWPr9lU5kA7i0AlSZxaJe08s0ruDvuEzFYlmH-jAi4Q=="
        ],
        "browser_use_customTab" : false
    }
]
```

```

    "browser_package_name": "com.explore.web.browser",
    "browser_signature_hashes": [
        "BzDzBVSAwah8f_A0MYJCP0kt0eb7WcIEw6Udn7VLcizjoU3wxAzVisCm6bw7uTs4WpMfBEJYf0nDgzTYvYHCag=="
    ],
    "browser_use_customTab" : false
},

{
    "browser_package_name": "com.ksmobile.cb",
    "browser_signature_hashes": [
        "1FDYx1Rwc7_XUn4K1fQk2k1XLufRyuGHLa3a7rNjqQMkMaxZueQfxukVTvA7yKKp3Md3XUeeDSWGIcRy7nouw=="
    ],
    "browser_use_customTab" : false
},

{
    "browser_package_name": "com.microsoft.emmx",
    "browser_signature_hashes": [
        "Ivy-Rk6ztai_IudfbUrSHugzRqAtHWs1FvHT0PTvLMsEKLUIgv7ZZbVxygWy_M5mOPpfjZrd3v0x3t-cA6fVQ=="
    ],
    "browser_use_customTab" : false
},

{
    "browser_package_name": "com.opera.browser",
    "browser_signature_hashes": [
        "FIJ3IIeqB7V0qHpRNEpYNkhEGA_eJaf7ntca-0a_6Feev3UkgngpuTNV31JdAmpEFPGNPo0RHqd1U0k-3jWJWw=="
    ],
    "browser_use_customTab" : false
},

{
    "browser_package_name": "com.opera.mini.native",
    "browser_signature_hashes": [
        "TOTyHs086iGIEdxrX_24aAewTzxV7Wbi6niS2ZrpPhLkjuzPAh1c3NQ_U4Lx1KdgyhQE4BiS36MifP6LbmmUYQ=="
    ],
    "browser_use_customTab" : false
},

{
    "browser_package_name": "mobi.mgeek.TunnyBrowser",
    "browser_signature_hashes": [
        "RMVoXuK1sfJZuGZ8onG1yhMc-sKiAV2NiB_GZfdNlN8XJ78XEE2wPM6LnQiyltF25GkHiPN2iKQiGwa02bkyyQ=="
    ],
    "browser_use_customTab" : false
},

{
    "browser_package_name": "org.mozilla.focus",
    "browser_signature_hashes": [
        "L72dT-stFqomSY7sYySrgBJ3VYKbipMZapmUXFTNqOzN_dekT5wdBACJkpz0C6P0yx5EmZ5IciI93Q0hq0oYA=="
    ],
    "browser_use_customTab" : false
}
]
}

```

Example basic configuration

The following example illustrates a basic configuration that specifies the client ID, redirect URI, whether a broker redirect is registered, and a list of authorities.

```
{  
    "client_id" : "4b0db8c2-9f26-4417-8bde-3f0e3656f8e0",  
    "redirect_uri" : "msauth://com.microsoft.identity.client.sample.local/1wIqXsqBj7w%2Bh11ZifsqnwgYKrY%3D",  
    "broker_redirect_uri_registered": true,  
    "authorities" : [  
        {  
            "type": "AAD",  
            "audience": {  
                "type": "AzureADandPersonalMicrosoftAccount"  
            }  
            "default": true  
        }  
    ]  
}
```

How to use a configuration file

1. Create a configuration file. We recommend that you create your custom configuration file in `res/raw/auth_config.json`. But you can put it anywhere that you wish.
2. Tell MSAL where to look for your configuration when you construct the `PublicClientApplication`. For example:

```
//On Worker Thread  
IMultipleAccountPublicClientApplication sampleApp = null;  
sampleApp = new  
PublicClientApplication.createMultipleAccountPublicClientApplication(getApplicationContext(),  
R.raw.auth_config);
```

Authentication and authorization error codes

8/30/2019 • 21 minutes to read • [Edit Online](#)

Looking for info about the AADSTS error codes that are returned from the Azure Active Directory (Azure AD) security token service (STS)? Read this document to find AADSTS error descriptions, fixes, and some suggested workarounds.

NOTE

This information is preliminary and subject to change. Have a question or can't find what you're looking for? Create a GitHub issue or see [Support and help options for developers](#) to learn about other ways you can get help and support.

This documentation is provided for developer and admin guidance, but should never be used by the client itself. Error codes are subject to change at any time in order to provide more granular error messages that are intended to help the developer while building their application. Apps that take a dependency on text or error code numbers will be broken over time.

Lookup current error code information

Error codes and messages are subject to change. For the most current info, take a look at the <https://login.microsoftonline.com/error> page to find AADSTS error descriptions, fixes, and some suggested workarounds.

Search on the numeric part of the returned error code. For example, if you received the error code "AADSTS16000" then do a search in <https://login.microsoftonline.com/error> for "16000". You can also link directly to a specific error by adding the error code number to the URL: <https://login.microsoftonline.com/error?code=16000>.

AADSTS error codes

ERROR	DESCRIPTION
AADSTS16000	SelectUserAccount - This is an interrupt thrown by Azure AD, which results in UI that allows the user to select from among multiple valid SSO sessions. This error is fairly common and may be returned to the application if <code>prompt=none</code> is specified.
AADSTS16001	UserAccountSelectionInvalid - You'll see this error if the user clicks on a tile that the session select logic has rejected. When triggered, this error allows the user to recover by picking from an updated list of tiles/sessions, or by choosing another account. This error can occur because of a code defect or race condition.
AADSTS16002	AppSessionSelectionInvalid - The app-specified SID requirement was not met.
AADSTS16003	SsoUserAccountNotFoundInResourceTenant - Indicates that the user hasn't been explicitly added to the tenant.

ERROR	DESCRIPTION
AADSTS17003	CredentialKeyProvisioningFailed - Azure AD can't provision the user key.
AADSTS20001	WsFedSignInResponseError - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS20012	WsFedMessageInvalid - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS20033	FedMetadataInvalidTenantName - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS40008	OAuth2IdPUnretryableServerError - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS40009	OAuth2IdPRefreshTokenRedemptionUserError - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS40010	OAuth2IdPRetryableServerError - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS40015	OAuth2IdPAuthCodeRedemptionUserError - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS50000	TokenIssuanceError - There's an issue with the sign-in service. Open a support ticket to resolve this issue.
AADSTS50001	InvalidResource - The resource is disabled or does not exist. Check your app's code to ensure that you have specified the exact resource URL for the resource you are trying to access.
AADSTS50002	NotAllowedTenant - Sign-in failed because of a restricted proxy access on the tenant. If it's your own tenant policy, you can change your restricted tenant settings to fix this issue.
AADSTS50003	MissingSigningKey - Sign-in failed because of a missing signing key or certificate. This might be because there was no signing key configured in the app. Check out the resolutions outlined at https://docs.microsoft.com/azure/active-directory/application-sign-in-problem-federated-sso-gallery#certificate-or-key-not-configured . If you still see issues, contact the app owner or an app admin.
AADSTS50005	DevicePolicyError - User tried to log in to a device from a platform that's currently not supported through Conditional Access policy.
AADSTS50006	InvalidSignature - Signature verification failed because of an invalid signature.

ERROR	DESCRIPTION
AADSTS50007	PartnerEncryptionCertificateMissing - The partner encryption certificate was not found for this app. Open a support ticket with Microsoft to get this fixed.
AADSTS50008	InvalidSamlToken - SAML assertion is missing or misconfigured in the token. Contact your federation provider.
AADSTS50010	AudienceUriValidationFailed - Audience URI validation for the app failed since no token audiences were configured.
AADSTS50011	InvalidReplyTo - The reply address is missing, misconfigured, or does not match reply addresses configured for the app. As a resolution ensure to add this missing reply address to the Azure Active Directory application or have someone with the permissions to manage your application in Active Directory do this for you.
AADSTS50012	<p>AuthenticationFailed - Authentication failed for one of the following reasons:</p> <ul style="list-style-type: none"> • The subject name of the signing certificate is not authorized • A matching trusted authority policy was not found for the authorized subject name • The certificate chain is not valid • The signing certificate is not valid • Policy is not configured on the tenant • Thumbprint of the signing certificate is not authorized • Client assertion contains an invalid signature
AADSTS50013	InvalidAssertion - Assertion is invalid because of various reasons - The token issuer doesn't match the api version within its valid time range -expired -malformed - Refresh token in the assertion is not a primary refresh token.
AADSTS50014	GuestUserInPendingState - The user's redemption is in a pending state. The guest user account is not fully created yet.
AADSTS50015	ViralUserLegalAgeConsentRequiredState - The user requires legal age group consent.
AADSTS50017	<p>CertificateValidationFailed - Certification validation failed, reasons for the following reasons:</p> <ul style="list-style-type: none"> • Cannot find issuing certificate in trusted certificates list • Unable to find expected CrlSegment • Cannot find issuing certificate in trusted certificates list • Delta CRL distribution point is configured without a corresponding CRL distribution point • Unable to retrieve valid CRL segments because of a timeout issue • Unable to download CRL <p>Contact the tenant admin.</p>
AADSTS50020	UserUnauthorized - Users are unauthorized to call this endpoint.

ERROR	DESCRIPTION
AADSTS50027	<p>InvalidJwtToken - Invalid JWT token because of the following reasons:</p> <ul style="list-style-type: none"> • doesn't contain nonce claim, sub claim • subject identifier mismatch • duplicate claim in idToken claims • unexpected issuer • unexpected audience • not within its valid time range • token format is not proper • External ID token from issuer failed signature verification.
AADSTS50029	Invalid URI - domain name contains invalid characters. Contact the tenant admin.
AADSTS50032	WeakRsaKey - Indicates the erroneous user attempt to use a weak RSA key.
AADSTS50033	RetryableError - Indicates a transient error not related to the database operations.
AADSTS50034	UserAccountNotFound - To sign into this application, the account must be added to the directory.
AADSTS50042	UnableToGeneratePairwiseIdentifierWithMissingSalt - The salt required to generate a pairwise identifier is missing in principle. Contact the tenant admin.
AADSTS50043	UnableToGeneratePairwiseIdentifierWithMultipleSalts
AADSTS50048	SubjectMismatchedIssuer - Subject mismatches Issuer claim in the client assertion. Contact the tenant admin.
AADSTS50049	NoSuchInstanceForDiscovery - Unknown or invalid instance.
AADSTS50050	MalformedDiscoveryRequest - The request is malformed.
AADSTS50053	IdsLocked - The account is locked because the user tried to sign in too many times with an incorrect user ID or password.
AADSTS50055	InvalidPasswordExpiredPassword - The password is expired.
AADSTS50056	Invalid or null password - Password does not exist in store for this user.
AADSTS50057	UserDisabled - The user account is disabled. The account has been disabled by an administrator.

ERROR	DESCRIPTION
AADSTS50058	<p>UserInformationNotProvided - This means that a user is not signed in. This is a common error that's expected when a user is unauthenticated and has not yet signed in.</p> <p>If this error is encouraged in an SSO context where the user has previously signed in, this means that the SSO session was either not found or invalid.</p> <p>This error may be returned to the application if prompt=none is specified.</p>
AADSTS50059	<p>MissingTenantRealmAndNoUserInformationProvided - Tenant-identifying information was not found in either the request or implied by any provided credentials. The user can contact the tenant admin to help resolve the issue.</p>
AADSTS50061	<p>SignoutInvalidRequest - The sign-out request is invalid.</p>
AADSTS50064	<p>CredentialAuthenticationError - Credential validation on username or password has failed.</p>
AADSTS50068	<p>SignoutInitiatorNotParticipant - Signout has failed. The app that initiated signout is not a participant in the current session.</p>
AADSTS50070	<p>SignoutUnknownSessionIdentifier - Signout has failed. The signout request specified a name identifier that didn't match the existing session(s).</p>
AADSTS50071	<p>SignoutMessageExpired - The logout request has expired.</p>
AADSTS50072	<p>UserStrongAuthEnrollmentRequiredInterrupt - User needs to enroll for second factor authentication (interactive).</p>
AADSTS50074	<p>UserStrongAuthClientAuthNRequiredInterrupt - Strong authentication is required and the user did not pass the MFA challenge.</p>
AADSTS50076	<p>UserStrongAuthClientAuthNRequired - Due to a configuration change made by the admin, or because you moved to a new location, the user must use multi-factor authentication to access the resource. Retry with a new authorize request for the resource.</p>
AADSTS50079	<p>UserStrongAuthEnrollmentRequired - Due to a configuration change made by the administrator, or because the user moved to a new location, the user is required to use multi-factor authentication.</p>
AADSTS50085	<p>Refresh token needs social IDP login. Have user try signing-in again with username -password</p>
AADSTS50086	<p>SasNonRetryableError</p>
AADSTS50087	<p>SasRetryableError - The service is temporarily unavailable. Try again.</p>

ERROR	DESCRIPTION
AADSTS50089	Flow token expired - Authentication Failed. Have the user try signing-in again with username -password.
AADSTS50097	DeviceAuthenticationRequired - Device authentication is required.
AADSTS50099	PKeyAuthInvalidJwtUnauthorized - The JWT signature is invalid.
AADSTS50105	EntitlementGrantsNotFound - The signed in user is not assigned to a role for the signed in app. Assign the user to the app. For more information: https://docs.microsoft.com/azure/active-directory/application-sign-in-problem-federated-sso-gallery#user-not-assigned-a-role .
AADSTS50107	InvalidRealmUri - The requested federation realm object does not exist. Contact the tenant admin.
AADSTS50120	ThresholdJwtInvalidJwtFormat - Issue with JWT header. Contact the tenant admin.
AADSTS50124	ClaimsTransformationInvalidInputParameter - Claims Transformation contains invalid input parameter. Contact the tenant admin to update the policy.
AADSTS50125	PasswordResetRegistrationRequiredInterrupt - Sign-in was interrupted because of a password reset or password registration entry.
AADSTS50126	InvalidUserNameOrPassword - Error validating credentials due to invalid username or password.
AADSTS50127	BrokerAppNotInstalled - User needs to install a broker app to gain access to this content.
AADSTS50128	Invalid domain name - No tenant-identifying information found in either the request or implied by any provided credentials.
AADSTS50129	DeviceIsNotWorkplaceJoined - Workplace join is required to register the device.
AADSTS50131	ConditionalAccessFailed - Indicates various Conditional Access errors such as bad Windows device state, request blocked due to suspicious activity, access policy, or security policy decisions.
AADSTS50132	SsoArtifactInvalidOrExpired - The session is not valid due to password expiration or recent password change.
AADSTS50133	SsoArtifactRevoked - The session is not valid due to password expiration or recent password change.

ERROR	DESCRIPTION
AADSTS50134	DeviceFlowAuthorizeWrongDatacenter - Wrong data center. To authorize a request that was initiated by an app in the OAuth 2.0 device flow, the authorizing party must be in the same data center where the original request resides.
AADSTS50135	PasswordChangeCompromisedPassword - Password change is required due to account risk.
AADSTS50136	RedirectMsaSessionToApp - Single MSA session detected.
AADSTS50139	SessionMissingMsaOAuth2RefreshToken - The session is invalid due to a missing external refresh token.
AADSTS50140	KmsiInterrupt - This error occurred due to "Keep me signed in" interrupt when the user was signing-in. Open a support ticket with Correlation ID, Request ID, and Error code to get more details.
AADSTS50143	Session mismatch - Session is invalid because user tenant does not match the domain hint due to different resource. Open a support ticket with Correlation ID, Request ID, and Error code to get more details.
AADSTS50144	InvalidPasswordExpiredOnPremPassword - User's Active Directory password has expired. Generate a new password for the user or have the user use the self-service reset tool to reset their password.
AADSTS50146	MissingCustomSigningKey - This app is required to be configured with an app-specific signing key. It is either not configured with one, or the key has expired or is not yet valid.
AADSTS50147	MissingCodeChallenge - The size of the code challenge parameter is not valid.
AADSTS50155	DeviceAuthenticationFailed - Device authentication failed for this user.
AADSTS50158	ExternalSecurityChallenge - External security challenge was not satisfied.
AADSTS50161	InvalidExternalSecurityChallengeConfiguration - Claims sent by external provider is not enough or Missing claim requested to external provider.
AADSTS50166	ExternalClaimsProviderThrottled - Failed to send the request to the claims provider.
AADSTS50168	ChromeBrowserSsoInterruptRequired - The client is capable of obtaining an SSO token through the Windows 10 Accounts extension, but the token was not found in the request or the supplied token was expired.
AADSTS50169	InvalidRequestBadRealm - The realm is not a configured realm of the current service namespace.

ERROR	DESCRIPTION
AADSTS50170	MissingExternalClaimsProviderMapping - The external controls mapping is missing.
AADSTS50177	ExternalChallengeNotSupportedForPassthroughUsers - External challenge is not supported for passthrough users.
AADSTS50178	SessionControlNotSupportedForPassthroughUsers - Session control is not supported for passthrough users.
AADSTS50180	WindowsIntegratedAuthMissing - Integrated Windows authentication is needed. Enable the tenant for Seamless SSO.
AADSTS50187	DeviceInformationNotProvided - The service failed to perform device authentication.
AADSTS51000	RequiredFeatureNotEnabled - The feature is disabled.
AADSTS51001	DomainHintMustbePresent - Domain hint must be present with on-premises security identifier or on-premises UPN.
AADSTS51004	UserAccountNotInDirectory - The user account doesn't exist in the directory.
AADSTS51005	TemporaryRedirect - Equivalent to HTTP status 307, which indicates that the requested information is located at the URI specified in the location header. When you receive this status, follow the location header associated with the response. When the original request method was POST, the redirected request will also use the POST method.
AADSTS51006	ForceReauthDueToInsufficientAuth - Integrated Windows authentication is needed. User logged in using a session token that is missing the Integrated Windows authentication claim. Request the user to log in again.
AADSTS52004	DelegationDoesNotExistForLinkedIn - The user has not provided consent for access to LinkedIn resources.
AADSTS53000	DeviceNotCompliant - Conditional Access policy requires a compliant device, and the device is not compliant. The user must enroll their device with an approved MDM provider like Intune.
AADSTS53001	DeviceNotDomainJoined - Conditional Access policy requires a domain joined device, and the device is not domain joined. Have the user use a domain joined device.
AADSTS53002	ApplicationUsedIsNotAnApprovedApp - The app used is not an approved app for Conditional Access. User needs to use one of the apps from the list of approved apps to use in order to get access.
AADSTS53003	BlockedByConditionalAccess - Access has been blocked by Conditional Access policies. The access policy does not allow token issuance.

ERROR	DESCRIPTION
AADSTS53004	ProofUpBlockedDueToRisk - User needs to complete the multi-factor authentication registration process before accessing this content. User should register for multi-factor authentication.
AADSTS54000	MinorUserBlockedLegalAgeGroupRule
AADSTS65001	DelegationDoesNotExist - The user or administrator has not consented to use the application with ID X. Send an interactive authorization request for this user and resource.
AADSTS65004	UserDeclinedConsent - User declined to consent to access the app. Have the user retry the sign-in and consent to the app
AADSTS65005	MisconfiguredApplication - The app required resource access list does not contain apps discoverable by the resource or The client app has requested access to resource, which was not specified in its required resource access list or Graph service returned bad request or resource not found. If the app supports SAML, you may have configured the app with the wrong Identifier (Entity). Try out the resolution listed for SAML using the link below: https://docs.microsoft.com/azure/active-directory/application-sign-in-problem-federated-sso-gallery#no-resource-in-requiredresourceaccess-list
AADSTS67003	ActorNotValidServiceIdentity
AADSTS70000	InvalidGrant - Authentication failed. The refresh token is not valid. Error may be due to the following reasons: <ul style="list-style-type: none"> • Token binding header is empty • Token binding hash does not match
AADSTS70001	UnauthorizedClient - The application is disabled.
AADSTS70002	InvalidClient - Error validating the credentials. The specified client_secret does not match the expected value for this client. Correct the client_secret and try again. For more info, see Use the authorization code to request an access token .
AADSTS70003	UnsupportedGrantType - The app returned an unsupported grant type.
AADSTS70004	InvalidRedirectUri - The app returned an invalid redirect URI. The redirect address specified by the client does not match any configured addresses or any addresses on the OIDC approve list.
AADSTS70005	UnsupportedResponseType - The app returned an unsupported response type due to the following reasons: <ul style="list-style-type: none"> • response type 'token' is not enabled for the app • response type 'id_token' requires the 'OpenID' scope - contains an unsupported OAuth parameter value in the encoded wctx

ERROR	DESCRIPTION
AADSTS70007	UnsupportedResponseMode - The app returned an unsupported value of <code>response_mode</code> when requesting a token.
AADSTS70008	ExpiredOrRevokedGrant - The refresh token has expired due to inactivity. The token was issued on XXX and was inactive for a certain amount of time.
AADSTS70011	InvalidScope - The scope requested by the app is invalid.
AADSTS70012	MsaServerError - A server error occurred while authenticating an MSA (consumer) user. Try again. If it continues to fail, open a support ticket
AADSTS70016	AuthorizationPending - OAuth 2.0 device flow error. Authorization is pending. The device will retry polling the request.
AADSTS70018	BadVerificationCode - Invalid verification code due to User typing in wrong user code for device code flow. Authorization is not approved.
AADSTS70019	CodeExpired - Verification code expired. Have the user retry the sign-in.
AADSTS75001	BindingSerializationError - An error occurred during SAML message binding.
AADSTS75003	UnsupportedBindingError - The app returned an error related to unsupported binding (SAML protocol response cannot be sent via bindings other than HTTP POST).
AADSTS75005	Saml2MessageInvalid - Azure AD doesn't support the SAML request sent by the app for SSO.
AADSTS75008	RequestDeniedError - The request from the app was denied since the SAML request had an unexpected destination.
AADSTS75011	NoMatchedAuthnContextInOutputClaims - The authentication method by which the user authenticated with the service doesn't match requested authentication method.
AADSTS75016	Saml2AuthenticationRequestInvalidNameIDPolicy - SAML2 Authentication Request has invalid NameIdPolicy.
AADSTS80001	OnPremiseStoreIsNotAvailable - The Authentication Agent is unable to connect to Active Directory. Make sure that agent servers are members of the same AD forest as the users whose passwords need to be validated and they are able to connect to Active Directory.
AADSTS80002	OnPremisePasswordValidatorRequestTimedout - Password validation request timed out. Make sure that Active Directory is available and responding to requests from the agents.

ERROR	DESCRIPTION
AADSTS80005	OnPremisePasswordValidatorUnpredictableWebException - An unknown error occurred while processing the response from the Authentication Agent. Retry the request. If it continues to fail, open a support ticket to get more details on the error.
AADSTS80007	OnPremisePasswordValidatorErrorOccurredOnPrem - The Authentication Agent is unable to validate user's password. Check the agent logs for more info and verify that Active Directory is operating as expected.
AADSTS80010	OnPremisePasswordValidationEncryptionException - The Authentication Agent is unable to decrypt password.
AADSTS80012	OnPremisePasswordValidationAccountLogonInvalidHours - The users attempted to log on outside of the allowed hours (this is specified in AD).
AADSTS80013	OnPremisePasswordValidationTimeSkew - The authentication attempt could not be completed due to time skew between the machine running the authentication agent and AD. Fix time sync issues.
AADSTS81004	DesktopSsoIdentityInTicketIsNotAuthenticated - Kerberos authentication attempt failed.
AADSTS81005	DesktopSsoAuthenticationPackageNotSupported - The authentication package is not supported.
AADSTS81006	DesktopSsoNoAuthorizationHeader - No authorization header was found.
AADSTS81007	DesktopSsoTenantIsNotOptIn - The tenant is not enabled for Seamless SSO.
AADSTS81009	DesktopSsoAuthorizationHeaderValueWithBadFormat - Unable to validate user's Kerberos ticket.
AADSTS81010	DesktopSsoAuthTokenInvalid - Seamless SSO failed because the user's Kerberos ticket has expired or is invalid.
AADSTS81011	DesktopSsoLookupUserBySidFailed - Unable to find user object based on information in the user's Kerberos ticket.
AADSTS81012	DesktopSsoMismatchBetweenTokenUpnAndChosenUpn - The user trying to sign in to Azure AD is different from the user signed into the device.
AADSTS90002	InvalidTenantName - The tenant name wasn't found in the data store. Check to make sure you have the correct tenant ID.
AADSTS90004	InvalidRequestFormat - The request is not properly formatted.

ERROR	DESCRIPTION
AADSTS90005	InvalidRequestWithMultipleRequirements - Unable to complete the request. The request is not valid because the identifier and login hint can't be used together.
AADSTS90006	ExternalServerRetryableError - The service is temporarily unavailable.
AADSTS90007	InvalidSessionId - Bad request. The passed session ID can't be parsed.
AADSTS90008	TokenForItselfRequiresGraphPermission - The user or administrator hasn't consented to use the application. At the minimum, the application requires access to Azure AD by specifying the sign-in and read user profile permission.
AADSTS90009	TokenForItselfMissingIdenticalAppIdentifier - The application is requesting a token for itself. This scenario is supported only if the resource that's specified is using the GUID-based application ID.
AADSTS90010	NotSupported - Unable to create the algorithm.
AADSTS90012	RequestTimeout - The requested has timed out.
AADSTS90013	InvalidUserInput - The input from the user is not valid.
AADSTS90014	MissingRequiredField - This error code may appear in various cases when an expected field is not present in the credential.
AADSTS90015	QueryStringTooLong - The query string is too long.
AADSTS90016	MissingRequiredClaim - The access token isn't valid. The required claim is missing.
AADSTS90019	MissingTenantRealm - Azure AD was unable to determine the tenant identifier from the request.
AADSTS90022	AuthenticatedInvalidPrincipalNameFormat - The principal name format is not valid, or does not meet the expected <code>name[/host][@realm]</code> format. The principal name is required, host and realm are optional and may be set to null.
AADSTS90023	InvalidRequest - The authentication service request is not valid.
AADSTS9002313	InvalidRequest - Request is malformed or invalid. - The issue here is because there was something wrong with the request to a certain endpoint. The suggestion to this issue is to get a fiddler trace of the error occurring and looking to see if the request is actually properly formatted or not.
AADSTS90024	RequestBudgetExceededError - A transient error has occurred. Try again.

ERROR	DESCRIPTION
AADSTS90033	MsodsServiceUnavailable - The Microsoft Online Directory Service (MSODS) is not available.
AADSTS90036	MsodsServiceUnretryableFailure - An unexpected, non-retryable error from the WCF service hosted by MSODS has occurred. Open a support ticket to get more details on the error.
AADSTS90038	NationalCloudTenantRedirection - The specified tenant 'Y' belongs to the National Cloud 'X'. Current cloud instance 'Z' does not federate with X. A cloud redirect error is returned.
AADSTS90043	NationalCloudAuthCodeRedirection - The feature is disabled.
AADSTS90051	InvalidNationalCloudId - The national cloud identifier contains an invalid cloud identifier.
AADSTS90055	TenantThrottlingError - There are too many incoming requests. This exception is thrown for blocked tenants.
AADSTS90056	<p>BadResourceRequest - To redeem the code for an access token, the app should send a POST request to the <code>/token</code> endpoint. Also, prior to this, you should provide an authorization code and send it in the POST request to the <code>/token</code> endpoint. Refer to this article for an overview of OAuth 2.0 authorization code flow: https://docs.microsoft.com/azure/active-directory/develop/active-directory-protocols-oauth-code.</p> <p>Direct the user to the <code>/authorize</code> endpoint, which will return an <code>authorization_code</code>. By posting a request to the <code>/token</code> endpoint, the user gets the access token. Log in the Azure portal, and check App registrations > Endpoints to confirm that the two endpoints were configured correctly.</p>
AADSTS90072	PassThroughUserMfaError - The external account that the user signs in with doesn't exist on the tenant that they signed into; so the user can't satisfy the MFA requirements for the tenant. The account must be added as an external user in the tenant first. Sign out and sign in with a different Azure AD user account.
AADSTS90081	OrgIdWsFederationMessageInvalid - An error occurred when the service tried to process a WS-Federation message. The message is not valid.
AADSTS90082	OrgIdWsFederationNotSupported - The selected authentication policy for the request isn't currently supported.
AADSTS90084	OrgIdWsFederationGuestNotAllowed - Guest accounts aren't allowed for this site.
AADSTS90085	OrgIdWsFederationSltRedemptionFailed - The service is unable to issue a token because the company object hasn't been provisioned yet.

ERROR	DESCRIPTION
AADSTS90086	OrgIdWsTrustDaTokenExpired - The user DA token is expired.
AADSTS90087	OrgIdWsFederationMessageCreationFromUriFailed - An error occurred while creating the WS-Federation message from the URI.
AADSTS90090	GraphRetryableError - The service is temporarily unavailable.
AADSTS90091	GraphServiceUnreachable
AADSTS90092	GraphNonRetryableError
AADSTS90093	GraphUserUnauthorized - Graph returned with a forbidden error code for the request.
AADSTS90094	AdminConsentRequired - Administrator consent is required.
AADSTS90100	InvalidRequestParameter - The parameter is empty or not valid.
AADSTS90102	AADSTS901002: The 'resource' request parameter is not supported.
AADSTS90101	InvalidEmailAddress - The supplied data isn't a valid email address. The email address must be in the format <code>someone@example.com</code> .
AADSTS90102	InvalidUriParameter - The value must be a valid absolute URI.
AADSTS90107	InvalidXml - The request is not valid. Make sure your data doesn't have invalid characters.
AADSTS90114	InvalidExpiryDate - The bulk token expiration timestamp will cause an expired token to be issued.
AADSTS90117	InvalidRequestInput
AADSTS90119	InvalidUserCode - The user code is null or empty.
AADSTS90120	InvalidDeviceFlowRequest - The request was already authorized or declined.
AADSTS90121	InvalidEmptyRequest - Invalid empty request.
AADSTS90123	IdentityProviderAccessDenied - The token can't be issued because the identity or claim issuance provider denied the request.
AADSTS90124	V1ResourceV2GlobalEndpointNotSupported - The resource is not supported over the <code>/common</code> or <code>/consumers</code> endpoints. Use the <code>/organizations</code> or tenant-specific endpoint instead.

ERROR	DESCRIPTION
AADSTS90125	DebugModeEnrollTenantNotFound - The user isn't in the system. Make sure you entered the user name correctly.
AADSTS90126	DebugModeEnrollTenantNotInferred - The user type is not supported on this endpoint. The system can't infer the user's tenant from the user name.
AADSTS90130	NonConvergedAppV2GlobalEndpointNotSupported - The application is not supported over the <code>/common</code> or <code>/consumers</code> endpoints. Use the <code>/organizations</code> or tenant-specific endpoint instead.
AADSTS120000	PasswordChangeIncorrectCurrentPassword
AADSTS120002	PasswordChangeInvalidNewPasswordWeak
AADSTS120003	PasswordChangeInvalidNewPasswordContainsMemberName
AADSTS120004	PasswordChangeOnPremComplexity
AADSTS120005	PasswordChangeOnPremSuccessCloudFail
AADSTS120008	PasswordChangeAsyncJobStateTerminated - A non-retryable error has occurred.
AADSTS120011	PasswordChangeAsyncUpnInferenceFailed
AADSTS120012	PasswordChangeNeedsToHappenOnPrem
AADSTS120013	PasswordChangeOnPremisesConnectivityFailure
AADSTS120014	PasswordChangeOnPremUserAccountLockedOutOrDisabled
AADSTS120015	PasswordChangeADAdminActionRequired
AADSTS120016	PasswordChangeUserNotFoundBySspr
AADSTS120018	PasswordChangePasswordDoesnotComplyFuzzyPolicy
AADSTS120020	PasswordChangeFailure
AADSTS120021	PartnerServiceSsprInternalServiceError
AADSTS130004	NgcKeyNotFound - The user principal doesn't have the NGC ID key configured.
AADSTS130005	NgcInvalidSignature - NGC key signature verified failed.
AADSTS130006	NgcTransportKeyNotFound - The NGC transport key isn't configured on the device.
AADSTS130007	NgcDeviceIsDisabled - The device is disabled.

ERROR	DESCRIPTION
AADSTS130008	NgcDeviceNotFound - The device referenced by the NGC key wasn't found.
AADSTS135010	KeyNotFound
AADSTS140000	InvalidRequestNonce - Request nonce is not provided.
AADSTS140001	InvalidSessionKey - The session key is not valid.
AADSTS165900	InvalidApiRequest - Invalid request.
AADSTS220450	UnsupportedAndroidWebViewVersion - The Chrome WebView version is not supported.
AADSTS220501	InvalidCrlDownload
AADSTS221000	DeviceOnlyTokensNotSupportedByResource - The resource is not configured to accept device-only tokens.
AADSTS240001	BulkAADJTokenUnauthorized - The user isn't authorized to register devices in Azure AD.
AADSTS240002	RequiredClaimIsMissing - The id_token can't be used as <code>urn:ietf:params:oauth:grant-type:jwt-bearer</code> grant.
AADSTS530032	BlockedByConditionalAccessOnSecurityPolicy - The tenant admin has configured a security policy that blocks this request. Check the security policies that are defined on the tenant level to determine if your request meets the policy requirements.
AADSTS700016	UnauthorizedClient_DoesNotMatchRequest - The application wasn't found in the directory/tenant. 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 misconfigured the identifier value for the application or sent your authentication request to the wrong tenant.
AADSTS700020	InteractionRequired - The access grant requires interaction.
AADSTS700022	InvalidMultipleResourcesScope - The provided value for the input parameter scope isn't valid because it contains more than one resource.
AADSTS700023	InvalidResourcelessScope - The provided value for the input parameter scope isn't valid when request an access token.
AADSTS100000	UserNotBoundError - The Bind API requires the Azure AD user to also authenticate with an external IDP, which hasn't happened yet.
AADSTS100002	BindCompleteInterruptError - The bind completed successfully, but the user must be informed.

ERROR	DESCRIPTION
AADSTS7000112	UnauthorizedClientApplicationDisabled - The application is disabled.

Next steps

- Have a question or can't find what you're looking for? Create a GitHub issue or see [Support and help options for developers](#) to learn about other ways you can get help and support.

What's new for authentication?

8/28/2019 • 7 minutes to read • [Edit Online](#)

Get notified about updates to this page. Just add [this URL](#) to your RSS feed reader.

The authentication system alters and adds features on an ongoing basis to improve security and standards compliance. To stay up-to-date with the most recent developments, this article provides you with information about the following details:

- Latest features
- Known issues
- Protocol changes
- Deprecated functionality

TIP

This page is updated regularly, so visit often. Unless otherwise noted, these changes are only put in place for newly registered applications.

Upcoming changes

September 2019: Additional enforcement of POST semantics according to URL parsing rules - duplicate parameters will trigger an error and **BOM** ignored.

August 2019

POST form semantics will be enforced more strictly - spaces and quotes will be ignored

Effective date: September 2, 2019

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: Anywhere POST is used ([client credentials](#), [authorization code redemption](#), [ROPC](#), [OBO](#), and [refresh token redemption](#))

Starting the week of 9/2, authentication requests that use the POST method will be validated using stricter HTTP standards. Specifically, spaces and double-quotes ("") will no longer be removed from request form values. These changes are not expected to break any existing clients, and will ensure that requests sent to Azure AD are reliably handled every time. In the future (see above) we plan to additionally reject duplicate parameters and ignore the BOM within requests.

Example:

Today, `?e= "f"&g=h` is parsed identically as `?e=f&g=h` - so `e == f`. With this change, it would now be parsed so that `e == "f"` - this is unlikely to be a valid argument, and the request would now fail.

July 2019

App-only tokens for single-tenant applications are only issued if the client app exists in the resource tenant

Effective date: July 26, 2019

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: Client Credentials (app-only tokens)

A security change went live July 26th that changes the way app-only tokens (via the client credentials grant) are issued. Previously, applications were allowed to get tokens to call any other app, regardless of presence in the tenant or roles consented to for that application. This behavior has been updated so that for resources (sometimes called Web APIs) set to be single-tenant (the default), the client application must exist within the resource tenant. Note that existing consent between the client and the API is still not required, and apps should still be doing their own authorization checks to ensure that a `roles` claim is present and contains the expected value for the API.

The error message for this scenario currently states:

```
The service principal named <appName> was not found in the tenant named <tenant_name>. This can happen if the application has not been installed by the administrator of the tenant.
```

To remedy this issue, use the Admin Consent experience to create the client application service principal in your tenant, or create it manually. This requirement ensures that the tenant has given the application permission to operate within the tenant.

Example request

```
https://login.microsoftonline.com/contoso.com/oauth2/authorize?  
resource=https://gateway.contoso.com/api&response_type=token&client_id=14c88eee-b3e2-4bb0-9233-f5e3053b3a28&...
```

In this example, the resource tenant (authority) is contoso.com, the resource app is a single-tenant app called `gateway.contoso.com/api` for the Contoso tenant, and the client app is `14c88eee-b3e2-4bb0-9233-f5e3053b3a28`. If the client app has a service principal within Contoso.com, this request can continue. If it doesn't, however, then the request will fail with the error above.

If the Contoso gateway app were a multi-tenant application, however, then the request would continue regardless of the client app having a service principal within Contoso.com.

Redirect URIs can now contain query string parameters

Effective date: July 22, 2019

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: All flows

Per [RFC 6749](#), Azure AD applications can now register and use redirect (reply) URIs with static query parameters (such as <https://contoso.com/oauth2?idp=microsoft>) for OAuth 2.0 requests. Dynamic redirect URIs are still forbidden as they represent a security risk, and this cannot be used to retain state information across an authentication request - for that, use the `state` parameter.

The static query parameter is subject to string matching for redirect URIs like any other part of the redirect URI - if no string is registered that matches the URI-decoded `redirect_uri`, then the request will be rejected. If the URI is found in the app registration, then the entire string will be used to redirect the user, including the static query parameter.

Note that at this time (End of July 2019), the app registration UX in Azure portal still block query parameters. However, you can edit the application manifest manually to add query parameters and test this in your app.

March 2019

Looping clients will be interrupted

Effective date: March 25, 2019

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: All flows

Client applications can sometimes misbehave, issuing hundreds of the same login request over a short period of time. These requests may or may not be successful, but they all contribute to poor user experience and heightened workloads for the IDP, increasing latency for all users and reducing availability of the IDP. These applications are operating outside the bounds of normal usage, and should be updated to behave correctly.

Clients that issue duplicate requests multiple times will be sent an `invalid_grant` error:

AADSTS50196: The server terminated an operation because it encountered a loop while processing a request.

Most clients will not need to change behavior to avoid this error. Only misconfigured clients (those without token caching or those exhibiting prompt loops already) will be impacted by this error. Clients are tracked on a per-instance basis locally (via cookie) on the following factors:

- User hint, if any
- Scopes or resource being requested
- Client ID
- Redirect URI
- Response type and mode

Apps making multiple requests (15+) in a short period of time (5 minutes) will receive an `invalid_grant` error explaining that they are looping. The tokens being requested have sufficiently long-lived lifetimes (10 minutes minimum, 60 minutes by default), so repeated requests over this time period are unnecessary.

All apps should handle `invalid_grant` by showing an interactive prompt, rather than silently requesting a token. In order to avoid this error, clients should ensure they are correctly caching the tokens they receive.

October 2018

Authorization codes can no longer be reused

Effective date: November 15, 2018

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: [Code flow](#)

Starting on November 15, 2018, Azure AD will stop accepting previously used authentication codes for apps. This security change helps to bring Azure AD in line with the OAuth specification and will be enforced on both the v1 and v2 endpoints.

If your app reuses authorization codes to get tokens for multiple resources, we recommend that you use the code to get a refresh token, and then use that refresh token to acquire additional tokens for other resources.

Authorization codes can only be used once, but refresh tokens can be used multiple times across multiple resources. Any new app that attempts to reuse an authentication code during the OAuth code flow will get an `invalid_grant` error.

For more information about refresh tokens, see [Refreshing the access tokens](#). If using ADAL or MSAL, this is handled for you by the library - replace the second instance of 'AcquireTokenByAuthorizationCodeAsync' with 'AcquireTokenSilentAsync'.

May 2018

ID tokens cannot be used for the OBO flow

Date: May 1, 2018

Endpoints impacted: Both v1.0 and v2.0

Protocols impacted: Implicit flow and [OBO flow](#)

After May 1, 2018, id_tokens cannot be used as the assertion in an OBO flow for new applications. Access tokens should be used instead to secure APIs, even between a client and middle tier of the same application. Apps registered before May 1, 2018 will continue to work and be able to exchange id_tokens for an access token; however, this pattern is not considered a best practice.

To work around this change, you can do the following:

1. Create a Web API for your application, with one or more scopes. This explicit entry point will allow finer grained control and security.
2. In your app's manifest, in the [Azure portal](#) or the [app registration portal](#), ensure that the app is allowed to issue access tokens via the implicit flow. This is controlled through the `oauth2AllowImplicitFlow` key.
3. When your client application requests an id_token via `response_type=id_token`, also request an access token (`response_type=token`) for the Web API created above. Thus, when using the v2.0 endpoint the `scope` parameter should look similar to `api://GUID/SCOPE`. On the v1.0 endpoint, the `resource` parameter should be the app URI of the web API.
4. Pass this access token to the middle tier in place of the id_token.

Azure Active Directory for developers (v1.0) overview

10/15/2019 • 2 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) is a cloud identity service that allows developers to build apps that securely sign in users with a Microsoft work or school account. Azure AD supports developers building both single-tenant, line-of-business (LOB) apps, as well as developers looking to develop multi-tenant apps. In addition to basic sign in, Azure AD also lets apps call both Microsoft APIs like [Microsoft Graph](#) and custom APIs that are built on the Azure AD platform. This documentation shows you how to add Azure AD support to your application by using industry standard protocols like OAuth2.0 and OpenID Connect.

NOTE

Most of the content on this page focuses on the v1.0 endpoint and platform, which supports only Microsoft work or school accounts. If you want to sign in consumer or personal Microsoft accounts, see the information on the [v2.0 endpoint and platform](#). The v2.0 endpoint offers a unified developer experience for apps that want to sign in all Microsoft identities.

Authentication basics	An introduction to authentication with Azure AD.
Types of applications	An overview of the authentication scenarios that are supported by Azure AD.

Get started

The v1.0 quickstarts and tutorials walk you through building an app on your preferred platform using the Azure AD Authentication Library (ADAL) SDK. See the **v1.0 Quickstarts** and **v1.0 Tutorials** in [Microsoft identity platform \(Azure Active Directory for developers\)](#) to get started.

How-to guides

See the **v1.0 How-to guides** for detailed info and walkthroughs of the most common tasks in Azure AD.

Reference topics

The following articles provide detailed information about APIs, protocol messages, and terms that are used in Azure AD.

Authentication Libraries (ADAL)	An overview of the libraries and SDKs that are provided by Azure AD.
Code samples	A list of all of the Azure AD code samples.
Glossary	Terminology and definitions of words that are used throughout this documentation.

Help and support

If you need help, want to report an issue, or want to learn more about your support options, see the following article:

[Help and support for developers](#)

Quickstart: Set up a tenant

9/25/2019 • 3 minutes to read • [Edit Online](#)

The Microsoft identity platform allows developers to build apps targeting a wide variety of custom Microsoft 365 environments and identities. To get started using Microsoft identity platform, you will need access to an environment, also called an Azure AD tenant, that can register and manage apps, have access to Microsoft 365 data, and deploy custom Conditional Access and tenant restrictions.

A tenant is a representation of an organization. It's a dedicated instance of Azure AD that an organization or app developer receives when the organization or app developer creates a relationship with Microsoft-- like signing up for Azure, Microsoft Intune, or Microsoft 365.

Each Azure AD tenant is distinct and separate from other Azure AD tenants and has its own representation of work and school identities, consumer identities (if it's an Azure AD B2C tenant), and app registrations. An app registration inside of your tenant can allow authentications from accounts only within your tenant or all tenants.

Determining environment type

There are two types of environments you can create. Deciding which you need is based solely on the types of users your app will authenticate.

- Work and school (Azure AD accounts) or Microsoft accounts (such as outlook.com and live.com)
- Social and local accounts (Azure AD B2C)

The quickstart is broken into two scenarios depending on the type of app you want to build. If you need more help targeting an identity type, take a look at [about Microsoft identity platform](#)

Work and school accounts, or personal Microsoft accounts

Use an existing tenant

Many developers already have tenants through services or subscriptions that are tied to Azure AD tenants such as Microsoft 365 or Azure subscriptions.

1. To check the tenant, sign in to the [Azure portal](#) with the account you want to use to manage your application.
2. Check the upper right corner. If you have a tenant, you'll automatically be logged in and can see the tenant name directly under your account name.
 - Hover over your account name on the upper right-hand side of the Azure portal to see your name, email, directory / tenant ID (a GUID), and your domain.
 - If your account is associated with multiple tenants, you can select your account name to open a menu where you can switch between tenants. Each tenant has its own tenant ID.

TIP

If you need to find the tenant ID, you can:

- Hover over your account name to get the directory / tenant ID, or
- Select **Azure Active Directory > Properties > Directory ID** in the Azure portal

If you don't have an existing tenant associated with your account, you'll see a GUID under your account name and you won't be able to perform actions like registering apps until you follow the steps of the next section.

Create a new Azure AD tenant

If you don't already have an Azure AD tenant or want to create a new one for development, follow the [directory creation experience](#). You will have to provide the following info to create your new tenant:

- **Organization name**
- **Initial domain** - this will be part of *.onmicrosoft.com. You can customize the domain more later.
- **Country or region**

NOTE

When naming your tenant, use alphanumeric characters. Special characters are not allowed. The name must not exceed 256 characters.

Social and local accounts

To begin building apps that sign in social and local accounts, you'll need to create an Azure AD B2C tenant. To begin, follow [creating an Azure AD B2C tenant](#).

Next steps

- Try a coding quickstart and begin authenticating users.
- For more in-depth code samples, check out the **Tutorials** section of the documentation.
- Want to deploy your app to the cloud? Check out [deploying containers to Azure](#).

Quickstart: Register an application with the Microsoft identity platform

11/4/2019 • 3 minutes to read • [Edit Online](#)

Enterprise developers and software-as-a-service (SaaS) providers can develop commercial cloud services or line-of-business applications that can be integrated with Microsoft identity platform to provide secure sign-in and authorization for their services.

This quickstart shows you how to add and register an application using the **App registrations** experience in the Azure portal so that your app can be integrated with the Microsoft identity platform. To learn more about the new features and improvements in the new app registrations experience, see [this blog post](#).

Register a new application using the Azure portal

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the Azure AD tenant that you want.
3. Search for and select **Azure Active Directory**. On the **Active Directory** page, select **App registrations** and then select **New registration**.
4. When the **Register an application** page appears, enter your application's registration information:
 - **Name** - Enter a meaningful application name that will be displayed to users of the app.
 - **Supported account types** - Select which accounts you would like your application to support.

SUPPORTED ACCOUNT TYPES	DESCRIPTION
Accounts in this organizational directory only	Select this option if you're building a line-of-business (LOB) application. This option is not available if you're not registering the application in a directory. This option maps to Azure AD only single-tenant. This is the default option unless you're registering the app outside of a directory. In cases where the app is registered outside of a directory, the default is Azure AD multi-tenant and personal Microsoft accounts.
Accounts in any organizational directory	Select this option if you would like to target all business and educational customers. This option maps to an Azure AD only multi-tenant. If you registered the app as Azure AD only single-tenant, you can update it to be Azure AD multi-tenant and back to single-tenant through the Authentication blade.

Supported account types	Description
Accounts in any organizational directory and personal Microsoft accounts	<p>Select this option to target the widest set of customers.</p> <p>This option maps to Azure AD multi-tenant and personal Microsoft accounts.</p> <p>If you registered the app as Azure AD multi-tenant and personal Microsoft accounts, you cannot change this in the UI. Instead, you must use the application manifest editor to change the supported account types.</p>

- **Redirect URI (optional)** - Select the type of app you're building, **Web** or **Public client (mobile & desktop)**, and then enter the redirect URI (or reply URL) for your application.
 - For web applications, provide the base URL of your app. For example, `https://localhost:31544` might be the URL for a web app running on your local machine. Users would use this URL to sign in to a web client application.
 - For public client applications, provide the URI used by Azure AD to return token responses. Enter a value specific to your application, such as `myapp://auth`.
- To see specific examples for web applications or native applications, check out our [quickstarts](#).

5. When finished, select **Register**.

The screenshot shows the 'Register an application' dialog in the Azure portal. The 'Name' field is filled with 'ContosoApp_1'. Under 'Supported account types', the 'Accounts in this organizational directory only (Contoso Enterprises)' radio button is selected. The 'Redirect URI (optional)' field shows 'Web' selected and 'https://contosoapp1/auth' entered. A 'Register' button is at the bottom.

Azure AD assigns a unique application (client) ID to your app, and you're taken to your application's **Overview** page. To add additional capabilities to your application, you can select other configuration options including branding, certificates and secrets, API permissions, and more.

Home > App registrations > ContosoApp_1

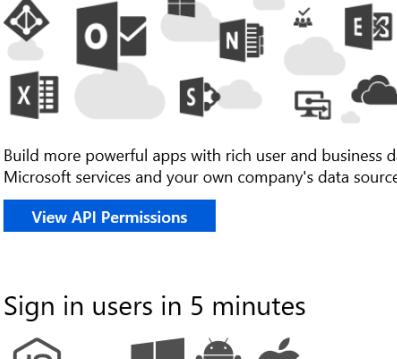
ContosoApp_1
PREVIEW

12:18 PM X

Create application
Successfully created application ContosoApp_1.

Overview	Delete Endpoints
Display name ContosoApp_1	Supported account types My organization only
Application (client) ID 95c232bc-5ab2-4954-8640-2a865eeb8597	Redirect URIs 1 web, 0 public client
Directory (tenant) ID 73e589d0-adbb-451c-8382-8c7f992efcccd	Managed application in local directory ContosoApp_1

Call APIs



Build more powerful apps with rich user and business data from Microsoft services and your own company's data sources.

View API Permissions

Documentation

- Azure Active Directory for Developers
- Authentication scenarios
- Authentication libraries
- Code samples and tutorials
- Microsoft Graph
- Glossary
- Help and Support

Sign in users in 5 minutes



Use our SDKs to sign in users and call APIs in a few steps

View all quickstart guides

Next steps

- Learn about the [permissions and consent](#).
- To enable additional configuration features in your application registration, such as credentials and permissions, and enable sign-in for users from other tenants, see these quickstarts:
 - [Configure a client application to access web APIs](#)
 - [Configure an application to expose web APIs](#)
 - [Modify the accounts supported by an application](#)
- Choose a [quickstart](#) to quickly build an app and add functionality like getting tokens, refreshing tokens, signing in a user, displaying some user info, and more.
- Learn more about the two Azure AD objects that represent a registered application and the relationship between them, see [Application objects and service principal objects](#).
- Learn more about the branding guidelines you should use when developing apps, see [Branding guidelines for applications](#).

Quickstart: Configure a client application to access web APIs

11/4/2019 • 7 minutes to read • [Edit Online](#)

For a web/confidential client application to be able to participate in an authorization grant flow that requires authentication (and obtain an access token), it must establish secure credentials. The default authentication method supported by the Azure portal is client ID + secret key.

Additionally, before a client can access a web API exposed by a resource application (such as Microsoft Graph API), the consent framework ensures the client obtains the permission grant required based on the permissions requested. By default, all applications can choose permissions from the Microsoft Graph API. The [Graph API "Sign-in and read user profile" permission](#) is selected by default. You can select from [two types of permissions](#) for each desired web API:

- **Application permissions** - Your client application needs to access the web API directly as itself (no user context). This type of permission requires administrator consent and is also not available for public (desktop and mobile) client applications.
- **Delegated permissions** - Your client application needs to access the web API as the signed-in user, but with access limited by the selected permission. This type of permission can be granted by a user unless the permission requires administrator consent.

NOTE

Adding a delegated permission to an application does not automatically grant consent to the users within the tenant. Users must still manually consent for the added delegated permissions at runtime, unless the administrator grants consent on behalf of all users.

In this quickstart, we'll show you how to configure your app to:

- [Add redirect URIs to your application](#)
- [Configure advanced settings for your application](#)
- [Modify supported account types](#)
- [Add credentials to your web application](#)
- [Add permissions to access web APIs](#)

Prerequisites

To get started, make sure you complete these prerequisites:

- Learn about the supported [permissions and consent](#), which is important to understand when building applications that need to be used by other users or applications.
- Have a tenant that has applications registered to it.
 - If you don't have apps registered, [learn how to register applications with the Microsoft identity platform](#).

Sign in to the Azure portal and select the app

Before you can configure the app, follow these steps:

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top-right corner, and set your portal session to the desired Azure AD tenant.
3. Search for and select **Azure Active Directory**.
4. From the left pane, select **App registrations**.
5. Find and select the application you want to configure. Once you've selected the app, you'll see the application's **Overview** or main registration page.
6. Follow the steps to configure your application to access web APIs:
 - [Add redirect URIs to your application](#)
 - [Configure advanced settings for your application](#)
 - [Modify supported account types](#)
 - [Add credentials to your web application](#)
 - [Add permissions to access web APIs](#)

Add redirect URI(s) to your application

To add a redirect URI to your application:

1. From the app's **Overview** page, select the **Authentication** section.
2. To add a custom redirect URI for web and public client applications, follow these steps:
 - a. Locate the **Redirect URI** section.
 - b. Select the type of application you're building, **Web** or **Public client (mobile & desktop)**.
 - c. Enter the Redirect URI for your application.
 - For web applications, provide the base URL of your application. For example, `http://localhost:31544` might be the URL for a web application running on your local machine. Users would use this URL to sign into a web client application.
 - For public applications, provide the URI used by Azure AD to return token responses. Enter a value specific to your application, for example: `https://MyFirstApp`.
3. To choose from suggested Redirect URIs for public clients (mobile, desktop), follow these steps:
 - a. Locate the Suggested **Redirect URIs for public clients (mobile, desktop)** section.
 - b. Select the appropriate Redirect URI(s) for your application using the checkboxes. You can also enter a custom redirect URI. If you're not sure what to use, check out the library documentation.

There are certain restrictions that apply to redirect URIs. Learn more about [redirect URI restrictions and limitations](#).

NOTE

Try out the new **Authentication** settings experience where you can configure settings for your application based on the platform or device that you want to target.

To see this view, select **Try out the new experience** from the default **Authentication** page view.



This takes you to the [new Platform configurations](#) page.

Configure advanced settings for your application

Depending on the application you're registering, there are some additional settings that you may need to

configure, such as:

- **Logout URL**
- For single-page apps, you can enable **Implicit grant** and select the tokens that you'd like the authorization endpoint to issue.
- For desktop apps that are acquiring tokens with Integrated Windows Authentication, device code flow, or username/password in the **Default client type** section, configure the **Treat application as public client** setting to **Yes**.
- For legacy apps that were using the Live SDK to integrate with the Microsoft account service, configure **Live SDK support**. New apps don't need this setting.
- **Default client type**

Modify supported account types

The **Supported account types** specify who can use the application or access the API.

Once you've [configured the supported account types](#) when you initially registered the application, you can only change this setting using the application manifest editor if:

- You change account types from **AzureADMyOrg** or **AzureADMultipleOrgs** to **AzureADandPersonalMicrosoftAccount**, or vice versa.
- You change account types from **AzureADMyOrg** to **AzureADMultipleOrgs**, or vice versa.

To change the supported account types for an existing app registration:

- See [Configure the application manifest](#) and update the `signInAudience` key.

Configure platform settings for your application

New_Web_App - Platform configurations

Search (Ctrl+ /) | Save | Discard | Switch to the old experience | Got feedback?

Platform configurations

Depending on the platform or device this application is targeting, additional configuration may be required such as redirect URIs, specific authentication settings, or fields specific to the platform.

[+ Add a platform](#)

Web Quickstart Docs [Delete](#)

REDIRECT URI
http://localhost:31544 [Delete](#)

Add URI

LOGOUT URL [Edit](#)
e.g. https://myapp.com/logout [Checkmark](#)

IMPLICIT GRANT [Edit](#)

Allows an application to request a token directly from the authorization endpoint. Recommended only if the application has a single page architecture (SPA), has no backend components, or invokes a Web API via JavaScript.

To enable the implicit grant flow, select the tokens you would like to be issued by the authorization endpoint:

Access tokens
 ID tokens

Supported account types

Who can use this application or access this API?

Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)

All users with a work or school, or personal Microsoft account can use your application or API. This includes Office 365 subscribers.

⚠️ To change the supported accounts for an existing registration, use the manifest editor. Take care, as certain properties may cause errors for personal accounts.

Advanced settings

Live SDK support [Edit](#)

Allow direct integration with the Microsoft account service (login.live.com). Required for integration with Microsoft account SDKs such as Xbox or Bing Ads

Yes **No**

Default client type [Edit](#)

Treat application as a public client. Required for the use of the following flows where a redirect URI is not used:

Yes **No**

- Resource owner password credential (ROPC) [Learn more](#)
- Device code flow [Learn more](#)

To configure application settings based on the platform or device, you're targeting:

1. In the **Platform configurations** page, select **Add a platform** and choose from the available options.

Configure platforms

Web applications



Web

Single-page apps, Web apps

Mobile applications



iOS

Objective-C, Swift, Xamarin



Android

Java, Kotlin, Xamarin

Desktop + devices



Desktop + devices

Windows, UWP, Console, IoT & Limited-entry Devices, Classic iOS + Android

2. Enter the settings info based on the platform you selected.

PLATFORM	CHOICES	CONFIGURATION SETTINGS
Web applications	Web	Enter the Redirect URI for your application.
Mobile applications	iOS	Enter the app's Bundle ID , which you can find in XCode in Info.plist, or Build Settings. Adding the bundle ID automatically creates a redirect URI for the application.
	Android	* Provide the app's Package name , which you can find in the AndroidManifest.xml file. * Generate and enter the Signature hash . Adding the signature hash automatically creates a redirect URI for the application.
Desktop + devices	Desktop + devices	* Optional. Select one of the recommended Suggested redirect URLs if you're building apps for desktop and devices. * Optional. Enter a Custom redirect URI , which is used as the location where Azure AD will redirect users in response to authentication requests. For example, for .NET Core applications where you want interaction, use <code>https://localhost</code> .

IMPORTANT

For mobile applications that aren't using the latest MSAL library or not using a broker, you must configure the redirect URLs for these applications in **Desktop + devices**.

3. Depending on the platform you chose, there may be additional settings that you can configure. For **Web** apps, you can:

- Add more redirect URLs
- Configure **Implicit grant** to select the tokens you'd like to be issued by the authorization endpoint:
 - For single-page apps, select both **Access tokens** and **ID tokens**
 - For web apps, select **ID tokens**

Add credentials to your web application

To add a credential to your web application:

1. From the app's **Overview** page, select the **Certificates & secrets** section.

2. To add a certificate, follow these steps:

- a. Select **Upload certificate**.
- b. Select the file you'd like to upload. It must be one of the following file types: .cer, .pem, .crt.
- c. Select **Add**.

3. To add a client secret, follow these steps:

- a. Select **New client secret**.
- b. Add a description for your client secret.
- c. Select a duration.
- d. Select **Add**.

NOTE

After you save the configuration changes, the right-most column will contain the client secret value. **Be sure to copy the value** for use in your client application code as it's not accessible once you leave this page.

Add permissions to access web APIs

To add permission(s) to access resource APIs from your client:

1. From the app's **Overview** page, select **API permissions**.
2. Select the **Add a permission** button.
3. By default, the view allows you to select from **Microsoft APIs**. Select the section of APIs that you're interested in:
 - **Microsoft APIs** - Lets you select permissions for Microsoft APIs such as Microsoft Graph.
 - **APIs my organization uses** - Lets you select permissions for APIs that have been exposed by your organization, or APIs that your organization has integrated with.
 - **My APIs** - Lets you select permissions for APIs that you have exposed.
4. Once you've selected the APIs, you'll see the **Request API Permissions** page. If the API exposes both delegated and application permissions, select which type of permission your application needs.
5. When finished, select **Add permissions**. You will return to the **API permissions** page, where the permissions have been saved and added to the table.

Next steps

Learn about these other related app management quickstarts for apps:

- [Register an application with the Microsoft identity platform](#)
- [Configure an application to expose web APIs](#)
- [Modify the accounts supported by an application](#)
- [Remove an application registered with the Microsoft identity platform](#)

To learn more about the two Azure AD objects that represent a registered application and the relationship between them, see [Application objects and service principal objects](#).

To learn more about the branding guidelines you should use when developing applications with Azure Active Directory, see [Branding guidelines for applications](#).

Quickstart: Configure an application to expose web APIs

8/13/2019 • 5 minutes to read • [Edit Online](#)

You can develop a web API and make it available to client applications by exposing [permissions/scopes](#) and [roles](#). A correctly configured web API is made available just like the other Microsoft web APIs, including the Graph API and the Office 365 APIs.

In this quickstart, you'll learn how to configure an application to expose a new scope to make it available to client applications.

Prerequisites

To get started, make sure you complete these prerequisites:

- Learn about the supported [permissions and consent](#), which is important to understand when building applications that need to be used by other users or applications.
- Have a tenant that has applications registered to it.
 - If you don't have apps registered, [learn how to register applications with the Microsoft identity platform](#).

Sign in to the Azure portal and select the app

Before you can configure the app, follow these steps:

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. In the left-hand navigation pane, select the **Azure Active Directory** service and then select **App registrations**.
4. Find and select the application you want to configure. Once you've selected the app, you'll see the application's **Overview** or main registration page.
5. Choose which method you want to use, UI or application manifest, to expose a new scope:
 - [Expose a new scope through the UI](#)
 - [Expose a new scope or role through the application manifest](#)

Expose a new scope through the UI

To expose a new scope through the UI:

1. From the app's **Overview** page, select the **Expose an API** section.
2. Select **Add a scope**.
3. If you have not set an **Application ID URI**, you will see a prompt to enter one. Enter your application ID URI or use the one provided and then select **Save and continue**.
4. When the **Add a scope** page appears, enter your scope's information:

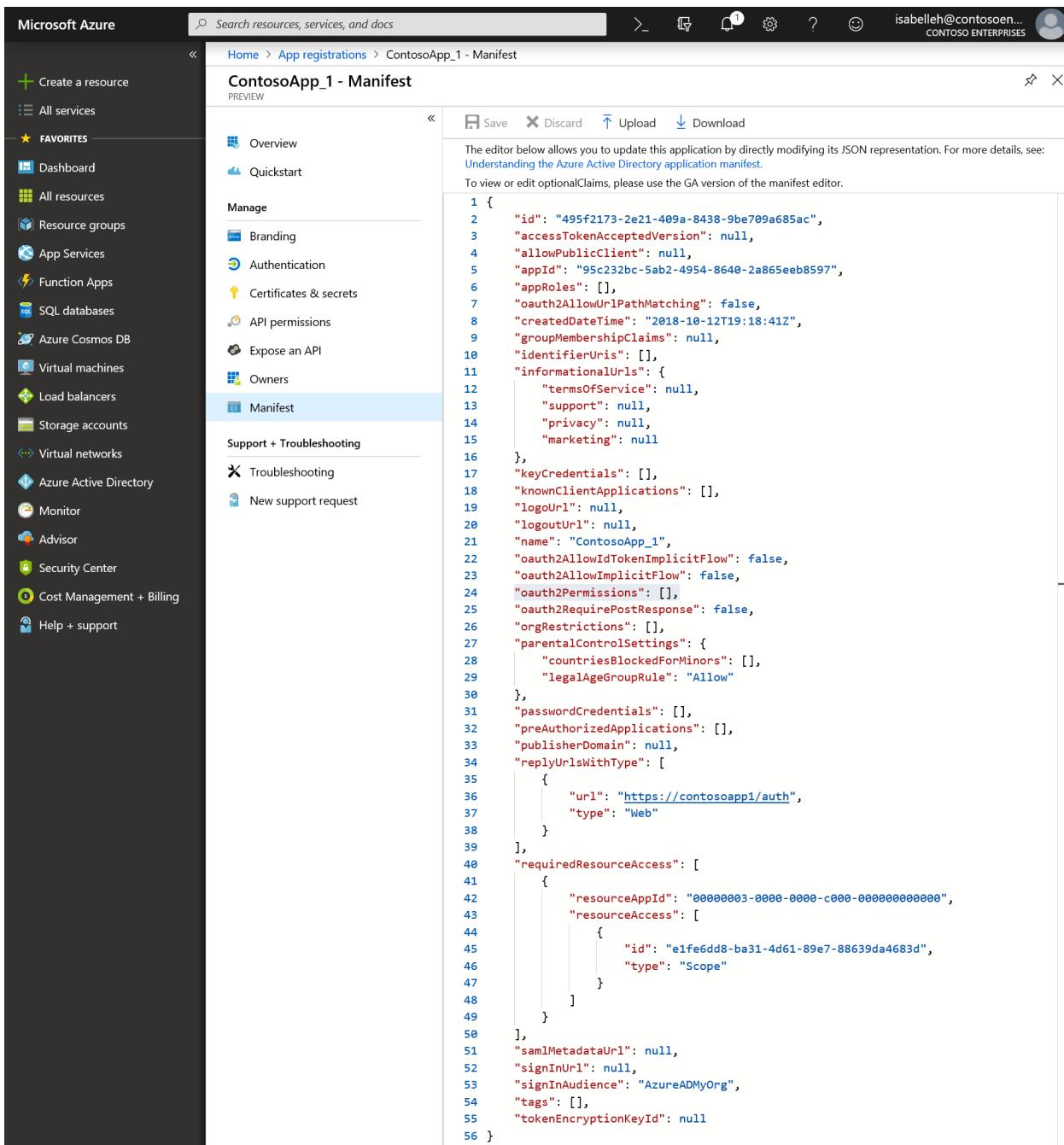
FIELD	DESCRIPTION
Scope name	Enter a meaningful name for your scope. For example, <code>Employees.Read.All</code> .
Who can consent	Select whether this scope can be consented to by users, or if admin consent is required. Select Admins only for higher-privileged permissions.
Admin consent display name	Enter a meaningful description for your scope, which admins will see. For example, <code>Read-only access to Employee records</code>
Admin consent description	Enter a meaningful description for your scope, which admins will see. For example, <code>Allow the application to have read-only access to all Employee data.</code>

If users can consent to your scope, also add values for the following fields:

FIELD	DESCRIPTION
User consent display name	<p>Enter a meaningful name for your scope, which users will see.</p> <p>For example,</p> <div style="border: 1px solid #ccc; padding: 5px; display: inline-block;">Read-only access to your Employee records</div>
User consent description	<p>Enter a meaningful description for your scope, which users will see.</p> <p>For example,</p> <div style="border: 1px solid #ccc; padding: 5px; display: inline-block;">Allow the application to have read-only access to your Employee data.</div>

5. Set the **State** and select **Add scope** when you're done.
6. Follow the steps to [verify that the web API is exposed to other applications](#).

Expose a new scope or role through the application manifest



The screenshot shows the Microsoft Azure portal interface. On the left, there's a dark sidebar with various service icons like Create a resource, All services, Favorites, Dashboard, etc. The main content area has a header "ContosoApp_1 - Manifest" with tabs for Overview, Quickstart, Manage, Support + Troubleshooting, and Troubleshooting. The "Manifest" tab is currently selected. Below the tabs, there are buttons for Save, Discard, Upload, and Download. A note says: "The editor below allows you to update this application by directly modifying its JSON representation. For more details, see: Understanding the Azure Active Directory application manifest." It also says "To view or edit optionalClaims, please use the GA version of the manifest editor." The main pane displays the following JSON manifest code:

```

1 {
2   "id": "495f2173-2e21-409a-8438-9be709a685ac",
3   "accessTokenAcceptedVersion": null,
4   "allowPublicClient": null,
5   "appId": "95c232bc-5ab2-4954-8640-2a865eeb8597",
6   "appRoles": [],
7   "oauth2AllowUrlPathMatching": false,
8   "createdDateTime": "2018-10-12T19:18:41Z",
9   "groupMembershipClaims": null,
10  "identifierUris": [],
11  "informationalUrls": {
12    "termsOfService": null,
13    "support": null,
14    "privacy": null,
15    "marketing": null
16  },
17  "keyCredentials": [],
18  "knownClientApplications": [],
19  "logonUrl": null,
20  "logoutUrl": null,
21  "name": "ContosoApp_1",
22  "oauth2AllowIdTokenImplicitFlow": false,
23  "oauth2AllowImplicitFlow": false,
24  "oauth2Permissions": [],
25  "oauth2RequirePostResponse": false,
26  "orgRestrictions": [],
27  "parentalControlSettings": {
28    "countriesBlockedForMinors": [],
29    "legalAgeGroupRule": "Allow"
30  },
31  "passwordCredentials": [],
32  "preAuthorizedApplications": [],
33  "publisherDomain": null,
34  "replyUrlsWithType": [
35    {
36      "url": "https://contosoapp1/auth",
37      "type": "Web"
38    }
39  ],
40  "requiredResourceAccess": [
41    {
42      "resourceAppId": "00000003-0000-0000-000000000000",
43      "resourceAccess": [
44        {
45          "id": "e1fe6dd8-ba31-4d61-89e7-88639da4683d",
46          "type": "Scope"
47        }
48      ]
49    }
50  ],
51  "samlMetadataUrl": null,
52  "signInUrl": null,
53  "signInAudience": "AzureADMyOrg",
54  "tags": [],
55  "tokenEncryptionKeyId": null
56 }

```

To expose a new scope through the application manifest:

1. From the app's **Overview** page, select the **Manifest** section. A web-based manifest editor opens, allowing you to **Edit** the manifest within the portal. Optionally, you can select **Download** and edit the manifest locally, and then use **Upload** to reapply it to your application.

The following example shows how to expose a new scope called `Employees.Read.All` in the resource/API by adding the following JSON element to the `oauth2Permissions` collection.

```
{  
    "adminConsentDescription": "Allow the application to have read-only access to all Employee data.",  
    "adminConsentDisplayName": "Read-only access to Employee records",  
    "id": "2b351394-d7a7-4a84-841e-08a6a17e4cb8",  
    "isEnabled": true,  
    "type": "User",  
    "userConsentDescription": "Allow the application to have read-only access to your Employee data.",  
    "userConsentDisplayName": "Read-only access to your Employee records",  
    "value": "Employees.Read.All"  
}
```

NOTE

The `id` value must be generated programmatically or by using a GUID generation tool such as [guidgen](#). The `id` represents a unique identifier for the scope as exposed by the web API. Once a client is appropriately configured with permissions to access your web API, it is issued an OAuth 2.0 access token by Azure AD. When the client calls the web API, it presents the access token that has the scope (scp) claim set to the permissions requested in its application registration.

You can expose additional scopes later as necessary. Consider that your web API might expose multiple scopes associated with a variety of different functions. Your resource can control access to the web API at runtime by evaluating the scope (`scp`) claim(s) in the received OAuth 2.0 access token.

2. When finished, click **Save**. Now your web API is configured for use by other applications in your directory.
3. Follow the steps to [verify that the web API is exposed to other applications](#).

Verify the web API is exposed to other applications

1. Go back to your Azure AD tenant, select **App registrations**, find and select the client application you want to configure.
2. Repeat the steps outlined in [Configure a client application to access web APIs](#).
3. When you get to the step to [select an API](#), select your resource. You should see the new scope, available for client permission requests.

More on the application manifest

The application manifest serves as a mechanism for updating the application entity, which defines all attributes of an Azure AD application's identity configuration. For more information on the Application entity and its schema, see the [Graph API Application entity documentation](#). The article contains complete reference information on the Application entity members used to specify permissions for your API, including:

- The `appRoles` member, which is a collection of [AppRole](#) entities, used to define [application permissions](#) for a web API.
- The `oauth2Permissions` member, which is a collection of [OAuth2Permission](#) entities, used to define [delegated permissions](#) for a web API.

For more information on application manifest concepts in general, see [Understanding the Azure Active Directory application manifest](#).

Next steps

Learn about these other related app management quickstarts for apps:

- [Register an application with the Microsoft identity platform](#)
- [Configure a client application to access web APIs](#)
- [Modify the accounts supported by an application](#)
- [Remove an application registered with the Microsoft identity platform](#)

To learn more about the two Azure AD objects that represent a registered application and the relationship between them, see [Application objects and service principal objects](#).

To learn more about the branding guidelines you should use when developing applications with Azure Active Directory, see [Branding guidelines for applications](#).

Quickstart: Modify the accounts supported by an application

7/22/2019 • 4 minutes to read • [Edit Online](#)

When registering an application in the Microsoft identity platform, you may want your application to be accessed only by users in your organization. Alternatively, you may also want your application to be accessible by users in external organizations, or by users in external organizations as well as users that are not necessarily part of an organization (personal accounts).

In this quickstart, you'll learn how to modify your application's configuration to change who, or what accounts, can access the application.

Prerequisites

To get started, make sure you complete these prerequisites:

- Learn about the supported [permissions and consent](#), which is important to understand when building applications that need to be used by other users or applications.
- Have a tenant that has applications registered to it.
 - If you don't have apps registered, [learn how to register applications with the Microsoft identity platform](#).

Sign in to the Azure portal and select the app

Before you can configure the app, follow these steps:

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. In the left-hand navigation pane, select the **Azure Active Directory** service and then select **App registrations**.
4. Find and select the application you want to configure. Once you've selected the app, you'll see the application's **Overview** or main registration page.
5. Follow the steps to [change the application registration to support different accounts](#).
6. If you have a single-page application, [enable OAuth 2.0 implicit grant](#).

Change the application registration to support different accounts

If you are writing an application that you want to make available to your customers or partners outside of your organization, you need to update the application definition in the Azure portal.

IMPORTANT

Azure AD requires the Application ID URI of multi-tenant applications to be globally unique. The App ID URI is one of the ways an application is identified in protocol messages. For a single-tenant application, it is sufficient for the App ID URI to be unique within that tenant. For a multi-tenant application, it must be globally unique so Azure AD can find the application across all tenants. Global uniqueness is enforced by requiring the App ID URI to have a host name that matches a verified domain of the Azure AD tenant. For example, if the name of your tenant is contoso.onmicrosoft.com, then a valid App ID URI would be <https://contoso.onmicrosoft.com/myapp>. If your tenant has a verified domain of contoso.com, then a valid App ID URI would also be <https://contoso.com/myapp>. If the App ID URI doesn't follow this pattern, setting an application as multi-tenant fails.

To change who can access your application

1. From the app's **Overview** page, select the **Authentication** section and change the value selected under **Supported account types**.
 - Select **Accounts in this directory only** if you are building a line-of-business (LOB) application. This option is not available if the application is not registered in a directory.
 - Select **Accounts in any organizational directory** if you would like to target all business and educational customers.
 - Select **Accounts in any organizational directory and personal Microsoft accounts** to target the widest set of customers.
2. Select **Save**.

Enable OAuth 2.0 implicit grant for single-page applications

Single-page applications (SPAs) are typically structured with a JavaScript-heavy front end that runs in the browser, which calls the application's web API back-end to perform its business logic. For SPAs hosted in Azure AD, you use OAuth 2.0 Implicit Grant to authenticate the user with Azure AD and obtain a token that you can use to secure calls from the application's JavaScript client to its back-end web API.

After the user has granted consent, this same authentication protocol can be used to obtain tokens to secure calls between the client and other web API resources configured for the application. To learn more about the implicit authorization grant, and help you decide whether it's right for your application scenario, learn about the OAuth 2.0 implicit grant flow in Azure AD [v1.0](#) and [v2.0](#).

By default, OAuth 2.0 implicit grant is disabled for applications. You can enable OAuth 2.0 implicit grant for your application by following the steps outlined below.

To enable OAuth 2.0 implicit grant

1. From the app's **Overview** page, select the **Authentication** section.
2. Under **Advanced settings**, locate the **Implicit grant** section.
3. Select **ID tokens**, **Access tokens**, or both.
4. Select **Save**.

Next steps

Learn about these other related app management quickstarts for apps:

- [Register an application with the Microsoft identity platform](#)
- [Configure a client application to access web APIs](#)
- [Configure an application to expose web APIs](#)
- [Remove an application registered with the Microsoft identity platform](#)

To learn more about the two Azure AD objects that represent a registered application and the relationship

between them, see [Application objects and service principal objects](#).

To learn more about the branding guidelines you should use when developing applications with Azure Active Directory, see [Branding guidelines for applications](#).

Quickstart: Remove an application registered with the Microsoft identity platform

5/10/2019 • 2 minutes to read • [Edit Online](#)

Enterprise developers and software-as-a-service (SaaS) providers who have registered applications with Microsoft identity platform may need to remove an application's registration.

In this quickstart, you'll learn how to:

- Remove an application authored by you or your organization
- Remove an application authored by another organization

Prerequisites

You must have a tenant that has applications registered to it. To learn how to add and register apps, see [Register an application with the Microsoft identity platform](#).

Remove an application authored by you or your organization

Applications that you or your organization have registered are represented by both an application object and service principal object in your tenant. For more information, see [Application Objects and Service Principal Objects](#).

To remove an application

1. Sign in to the [Azure portal](#) using either a work or school account or a personal Microsoft account.
2. If your account gives you access to more than one tenant, select your account in the top right corner, and set your portal session to the desired Azure AD tenant.
3. In the left-hand navigation pane, select the **Azure Active Directory** service, then select **App registrations**. Find and select the application that you want to configure. Once you've selected the app, you'll see the application's **Overview** page.
4. From the **Overview** page, select **Delete**.
5. Select **Yes** to confirm that you want to delete the app.

NOTE

To delete an application, you need to be listed as an owner of the application or have admin privileges.

Remove an application authored by another organization

If you are viewing **App registrations** in the context of a tenant, a subset of the applications that appear under the **All apps** tab are from another tenant and were registered into your tenant during the consent process. More specifically, they are represented by only a service principal object in your tenant, with no corresponding application object. For more information on the differences between application and service principal objects, see [Application and service principal objects in Azure AD](#).

In order to remove an application's access to your directory (after having granted consent), the company administrator must remove its service principal. The administrator must have global admin access, and can

remove the application through the Azure portal or use the [Azure AD PowerShell Cmdlets](#) to remove access.

Next steps

Learn about these other related app management quickstarts:

- [Register an application with the Microsoft identity platform](#)
- [Configure a client application to access web APIs](#)
- [Configure an application to expose web APIs](#)
- [Modify the accounts supported by an application](#)

Quickstart: Build an AngularJS single-page app for sign-in and sign out with Azure Active Directory

10/30/2019 • 6 minutes to read • [Edit Online](#)

Applies to:

- Azure AD v1.0 endpoint
- Azure Active Directory Authentication Library (ADAL)

IMPORTANT

Microsoft identity platform is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs such as Microsoft Graph or APIs that developers have built. If you need to enable sign-in for personal accounts in addition to work and school accounts, you can use the [*Microsoft identity platform endpoint*](#). This quickstart is for the older, Azure AD v1.0 endpoint. We recommend that you use the v2.0 endpoint for new projects. For more info, see [this JavaScript SPA tutorial](#) as well as [this article](#) explaining the [*Microsoft identity platform endpoint*](#).

Azure Active Directory (Azure AD) makes it simple and straightforward for you to add sign-in, sign-out, and secure OAuth API calls to your single-page apps. It enables your apps to authenticate users with their Windows Server Active Directory accounts and consume any web API that Azure AD helps protect, such as the Office 365 APIs or the Azure API.

For JavaScript applications running in a browser, Azure AD provides the Active Directory Authentication Library (ADAL), or adal.js. The sole purpose of adal.js is to make it easy for your app to get access tokens.

In this quickstart, you'll learn how to build an AngularJS To Do List application that:

- Signs the user in to the app by using Azure AD as the identity provider.
- Displays some information about the user.
- Securely calls the app's To Do List API by using bearer tokens from Azure AD.
- Signs the user out of the app.

To build the complete, working application, you'll need to:

1. Register your app with Azure AD.
2. Install ADAL and configure the single-page app.
3. Use ADAL to help secure pages in the single-page app.

Prerequisites

To get started, complete these prerequisites:

- [Download the app skeleton](#) or [download the completed sample](#).
- Have an Azure AD tenant in which you can create users and register an application. If you don't already have a tenant, [learn how to get one](#).

Step 1: Register the DirectorySearcher application

To enable your app to authenticate users and get tokens, you first need to register it in your Azure AD tenant:

1. Sign in to the [Azure portal](#).
2. If you are signed in to multiple directories, you may need to ensure you are viewing the correct directory. To do so, on the top bar, click your account. Under the **Directory** list, choose the Azure AD tenant where you want to register your application.
3. Click **All services** in the left pane, and then select **Azure Active Directory**.
4. Click **App registrations**, and then select **New registration**.
5. When the **Register an application** page appears, enter a name for your application.
6. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
7. Select the **Web** platform under the **Redirect URI** section and set the value to `https://localhost:44326/` (the location to which Azure AD will return tokens).
8. When finished, select **Register**. On the app **Overview** page, note down the **Application (client) ID** value.
9. Adal.js uses the OAuth implicit flow to communicate with Azure AD. You must enable the implicit flow for your application. In the left-hand navigation pane of the registered application, select **Authentication**.
10. In **Advanced settings**, under **Implicit grant**, enable both **ID tokens** and **Access tokens** checkboxes. ID tokens and access tokens are required since this app needs to sign in users and call an API.
11. Select **Save**.
12. Grant permissions across your tenant for your application. Go to **API permissions**, and select the **Grant admin consent** button under **Grant consent**.
13. Select **Yes** to confirm.

Step 2: Install ADAL and configure the single-page app

Now that you have an application in Azure AD, you can install adal.js and write your identity-related code.

Configure the JavaScript client

Begin by adding adal.js to the TodoSPA project by using the Package Manager Console:

1. Download [adal.js](#) and add it to the `App/Scripts/` project directory.
2. Download [adal-angular.js](#) and add it to the `App/Scripts/` project directory.
3. Load each script before the end of the `</body>` in `index.html`:

```
...
<script src="App/Scripts/adal.js"></script>
<script src="App/Scripts/adal-angular.js"></script>
...
```

Configure the back end server

For the single-page app's back-end To Do List API to accept tokens from the browser, the back end needs configuration information about the app registration. In the TodoSPA project, open `web.config`. Replace the values of the elements in the `<appSettings>` section to reflect the values that you used in the Azure portal. Your code will reference these values whenever it uses ADAL.

- `ida:Tenant` is the domain of your Azure AD tenant--for example, contoso.onmicrosoft.com.
- `ida:Audience` is the client ID of your application that you copied from the portal.

Step 3: Use ADAL to help secure pages in the single-page app

Adal.js integrates with AngularJS route and HTTP providers, so you can help secure individual views in your single-page app.

1. In `App/Scripts/app.js`, bring in the adal.js module:

```
angular.module('todoApp', ['ngRoute', 'AdalAngular'])
.config(['$routeProvider', '$httpProvider', 'adalAuthenticationServiceProvider',
  function ($routeProvider, $httpProvider, adalProvider) {
    ...
  }
]);
```

2. Initialize `adalProvider` by using the configuration values of your application registration, also in `App/Scripts/app.js`:

```
adalProvider.init(
{
  instance: 'https://login.microsoftonline.com/',
  tenant: 'Enter your tenant name here e.g. contoso.onmicrosoft.com',
  clientId: 'Enter your client ID here e.g. e9a5a8b6-8af7-4719-9821-0deef255f68e',
  extraQueryParameter: 'nux=1',
  //cacheLocation: 'localStorage', // enable this for IE, as sessionStorage does not work for
localhost.
},
$httpProvider
);
```

3. Help secure the `TodoList` view in the app by using only one line of code: `requireADLogin`.

```
...
}).when("/TodoList", {
  controller: "todoListCtrl",
  templateUrl: "/App/Views/TodoList.html",
  requireADLogin: true,
...
});
```

Summary

You now have a secure single-page app that can sign in users and issue bearer-token-protected requests to its back-end API. When a user clicks the **TodoList** link, adal.js will automatically redirect to Azure AD for sign-in if necessary. In addition, adal.js will automatically attach an access token to any Ajax requests that are sent to the app's back end.

The preceding steps are the bare minimum necessary to build a single-page app by using adal.js. But a few other features are useful in single-page app:

- To explicitly issue sign-in and sign-out requests, you can define functions in your controllers that invoke adal.js. In `App/Scripts/homeCtrl.js`:

```
...
$scope.login = function () {
  adalService.login();
};

$scope.logout = function () {
  adalService.logOut();
};
...
});
```

- You might want to present user information in the app's UI. The ADAL service has already been added to the `userDataCtrl` controller, so you can access the `userInfo` object in the associated view,

```
App/Views/UserData.html :
```

```
<p>{{userInfo.userName}}</p>
<p>aud:{{userInfo.profile.aud}}</p>
<p>iss:{{userInfo.profile.iss}}</p>
...
...
```

- There are many scenarios in which you'll want to know if the user is signed in or not. You can also use the `userInfo` object to gather this information. For instance, in `index.html`, you can show either the **Login** or **Logout** button based on authentication status:

```
<li><a class="btn btn-link" ng-show="userInfo.isAuthenticated" ng-click="logout()">Logout</a></li>
<li><a class="btn btn-link" ng-hide=" userInfo.isAuthenticated" ng-click="login()">Login</a></li>
```

Your Azure AD-integrated single-page app can authenticate users, securely call its back end by using OAuth 2.0, and get basic information about the user. If you haven't already, now is the time to populate your tenant with some users. Run your To Do List single-page app, and sign in with one of those users. Add tasks to the user's to-do list, sign out, and sign back in.

Adal.js makes it easy to incorporate common identity features into your application. It takes care of all the dirty work for you: cache management, OAuth protocol support, presenting the user with a sign-in UI, refreshing expired tokens, and more.

For reference, the completed sample (without your configuration values) is available in [GitHub](#).

Next steps

You can now move on to additional scenarios.

[Call a CORS web API from a single-page app.](#)

Quickstart: Add sign-in with Microsoft to an ASP.NET web app

10/30/2019 • 2 minutes to read • [Edit Online](#)

[Microsoft identity platform](#) is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs such as Microsoft Graph or APIs that developers have built.

[Microsoft Authentication Library \(MSAL\)](#) enables developers to acquire tokens from the Microsoft identity platform endpoint in order to access secured Web APIs. Active Directory Authentication Library (ADAL) integrates with the Azure AD for developers (v1.0) endpoint, where MSAL integrates with the Microsoft identity platform (v2.0) endpoint.

Next steps

For new web applications, we recommend you use Microsoft identity platform (v2.0) and MSAL to acquire tokens and access secured web APIs. See [Quickstart: Add sign-in with Microsoft to an ASP.NET web app](#) to get started.

Quickstart: Build a .NET web API that integrates with Azure AD for authentication and authorization

7/22/2019 • 2 minutes to read • [Edit Online](#)

[Microsoft identity platform](#) is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs such as Microsoft Graph or APIs that developers have built.

[Microsoft Authentication Library \(MSAL\)](#) enables developers to acquire tokens from the Microsoft identity platform endpoint in order to access secured Web APIs. Active Directory Authentication Library (ADAL) integrates with the Azure AD for developers (v1.0) endpoint, where MSAL integrates with the Microsoft identity platform (v2.0) endpoint.

For new web APIs, we recommend you use Microsoft identity platform (v2.0) and MSAL to acquire tokens and access secured web APIs: [Quickstart: Add sign-in with Microsoft to an ASP.NET web app](#)

Quickstart: Secure a Web API with Azure Active Directory

10/30/2019 • 2 minutes to read • [Edit Online](#)

Applies to:

- Azure AD v1.0 endpoint

[Microsoft identity platform](#) is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs such as Microsoft Graph or APIs that developers have built.

Next steps

See the [Azure Active Directory OIDC Node.js Web Sample](#) which demonstrates how to set up OpenId Connect authentication with Azure AD v2.0 in a web application built using Node.js with Express. It is designed to run on any platform.

Quickstart: Sign in users and call the Microsoft Graph API from an Android app (v1.0)

10/30/2019 • 2 minutes to read • [Edit Online](#)

[Microsoft identity platform](#) is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs such as Microsoft Graph or APIs that developers have built.

[Microsoft Authentication Library \(MSAL\)](#) enables developers to acquire tokens from the Microsoft identity platform endpoint in order to access secured Web APIs. Active Directory Authentication Library (ADAL) integrates with the Azure AD for developers (v1.0) endpoint, where MSAL integrates with the Microsoft identity platform (v2.0) endpoint.

Next steps

For new android applications, we recommend you use Microsoft identity platform (v2.0) and MSAL to acquire tokens and access secured web APIs. See [Quickstart: Sign in users and call the Microsoft Graph API from an Android app](#) to get started.

Quickstart: Sign in users and call the Microsoft Graph API from an iOS app (v1.0)

10/30/2019 • 2 minutes to read • [Edit Online](#)

[Microsoft identity platform](#) is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs such as Microsoft Graph or APIs that developers have built.

[Microsoft Authentication Library \(MSAL\)](#) enables developers to acquire tokens from the Microsoft identity platform endpoint in order to access secured Web APIs. Active Directory Authentication Library (ADAL) integrates with the Azure AD for developers (v1.0) endpoint, where MSAL integrates with the Microsoft identity platform (v2.0) endpoint.

For new macOS and iOS applications, we recommend you use Microsoft identity platform (v2.0) and MSAL to acquire tokens and access secured web APIs: [Quickstart: Sign in users and call the Microsoft Graph API from an iOS or macOS app](#).

Quickstart: Sign in users and call the Microsoft Graph API from a .NET Desktop (WPF) app

7/18/2019 • 2 minutes to read • [Edit Online](#)

[Microsoft identity platform](#) is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs such as Microsoft Graph or APIs that developers have built.

[Microsoft Authentication Library \(MSAL\)](#) enables developers to acquire tokens from the Microsoft identity platform endpoint in order to access secured Web APIs. Active Directory Authentication Library (ADAL) integrates with the Azure AD for developers (v1.0) endpoint, where MSAL integrates with the Microsoft identity platform (v2.0) endpoint.

For new desktop applications, we recommend you use Microsoft identity platform (v2.0) and MSAL to acquire tokens and access secured web APIs: [Quickstart: Acquire a token and call Microsoft Graph API from a Windows desktop app](#)

Quickstart: Build a Xamarin app that integrates Microsoft sign-in

10/30/2019 • 2 minutes to read • [Edit Online](#)

[Microsoft identity platform](#) is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs such as Microsoft Graph or APIs that developers have built.

[Microsoft Authentication Library \(MSAL\)](#) enables developers to acquire tokens from the Microsoft identity platform endpoint in order to access secured Web APIs. Active Directory Authentication Library (ADAL) integrates with the Azure AD for developers (v1.0) endpoint, where MSAL integrates with the Microsoft identity platform (v2.0) endpoint.

Next steps

For new Xamarin applications, we recommend you use Microsoft identity platform (v2.0) and MSAL to acquire tokens and access secured web APIs. See [Integrate Microsoft identity and the Microsoft Graph into a Xamarin forms app using MSAL](#) (without the optional steps) to get started.

Quickstart: Sign in users and call the Microsoft Graph API from a .NET Desktop (WPF) app

7/18/2019 • 2 minutes to read • [Edit Online](#)

Microsoft identity platform is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs such as Microsoft Graph or APIs that developers have built.

[Microsoft Authentication Library \(MSAL\)](#) enables developers to acquire tokens from the Microsoft identity platform endpoint in order to access secured Web APIs. Active Directory Authentication Library (ADAL) integrates with the Azure AD for developers (v1.0) endpoint, where MSAL integrates with the Microsoft identity platform (v2.0) endpoint.

For new desktop applications, we recommend you use Microsoft identity platform (v2.0) and MSAL to acquire tokens and access secured web APIs: [Quickstart: Acquire a token and call Microsoft Graph API from a Windows desktop app](#)

Quickstart: Acquire token and call Microsoft Graph API with console app's identity (v1.0)

10/30/2019 • 2 minutes to read • [Edit Online](#)

[Microsoft identity platform](#) is an evolution of the Azure Active Directory (Azure AD) developer platform. It allows developers to build applications that sign in all Microsoft identities and get tokens to call Microsoft APIs such as Microsoft Graph or APIs that developers have built.

[Microsoft Authentication Library \(MSAL\)](#) enables developers to acquire tokens from the Microsoft identity platform endpoint in order to access secured Web APIs. Active Directory Authentication Library (ADAL) integrates with the Azure AD for developers (v1.0) endpoint, where MSAL integrates with the Microsoft identity platform (v2.0) endpoint.

Next steps

For new .NET daemon applications, we recommend you use Microsoft identity platform (v2.0) and MSAL to acquire tokens and access secured web APIs: [Quickstart: Acquire a token and call Microsoft Graph API from a console app using an app's identity](#).

Azure Active Directory code samples (v1.0 endpoint)

10/23/2019 • 3 minutes to read • [Edit Online](#)

Applies to:

- Azure AD v1.0 endpoint
- Azure Active Directory Authentication Library (ADAL)

You can use Microsoft Azure Active Directory (Azure AD) to add authentication and authorization to your web applications and web APIs.

This section provides links to samples you can use to learn more about the Azure AD v1.0 endpoint. These samples show you how it's done along with code snippets that you can use in your applications. On the code sample page, you'll find detailed read-me topics that help with requirements, installation, and set-up. And the code is commented to help you understand the critical sections.

NOTE

If you are interested in Azure AD V2 code samples, see [v2.0 code samples by scenario](#).

To understand the basic scenario for each sample type, see [Authentication scenarios for Azure AD](#).

You can also contribute to our samples on GitHub. To learn how, see [Microsoft Azure Active Directory samples and documentation](#).

Single-page applications

This sample shows how to write a single-page application secured with Azure AD.

PLATFORM	CALLS ITS OWN API	CALLS ANOTHER WEB API
	javascript-singlepageapp	
	angularjs-singlepageapp	angularjs-singlepageapp-cors

Web Applications

Web Applications signing in users, calling Microsoft Graph, or a Web API with the user's identity

The following samples illustrate Web applications signing users. Some of these applications also call the Microsoft Graph or your own Web API, in the name of the signed-in user.

PLATFORM	ONLY SIGNS IN USERS	CALLS MICROSOFT GRAPH OR AAD GRAPH	CALLS ANOTHER ASP.NET OR ASP.NET CORE 2.0 WEB API
 ASP.NET Core 2.0	dotnet-webapp-openidconnect-aspnetcore	webapp-webapi-multitenant-openidconnect-aspnetcore (AAD Graph)	dotnet-webapp-webapi-openidconnect-aspnetcore
 ASP.NET 4.5	webApp-openidconnect-dotnet webapp-WSFederation-dotNet dotnet-webapp-webapi-oauth2-useridentity	dotnet-webapp-multitenant-openidconnect (AAD Graph)	
		python-webapp-graphapi	
		java-webapp-openidconnect	
		php-graphapi-web	

Web applications demonstrating role-based access control (authorization)

The following samples show how to implement role-based access control (RBAC). RBAC is used to restrict the permissions of certain features in a web application to certain users. The users are authorized depending on whether they belong to an **Azure AD group** or have a given application **role**.

PLATFORM	SAMPLE	
 ASP.NET 4.5	dotnet-webapp-groupclaims dotnet-webapp-roleclaims	A .NET 4.5 MVC web app that uses Azure AD roles for authorization

Desktop and mobile public client applications calling Microsoft Graph or a Web API

The following samples illustrate public client applications (deskto/pmobile applications) that access the Microsoft Graph or a Web API in the name of a user. Depending on the devices and platforms, applications can sign in users in different ways (flows/grants):

- Interactively
- Silently (with Integrated Windows Authentication on Windows, or username/password)
- By delegating the interactive sign-in to another device (device code flow used on devices which don't provide web controls)

CLIENT APPLICATION	PLATFORM	FLOW/GANT	CALLS MICROSOFT GRAPH	CALLS AN ASP.NET OR ASP.NET CORE 2.X WEB API
Desktop (WPF)		Interactive	Part of dotnet-native-multitarget	Dotnet-native-desktop dotnet-native-aspnetcore dotnet-webapi-manual-jwt-validation
Mobile (UWP)		Interactive	dotnet-native-uwp-wam This sample uses WAM , not ADAL.NET	dotnet-windows-store (UWP application using ADAL.NET to call a single tenant Web API) dotnet-webapi-multitenant-windows-store (UWP application using ADAL.NET to call a multi-tenant Web API)
Mobile (Android, iOS, UWP)		Interactive	dotnet-native-multitarget	
Mobile (Android)		Interactive	android	
Mobile (iOS)		Interactive	nativeClient-iOS	
Desktop (Console)		Username / Password Integrated Windows Authentication		dotnet-native-headless
Desktop (Console)		Username / Password		java-native-headless
Desktop (Console)		Device code flow		dotnet-deviceprofile

Daemon applications (accessing web APIs with the application's identity)

The following samples show desktop or web applications that access the Microsoft Graph or a web API with no user interaction.

user (with the application identity).

CLIENT APPLICATION	PLATFORM	FLOW/GANT	CALLS AN ASP.NET OR ASP.NET CORE 2.0 WEB API
Daemon app (Console)		Client Credentials with app secret or certificate	dotnet-daemon dotnet-daemon-certificate-credential
Daemon app (Console)		Client Credentials with certificate	dotnetcore-daemon-certificate-credential
ASP.NET Web App		Client credentials	dotnet-webapp-webapi-oauth2-appidentity

Web APIs

Web API protected by Azure Active Directory

The following sample shows how to protect a node.js web API with Azure AD.

In the previous sections of this article, you can also find other samples illustrating a client application **calling** an ASP.NET or ASP.NET Core **Web API**. These samples are not mentioned again in this section, but you will find them in the last column of the tables above or below

PLATFORM	SAMPLE
	node-webapi

Web API calling Microsoft Graph or another Web API

The following samples demonstrate a web API that calls another web API. The second sample shows how to handle Conditional Access.

PLATFORM	CALLS MICROSOFT GRAPH	CALLS ANOTHER ASP.NET OR ASP.NET CORE 2.0 WEB API
 ASP.NET 4.5	dotnet-webapi-onbehalfof dotnet-webapi-onbehalfof-ca	dotnet-webapi-onbehalfof dotnet-webapi-onbehalfof-ca

Other Microsoft Graph samples

For samples and tutorials that demonstrate different usage patterns for the Microsoft Graph API, including authentication with Azure AD, see [Microsoft Graph Community Samples & Tutorials](#).

See also

[Azure Active Directory Developer's Guide](#)

[Azure Active Directory Authentication libraries](#)

[Azure AD Graph API Conceptual and Reference](#)

[Azure AD Graph API Helper Library](#)

Why update to Microsoft identity platform (v2.0)?

10/31/2019 • 13 minutes to read • [Edit Online](#)

When developing a new application, it's important to know the differences between the Microsoft identity platform (v2.0) and Azure Active Directory (v1.0) endpoints. This article covers the main differences between the endpoints and some existing limitations for Microsoft identity platform.

NOTE

The Microsoft identity platform endpoint doesn't support all Azure AD scenarios and features. To determine if you should use the Microsoft identity platform endpoint, read about [Microsoft identity platform limitations](#).

Who can sign in

- The v1.0 endpoint allows only work and school accounts to sign in to your application (Azure AD)
- The Microsoft identity platform endpoint allows work and school accounts from Azure AD and personal Microsoft accounts (MSA), such as hotmail.com, outlook.com, and msn.com, to sign in.
- Both endpoints also accept sign-ins of *guest users* of an Azure AD directory for applications configured as *single-tenant* or for *multi-tenant* applications configured to point to the tenant-specific endpoint (
`https://login.microsoftonline.com/{TenantId_or_Name}`).

The Microsoft identity platform endpoint allows you to write apps that accept sign-ins from personal Microsoft accounts, and work and school accounts. This gives you the ability to write your app completely account-agnostic. For example, if your app calls the [Microsoft Graph](#), some additional functionality and data will be available to work accounts, such as their SharePoint sites or directory data. But for many actions, such as [Reading a user's mail](#), the same code can access the email for both personal and work and school accounts.

For Microsoft identity platform endpoint, you can use the Microsoft Authentication Library (MSAL) to gain access to the consumer, educational, and enterprise worlds. The Azure AD v1.0 endpoint accepts sign-ins from work and school accounts only.

Incremental and dynamic consent

Apps using the Azure AD v1.0 endpoint are required to specify their required OAuth 2.0 permissions in advance, for example:

▼ Microsoft Graph

Calendars.Read	Delegated	Read user calendars
Files.Read	Delegated	Read user files
offline_access	Delegated	Access user's data anytime
profile	Delegated	View users' basic profile
User.Read	Delegated	Sign in and read user profile

The permissions set directly on the application registration are **static**. 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 was difficult to create apps that could access an arbitrary number of resources.

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, which means asking for a bare minimum set of permissions upfront and growing 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. To learn more, see [permissions, consent, and scopes](#).

Allowing an app to request permissions dynamically through the `scope` parameter gives developers full control over your user's experience. You can also front load your consent experience and ask for all permissions in one initial authorization request. If your app requires a large number of permissions, you can gather those permissions from the user incrementally as they try to use certain features of the app over time.

Admin consent done on behalf of an organization still requires the static permissions registered for the app, so you should 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.

Scopes, not resources

For apps using the v1.0 endpoint, an app can behave as a **resource**, or a recipient of tokens. A resource can define a number of **scopes** or **oAuth2Permissions** that it understands, allowing client apps to request tokens from that resource for a certain set of scopes. Consider the Azure AD Graph API as an example of a resource:

- Resource identifier, or `AppID URI` : `https://graph.windows.net/`
- Scopes, or `oAuth2Permissions` : `Directory.Read` , `Directory.Write` , and so on.

This holds true for the Microsoft identity platform endpoint. An app can still behave as a resource, define scopes, and be identified by a URI. Client apps can still request access to those scopes. However, the way that a client requests those permissions have changed.

For the v1.0 endpoint, an OAuth 2.0 authorize request to Azure AD might have looked like:

```
GET https://login.microsoftonline.com/common/oauth2/authorize?  
client_id=2d4d11a2-f814-46a7-890a-274a72a7309e  
&resource=https://graph.windows.net/  
...
```

Here, the **resource** parameter indicated which resource the client app is requesting authorization. Azure AD computed the permissions required by the app based on static configuration in the Azure portal, and issued tokens accordingly.

For applications using the Microsoft identity platform endpoint, the same OAuth 2.0 authorize request looks like:

```
GET https://login.microsoftonline.com/common/oauth2/v2.0/authorize?  
client_id=2d4d11a2-f814-46a7-890a-274a72a7309e  
&scope=https://graph.windows.net/directory.read%20https://graph.windows.net/directory.write  
...
```

Here, the **scope** parameter indicates which resource and permissions the app is requesting authorization. The desired resource is still present in the request - it's encompassed in each of the values of the scope parameter. Using the scope parameter in this manner allows the Microsoft identity platform endpoint to be more compliant with the OAuth 2.0 specification, and aligns more closely with common industry practices. It also enables apps to do [incremental consent](#) - only requesting permissions when the application requires them as opposed to up front.

Well-known scopes

Offline access

Apps using the Microsoft identity platform endpoint may require the use of a new well-known permission for apps - the `offline_access` scope. All apps will need to request this permission if they need to access resources on the behalf of a user for a prolonged period of time, even when the user may not be actively using the app. The `offline_access` scope will appear to the user in consent dialogs as **Access your data anytime**, which the user must agree to. Requesting the `offline_access` permission will enable your web app to receive OAuth 2.0 refresh_tokens from the Microsoft identity platform endpoint. Refresh tokens are long-lived, and can be exchanged for new OAuth 2.0 access tokens for extended periods of access.

If your app doesn't request the `offline_access` scope, it won't receive refresh tokens. This means that when you redeem an authorization code in the OAuth 2.0 authorization code flow, you'll only receive back an access token from the `/token` endpoint. That access token remains valid for a short period of time (typically one hour), but will eventually expire. At that point in time, your app will need to redirect the user back to the `/authorize` endpoint to retrieve a new authorization code. During this redirect, the user may or may not need to enter their credentials again or reconsent to permissions, depending on the type of app.

To learn more about OAuth 2.0, `refresh_tokens`, and `access_tokens`, check out the [Microsoft identity platform protocol reference](#).

OpenID, profile, and email

Historically, the most basic OpenID Connect sign-in flow with Microsoft identity platform would provide a lot of information about the user in the resulting `id_token`. The claims in an `id_token` can include the user's name, preferred username, email address, object ID, and more.

The information that the `openid` scope affords your app access to is now restricted. The `openid` scope will only allow your app to sign in the user and receive an app-specific identifier for the user. If you want to get personal data about the user in your app, your app needs to request additional permissions from the user. Two new scopes, `email` and `profile`, will allow you to request additional permissions.

- The `email` scope allows your app access to the user's primary email address through the `email` claim in the `id_token`, assuming the user has an addressable email address.
- The `profile` scope affords your app access to all other basic information about the user, such as their name, preferred username, object ID, and so on, in the `id_token`.

These scopes allow you to code your app in a minimal-disclosure fashion so you can only ask the user for the set of information that your app needs to do its job. For more information on these scopes, see [the Microsoft identity platform scope reference](#).

Token claims

The Microsoft identity platform endpoint issues a smaller set of claims in its tokens by default to keep payloads

small. If you have apps and services that have a dependency on a particular claim in a v1.0 token that is no longer provided by default in a Microsoft identity platform token, consider using the [optional claims](#) feature to include that claim.

IMPORTANT

v1.0 and v2.0 tokens can be issued by both the v1.0 and v2.0 endpoints! id_tokens *always* match the endpoint they're requested from, and access tokens *always* match the format expected by the Web API your client will call using that token. So if your app uses the v2.0 endpoint to get a token to call Microsoft Graph, which expects v1.0 format access tokens, your app will receive a token in the v1.0 format.

Limitations

There are a few restrictions to be aware of when using Microsoft identity platform.

When you build applications that integrate with the Microsoft identity platform, you need to decide whether the Microsoft identity platform endpoint and authentication protocols meet your needs. The v1.0 endpoint and platform is still fully supported and, in some respects, is more feature rich than Microsoft identity platform. However, Microsoft identity platform [introduces significant benefits](#) for developers.

Here's a simplified recommendation for developers now:

- If you want or need to support personal Microsoft accounts in your application, or you're writing a new application, use Microsoft identity platform. But before you do, make sure you understand the limitations discussed in this article.
- If you're migrating or updating an application that relies on SAML, you can't use Microsoft identity platform. Instead, refer to the [Azure AD v1.0 guide](#).

The Microsoft identity platform endpoint will evolve to eliminate the restrictions listed here, so that you'll only ever need to use the Microsoft identity platform endpoint. In the meantime, use this article to determine whether the Microsoft identity platform endpoint is right for you. We'll continue to update this article to reflect the current state of the Microsoft identity platform endpoint. Check back to reevaluate your requirements against Microsoft identity platform capabilities.

Restrictions on app registrations

For each app that you want to integrate with the Microsoft identity platform endpoint, you can create an app registration in the new [App registrations experience](#) in the Azure portal. Existing Microsoft account apps aren't compatible with the portal, but all Azure AD apps are, regardless of where or when they were registered.

App registrations that support work and school accounts and personal accounts have the following caveats:

- Only two app secrets are allowed per application ID.
- An application that wasn't registered in a tenant can only be managed by the account that registered it. It can't be shared with other developers. This is the case for most apps that were registered using a personal Microsoft account in the App Registration Portal. If you'd like to share your app registration with multiple developers, register the application in a tenant using the new [App registrations](#) section of the Azure portal.
- There are several restrictions on the format of the redirect URL that is allowed. For more information about redirect URL, see the next section.

Restrictions on redirect URLs

Apps that are registered for Microsoft identity platform are restricted to a limited set of redirect URL values. The redirect URL for web apps and services must begin with the scheme `https`, and all redirect URL values must share a single DNS domain. The registration system compares the whole DNS name of the existing redirect URL to the DNS name of the redirect URL that you're adding. `http://localhost` is also supported as a redirect URL.

The request to add the DNS name will fail if either of the following conditions is true:

- The whole DNS name of the new redirect URL doesn't match the DNS name of the existing redirect URL.
- The whole DNS name of the new redirect URL isn't a subdomain of the existing redirect URL.

Example 1

If the app has a redirect URL of `https://login.contoso.com`, you can add a redirect URL where the DNS name matches exactly, as shown in the following example:

`https://login.contoso.com/new`

Or, you can refer to a DNS subdomain of login.contoso.com, as shown in the following example:

`https://new.login.contoso.com`

Example 2

If you want to have an app that has `login-east.contoso.com` and `login-west.contoso.com` as redirect URLs, you must add those redirect URLs in the following order:

`https://contoso.com`

`https://login-east.contoso.com`

`https://login-west.contoso.com`

You can add the latter two because they're subdomains of the first redirect URL, contoso.com.

You can have only 20 reply URLs for a particular application - this limit applies across all app types that the registration supports (single-page application (SPA), native client, web app, and service).

To learn how to register an app for use with Microsoft identity platform, see [Register an app using the new App registrations experience](#).

Restrictions on libraries and SDKs

Currently, library support for the Microsoft identity platform endpoint is limited. If you want to use the Microsoft identity platform endpoint in a production application, you have these options:

- If you're building a web application, you can safely use the generally available server-side middleware to do sign-in and token validation. These include the OWIN OpenID Connect middleware for ASP.NET and the Node.js Passport plug-in. For code samples that use Microsoft middleware, see the [Microsoft identity platform getting started](#) section.
- If you're building a desktop or mobile application, you can use one of the Microsoft Authentication Libraries (MSAL). These libraries are generally available or in a production-supported preview, so it is safe to use them in production applications. You can read more about the terms of the preview and the available libraries in [authentication libraries reference](#).
- For platforms not covered by Microsoft libraries, you can integrate with the Microsoft identity platform endpoint by directly sending and receiving protocol messages in your application code. The OpenID Connect and OAuth protocols [are explicitly documented](#) to help you do such an integration.
- Finally, you can use open-source OpenID Connect and OAuth libraries to integrate with the Microsoft identity platform endpoint. The Microsoft identity platform endpoint should be compatible with many open-source protocol libraries without changes. The availability of these kinds of libraries varies by language and platform. The [OpenID Connect](#) and [OAuth 2.0](#) websites maintain a list of popular implementations. For more information, see [Microsoft identity platform and authentication libraries](#), and the list of open-source client libraries and samples that have been tested with the Microsoft identity platform endpoint.
- For reference, the `.well-known` endpoint for the Microsoft identity platform common endpoint is `https://login.microsoftonline.com/common/v2.0/.well-known/openid-configuration`. Replace `common` with your tenant ID to get data specific to your tenant.

Protocol changes

The Microsoft identity platform endpoint does not support SAML or WS-Federation; it only supports OpenID Connect and OAuth 2.0. The notable changes to the OAuth 2.0 protocols from the v1.0 endpoint are:

- The `email` claim is returned if an optional claim is configured **or** `scope=email` was specified in the request.
- The `scope` parameter is now supported in place of the `resource` parameter.
- Many responses have been modified to make them more compliant with the OAuth 2.0 specification, for example, correctly returning `expires_in` as an int instead of a string.

To better understand the scope of protocol functionality supported in the Microsoft identity platform endpoint, see [OpenID Connect and OAuth 2.0 protocol reference](#).

SAML restrictions

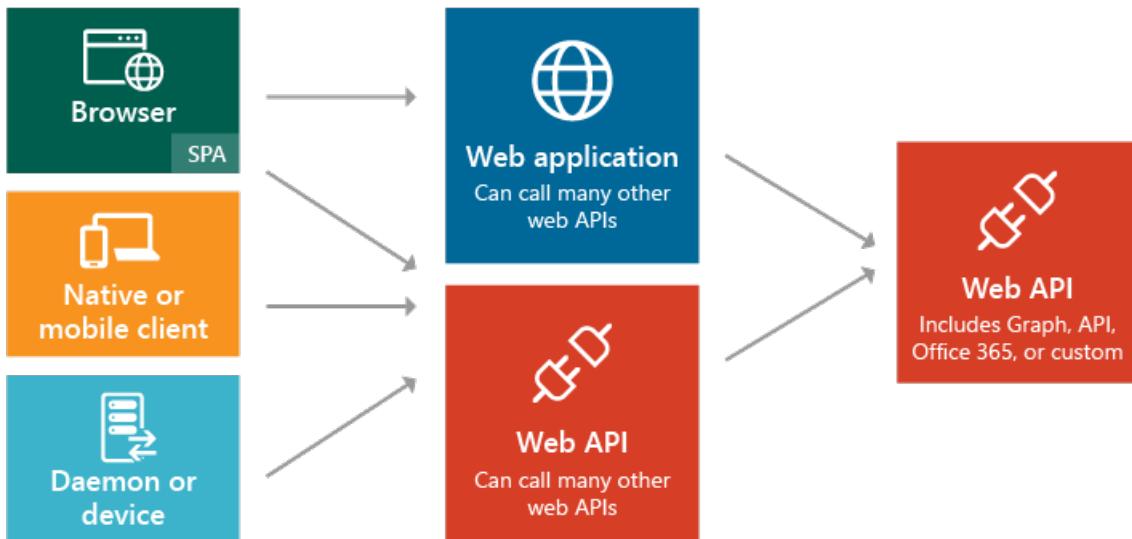
If you've used Active Directory Authentication Library (ADAL) in Windows applications, you might have taken advantage of Windows Integrated authentication, which uses the Security Assertion Markup Language (SAML) assertion grant. With this grant, users of federated Azure AD tenants can silently authenticate with their on-premises Active Directory instance without entering credentials. The SAML assertion grant isn't supported on the Microsoft identity platform endpoint.

Application types in v1.0

10/15/2019 • 5 minutes to read • [Edit Online](#)

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.

Follow the links to learn more about each type of app and understand the high-level scenarios before you start working with the code. You can also learn about the differences you need to know when writing a particular app that works with the v1.0 endpoint or v2.0 endpoint.

NOTE

The v2.0 endpoint doesn't support all Azure AD scenarios and features. To determine whether you should use the v2.0 endpoint, read about [v2.0 limitations](#).

You can develop any of the apps and scenarios described here using various languages and platforms. They are all backed by complete code samples available in the code samples guide: [v1.0 code samples by scenario](#) and [v2.0 code samples by scenario](#). You can also download the code samples directly from the corresponding [GitHub sample repositories](#).

In addition, if your application needs a specific piece or segment of an end-to-end scenario, in most cases that functionality can be added independently. For example, if you have a native application that calls a web API, you can easily add a web application that also calls the web API.

App registration

Registering an app that uses the Azure AD v1.0 endpoint

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.

For details, learn how to [register an app](#).

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. For more information, see [Overview of the Consent Framework](#).

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.

Each scenario listed in this document includes a subsection that describes its provisioning requirements. For more in-depth information about provisioning an application in Azure AD and the differences between single and multi-tenant applications, see [Integrating applications with Azure Active Directory](#) for more information. Continue reading to understand the common application scenarios in Azure AD.

Next steps

- Learn more about other Azure AD [authentication basics](#)

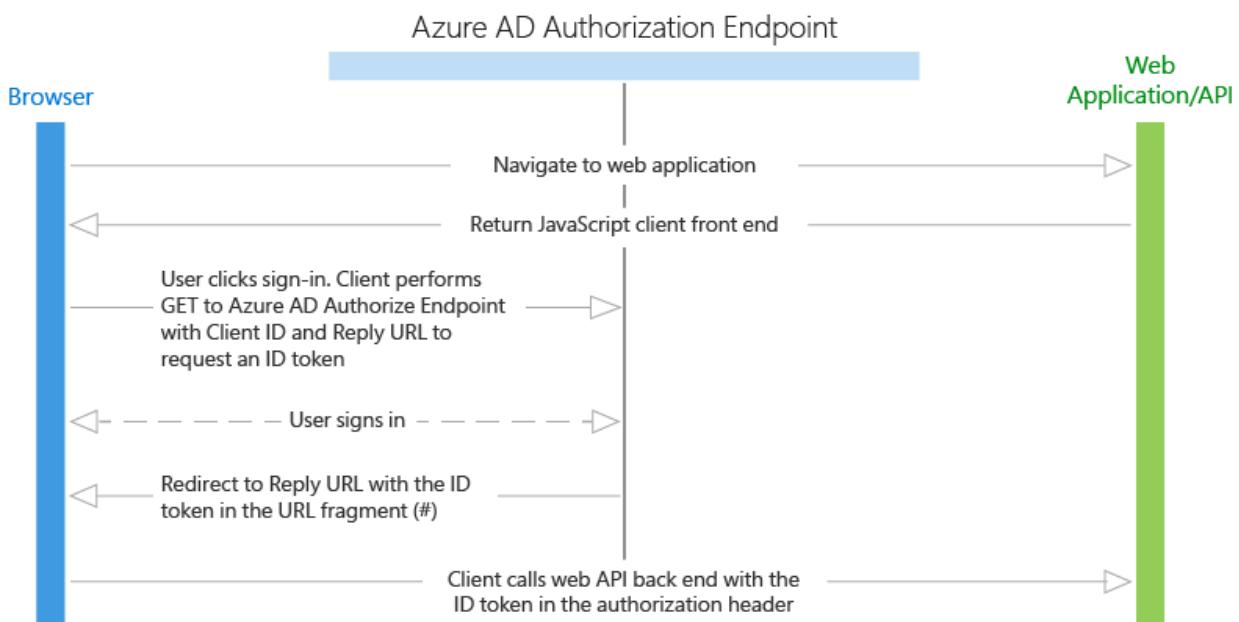
Single-page applications

10/15/2019 • 3 minutes to read • [Edit Online](#)

Single-page applications (SPAs) are typically structured as a JavaScript presentation layer (front end) that runs in the browser, and a web API back end that runs on a server and implements the application's business logic. To learn more about the implicit authorization grant, and help you decide whether it's right for your application scenario, see [Understanding the OAuth2 implicit grant flow in Azure Active Directory](#).

In this scenario, when the user signs in, the JavaScript front end uses [Active Directory Authentication Library for JavaScript \(ADAL.js\)](#) and the implicit authorization grant to obtain an ID token (`id_token`) from Azure AD. The token is cached and the client attaches it to the request as the bearer token when making calls to its Web API back end, which is secured using the OWIN middleware.

Diagram



Protocol flow

1. The user navigates to the web application.
2. The application returns the JavaScript front end (presentation layer) to the browser.
3. The user initiates sign in, for example by clicking a sign-in link. The browser sends a GET to the Azure AD authorization endpoint to request an ID token. This request includes the application ID and reply URL in the query parameters.
4. Azure AD validates the Reply URL against the registered Reply URL that was configured in the Azure portal.
5. The user signs in on the sign-in page.
6. If authentication is successful, Azure AD creates an ID token and returns it as a URL fragment (#) to the application's Reply URL. For a production application, this Reply URL should be HTTPS. The returned token includes claims about the user and Azure AD that are required by the application to validate the token.
7. The JavaScript client code running in the browser extracts the token from the response to use in securing calls to the application's web API back end.
8. The browser calls the application's web API back end with the ID token in the authorization header. The Azure AD authentication service issues an ID token that can be used as a bearer token if the resource is the same as the client ID (in this case, this is true as the web API is the app's own backend).

Code samples

See the [code samples for single-page application scenarios](#). Be sure to check back frequently as new samples are added frequently.

App registration

- Single tenant - If you are building an application just for your organization, it must be registered in your company's directory by using the Azure portal.
- Multi-tenant - If you are building an application that can be used by users outside your organization, it must be registered in your company's directory, but also must be registered in each organization's directory that will be using the application. To make your application available in their directory, you can include a sign-up process for your customers that enables them to consent to your application. When they sign up for your application, they will be presented with a dialog that shows the permissions the application requires, and then the option to consent. Depending on the required permissions, an administrator in the other organization may be required to give consent. When the user or administrator consents, the application is registered in their directory.

After registering the application, it must be configured to use OAuth 2.0 implicit grant protocol. By default, this protocol is disabled for applications. To enable the OAuth2 implicit grant protocol for your application, edit its application manifest from the Azure portal and set the "oauth2AllowImplicitFlow" value to true. For more info, see [Application manifest](#).

Token expiration

Using ADAL.js helps with:

- Refreshing an expired token
- Requesting an access token to call a web API resource

After a successful authentication, Azure AD writes a cookie in the user's browser to establish a session. Note the session exists between the user and Azure AD (not between the user and the web application). When a token expires, ADAL.js uses this session to silently obtain another token. ADAL.js uses a hidden iFrame to send and receive the request using the OAuth implicit grant protocol. ADAL.js can also use this same mechanism to silently obtain access tokens for other web API resources the application calls as long as these resources support cross-origin resource sharing (CORS), are registered in the user's directory, and any required consent was given by the user during sign-in.

Next steps

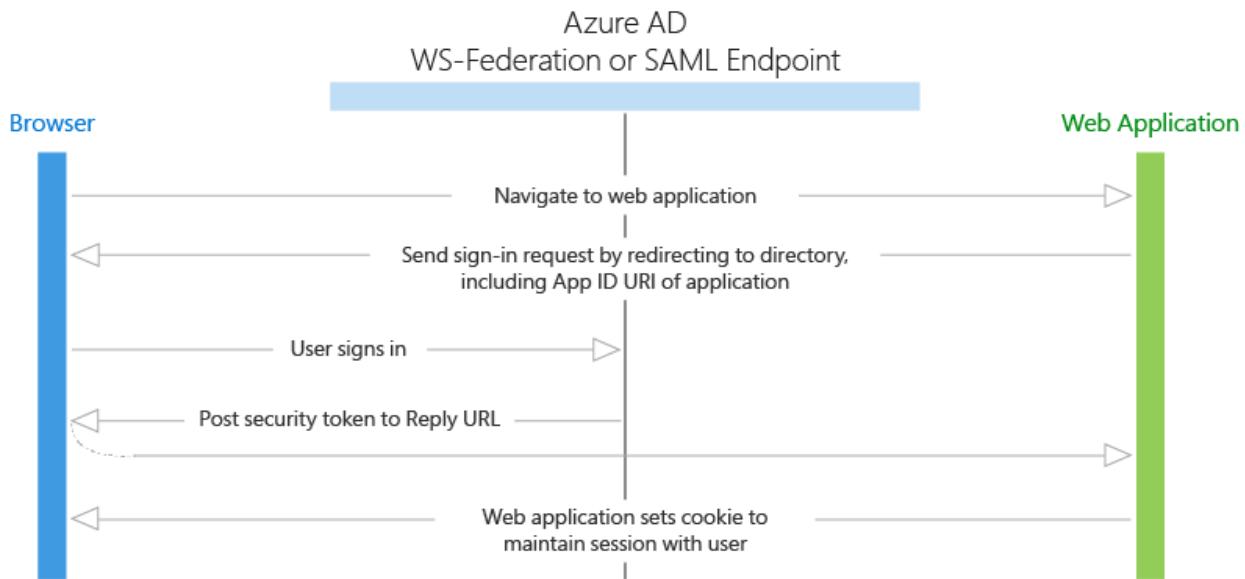
- Learn more about other [Application types and scenarios](#)
- Learn about the Azure AD [authentication basics](#)

Web apps

10/15/2019 • 2 minutes to read • [Edit Online](#)

Web apps are applications that authenticate a user in a web browser to a web application. In this scenario, the web application directs the user's browser to sign them in to Azure AD. Azure AD returns a sign-in response through the user's browser, which contains claims about the user in a security token. This scenario supports sign-on using the OpenID Connect, SAML 2.0, and WS-Federation protocols.

Diagram



Protocol flow

1. When a user visits the application and needs to sign in, they are redirected via a sign-in request to the authentication endpoint in Azure AD.
2. The user signs in on the sign-in page.
3. If authentication is successful, Azure AD creates an authentication token and returns a sign-in response to the application's Reply URL that was configured in the Azure portal. For a production application, this Reply URL should be HTTPS. The returned token includes claims about the user and Azure AD that are required by the application to validate the token.
4. The application validates the token by using a public signing key and issuer information available at the federation metadata document for Azure AD. After the application validates the token, it starts a new session with the user. This session allows the user to access the application until it expires.

Code samples

See the code samples for web browser to web application scenarios. And, check back frequently as new samples are added frequently.

App registration

To register a web app, see [Register an app](#).

- Single tenant - If you are building an application just for your organization, it must be registered in your company's directory by using the Azure portal.

- Multi-tenant - If you are building an application that can be used by users outside your organization, it must be registered in your company's directory, but also must be registered in each organization's directory that will be using the application. To make your application available in their directory, you can include a sign-up process for your customers that enables them to consent to your application. When they sign up for your application, they will be presented with a dialog that shows the permissions the application requires, and then the option to consent. Depending on the required permissions, an administrator in the other organization may be required to give consent. When the user or administrator consents, the application is registered in their directory.

Token expiration

The user's session expires when the lifetime of the token issued by Azure AD expires. Your application can shorten this time period if desired, such as signing out users based on a period of inactivity. When the session expires, the user will be prompted to sign in again.

Next steps

- Learn more about other [Application types and scenarios](#)
- Learn about the Azure AD [authentication basics](#)

Web API

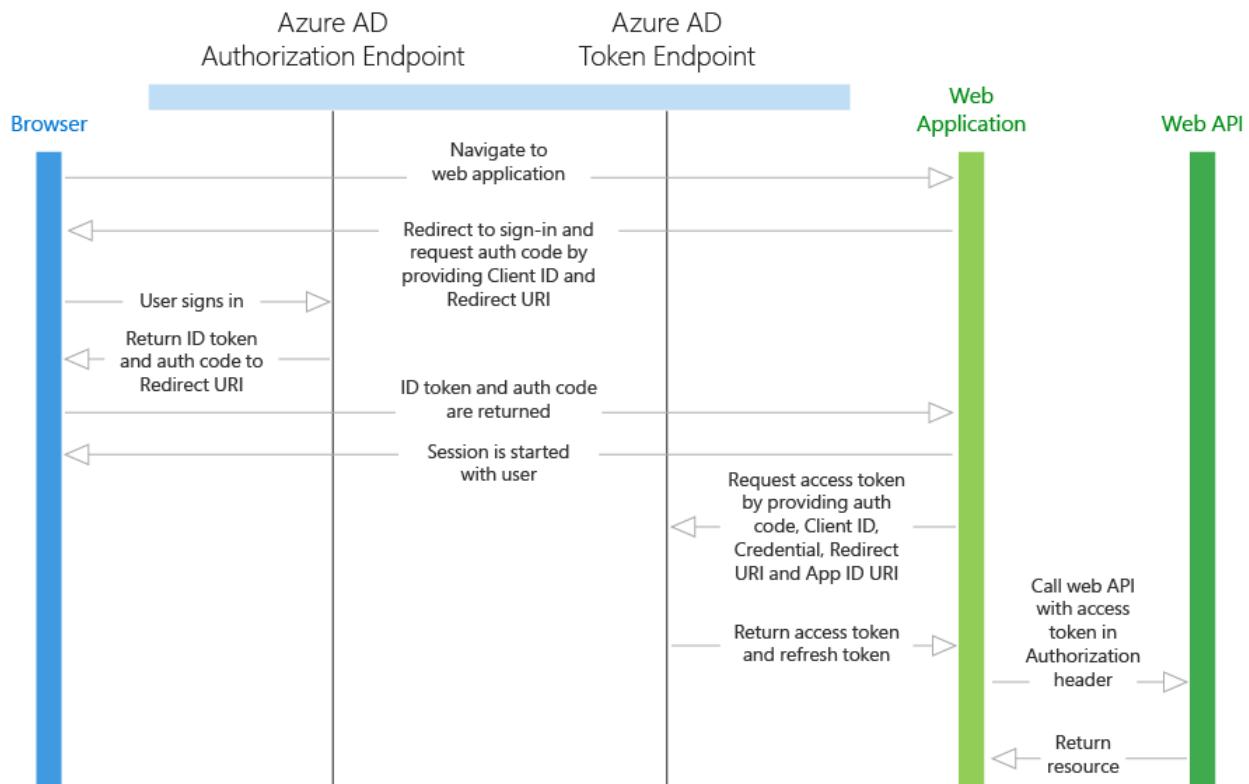
10/15/2019 • 6 minutes to read • [Edit Online](#)

Web API apps are web applications that need to get resources from a web API. In this scenario, there are two identity types that the web application can use to authenticate and call the web API:

- **Application identity** - This scenario uses OAuth 2.0 client credentials grant to authenticate as the application and access the web API. When using an application identity, the web API can only detect that the web application is calling it, as the web API does not receive any information about the user. If the application receives information about the user, it will be sent via the application protocol, and it is not signed by Azure AD. The web API trusts that the web application authenticated the user. For this reason, this pattern is called a trusted subsystem.
- **Delegated user identity** - This scenario can be accomplished in two ways: OpenID Connect, and OAuth 2.0 authorization code grant with a confidential client. The web application obtains an access token for the user, which proves to the web API that the user successfully authenticated to the web application and that the web application was able to obtain a delegated user identity to call the web API. This access token is sent in the request to the web API, which authorizes the user and returns the desired resource.

Both the application identity and delegated user identity types are discussed in the flow below. The key difference between them is that the delegated user identity must first acquire an authorization code before the user can sign in and gain access to the web API.

Diagram



Protocol flow

Application identity with OAuth 2.0 client credentials grant

1. A user is signed in to Azure AD in the web application (see the **Web apps** section for more info).

2. The web application needs to acquire an access token so that it can authenticate to the web API and retrieve the desired resource. It makes a request to Azure AD's token endpoint, providing the credential, application ID, and web API's application ID URI.
3. Azure AD authenticates the application and returns a JWT access token that is used to call the web API.
4. Over HTTPS, the web application uses the returned JWT access token to add the JWT string with a "Bearer" designation in the Authorization header of the request to the web API. The web API then validates the JWT token, and if validation is successful, returns the desired resource.

Delegated user identity with OpenID Connect

1. A user is signed in to a web application using Azure AD (see the Web Browser to Web Application section above). If the user of the web application has not yet consented to allowing the web application to call the web API on its behalf, the user will need to consent. The application will display the permissions it requires, and if any of these are administrator-level permissions, a normal user in the directory will not be able to consent. This consent process only applies to multi-tenant applications, not single tenant applications, as the application will already have the necessary permissions. When the user signed in, the web application received an ID token with information about the user, as well as an authorization code.
2. Using the authorization code issued by Azure AD, the web application sends a request to Azure AD's token endpoint that includes the authorization code, details about the client application (Application ID and redirect URI), and the desired resource (application ID URI for the web API).
3. The authorization code and information about the web application and web API are validated by Azure AD. Upon successful validation, Azure AD returns two tokens: a JWT access token and a JWT refresh token.
4. Over HTTPS, the web application uses the returned JWT access token to add the JWT string with a "Bearer" designation in the Authorization header of the request to the web API. The web API then validates the JWT token, and if validation is successful, returns the desired resource.

Delegated user identity with OAuth 2.0 authorization code grant

1. A user is already signed in to a web application, whose authentication mechanism is independent of Azure AD.
2. The web application requires an authorization code to acquire an access token, so it issues a request through the browser to Azure AD's authorization endpoint, providing the Application ID and redirect URI for the web application after successful authentication. The user signs in to Azure AD.
3. If the user of the web application has not yet consented to allowing the web application to call the web API on its behalf, the user will need to consent. The application will display the permissions it requires, and if any of these are administrator-level permissions, a normal user in the directory will not be able to consent. This consent applies to both single and multi-tenant application. In the single tenant case, an admin can perform admin consent to consent on behalf of their users. This can be done using the [Grant Permissions](#) button in the [Azure portal](#).
4. After the user has consented, the web application receives the authorization code that it needs to acquire an access token.
5. Using the authorization code issued by Azure AD, the web application sends a request to Azure AD's token endpoint that includes the authorization code, details about the client application (Application ID and redirect URI), and the desired resource (application ID URI for the web API).
6. The authorization code and information about the web application and web API are validated by Azure AD. Upon successful validation, Azure AD returns two tokens: a JWT access token and a JWT refresh token.
7. Over HTTPS, the web application uses the returned JWT access token to add the JWT string with a "Bearer" designation in the Authorization header of the request to the web API. The web API then validates the JWT token, and if validation is successful, returns the desired resource.

Code samples

See the code samples for Web Application to Web API scenarios. And, check back frequently -- new samples are added frequently. Web [Application to Web API](#).

App registration

To register an application with the Azure AD v1.0 endpoint, see [Register an app](#).

- Single tenant - For both the application identity and delegated user identity cases, the web application and the web API must be registered in the same directory in Azure AD. The web API can be configured to expose a set of permissions, which are used to limit the web application's access to its resources. If a delegated user identity type is being used, the web application needs to select the desired permissions from the **Permissions to other applications** drop-down menu in the Azure portal. This step is not required if the application identity type is being used.
- Multi-tenant - First, the web application is configured to indicate the permissions it requires to be functional. This list of required permissions is shown in a dialog when a user or administrator in the destination directory gives consent to the application, which makes it available to their organization. Some applications only require user-level permissions, which any user in the organization can consent to. Other applications require administrator-level permissions, which a user in the organization cannot consent to. Only a directory administrator can give consent to applications that require this level of permissions. When the user or administrator consents, the web application and the web API are both registered in their directory.

Token expiration

When the web application uses its authorization code to get a JWT access token, it also receives a JWT refresh token. When the access token expires, the refresh token can be used to reauthenticate the user without requiring them to sign in again. This refresh token is then used to authenticate the user, which results in a new access token and refresh token.

Next steps

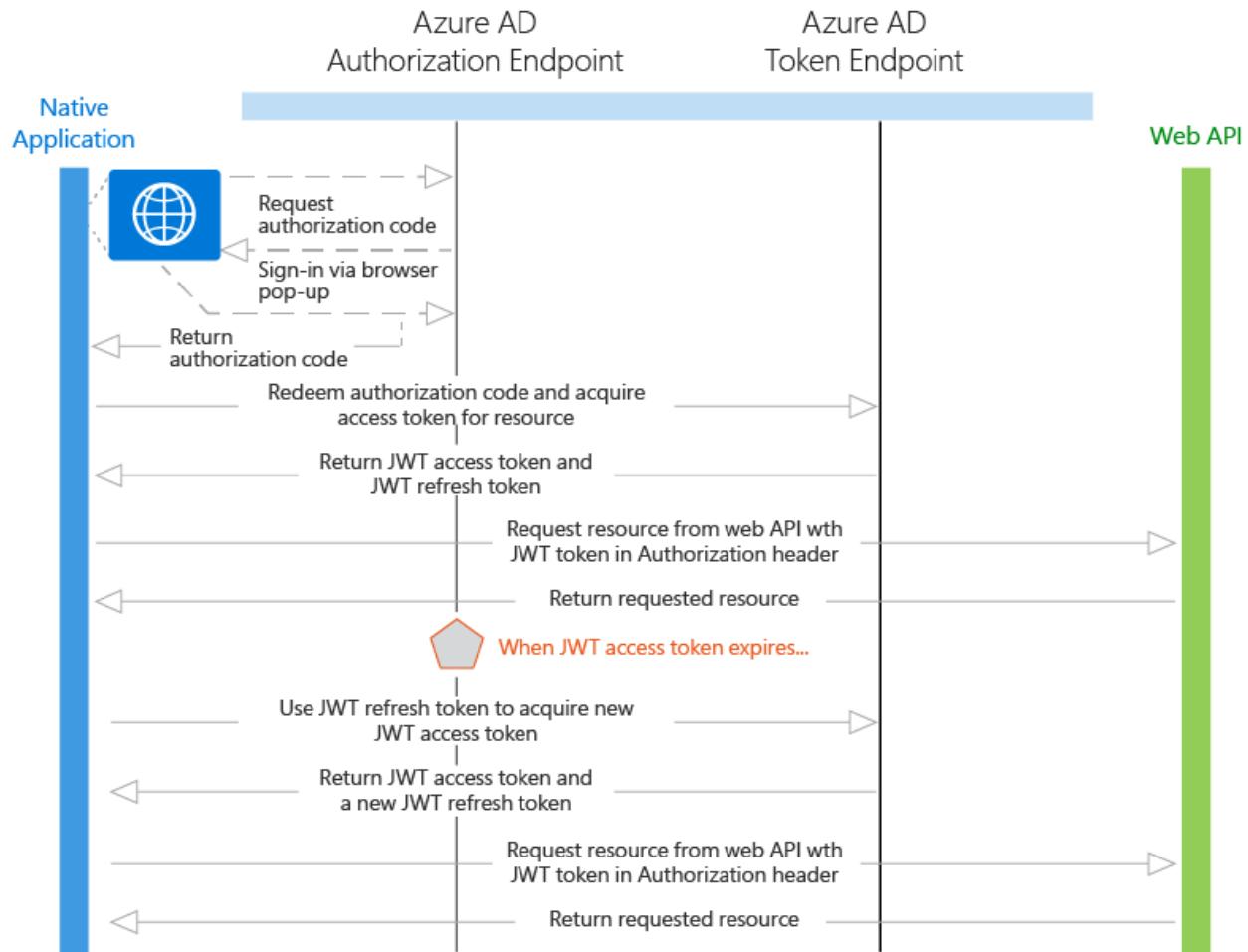
- Learn more about other [Application types and scenarios](#)
- Learn about the Azure AD [authentication basics](#)

Native apps

10/15/2019 • 4 minutes to read • [Edit Online](#)

Native apps are applications that call a web API on behalf of a user. This scenario is built on the OAuth 2.0 authorization code grant type with a public client, as described in section 4.1 of the [OAuth 2.0 specification](#). The native application obtains an access token for the user by using the OAuth 2.0 protocol. This access token is then sent in the request to the web API, which authorizes the user and returns the desired resource.

Diagram



Protocol flow

If you are using the AD Authentication Libraries, most of the protocol details described below are handled for you, such as the browser pop-up, token caching, and handling of refresh tokens.

1. Using a browser pop-up, the native application makes a request to the authorization endpoint in Azure AD. This request includes the Application ID and the redirect URI of the native application as shown in the Azure portal, and the application ID URI for the web API. If the user hasn't already signed in, they are prompted to sign in again.
2. Azure AD authenticates the user. If it is a multi-tenant application and consent is required to use the application, the user will be required to consent if they haven't already done so. After granting consent and upon successful authentication, Azure AD issues an authorization code response back to the client application's redirect URI.
3. When Azure AD issues an authorization code response back to the redirect URI, the client application stops browser interaction and extracts the authorization code from the response. Using this authorization code, the

client application sends a request to Azure AD's token endpoint that includes the authorization code, details about the client application (Application ID and redirect URI), and the desired resource (application ID URI for the web API).

4. The authorization code and information about the client application and web API are validated by Azure AD. Upon successful validation, Azure AD returns two tokens: a JWT access token and a JWT refresh token. In addition, Azure AD returns basic information about the user, such as their display name and tenant ID.
5. Over HTTPS, the client application uses the returned JWT access token to add the JWT string with a "Bearer" designation in the Authorization header of the request to the web API. The web API then validates the JWT token, and if validation is successful, returns the desired resource.
6. When the access token expires, the client application will receive an error that indicates the user needs to authenticate again. If the application has a valid refresh token, it can be used to acquire a new access token without prompting the user to sign in again. If the refresh token expires, the application will need to interactively authenticate the user once again.

NOTE

The refresh token issued by Azure AD can be used to access multiple resources. For example, if you have a client application that has permission to call two web APIs, the refresh token can be used to get an access token to the other web API as well.

Code samples

See the code samples for Native Application to Web API scenarios. And, check back frequently -- we add new samples frequently. [Native Application to Web API](#).

App registration

To register an application with the Azure AD v1.0 endpoint, see [Register an app](#).

- Single tenant - Both the native application and the web API must be registered in the same directory in Azure AD. The web API can be configured to expose a set of permissions, which are used to limit the native application's access to its resources. The client application then selects the desired permissions from the "Permissions to Other Applications" drop-down menu in the Azure portal.
- Multi-tenant - First, the native application only ever registered in the developer or publisher's directory. Second, the native application is configured to indicate the permissions it requires to be functional. This list of required permissions is shown in a dialog when a user or administrator in the destination directory gives consent to the application, which makes it available to their organization. Some applications only require user-level permissions, which any user in the organization can consent to. Other applications require administrator-level permissions, which a user in the organization cannot consent to. Only a directory administrator can give consent to applications that require this level of permissions. When the user or administrator consents, only the web API is registered in their directory.

Token expiration

When the native application uses its authorization code to get a JWT access token, it also receives a JWT refresh token. When the access token expires, the refresh token can be used to re-authenticate the user without requiring them to sign in again. This refresh token is then used to authenticate the user, which results in a new access token and refresh token.

Next steps

- Learn more about other [Application types and scenarios](#)
- Learn about the [Azure AD authentication basics](#)

Service-to-service apps

10/15/2019 • 4 minutes to read • [Edit Online](#)

Service-to-service applications can be a daemon or server application that needs to get resources from a web API. There are two sub-scenarios that apply to this section:

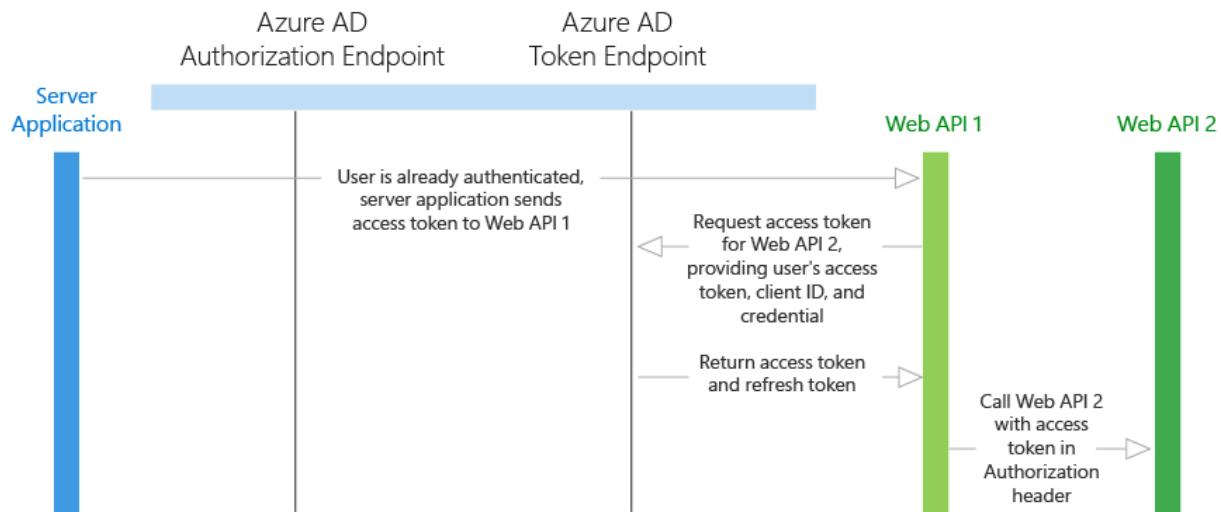
- A daemon that needs to call a web API, built on OAuth 2.0 client credentials grant type

In this scenario, it's important to understand a few things. First, user interaction is not possible with a daemon application, which requires the application to have its own identity. An example of a daemon application is a batch job, or an operating system service running in the background. This type of application requests an access token by using its application identity and presenting its Application ID, credential (password or certificate), and application ID URI to Azure AD. After successful authentication, the daemon receives an access token from Azure AD, which is then used to call the web API.

- A server application (such as a web API) that needs to call a web API, built on OAuth 2.0 On-Behalf-Of draft specification

In this scenario, imagine that a user has authenticated on a native application, and this native application needs to call a web API. Azure AD issues a JWT access token to call the web API. If the web API needs to call another downstream web API, it can use the on-behalf-of flow to delegate the user's identity and authenticate to the second-tier web API.

Diagram



DProtocol flow

Application identity with OAuth 2.0 client credentials grant

1. First, the server application needs to authenticate with Azure AD as itself, without any human interaction such as an interactive sign-on dialog. It makes a request to Azure AD's token endpoint, providing the credential, Application ID, and application ID URI.
2. Azure AD authenticates the application and returns a JWT access token that is used to call the web API.
3. Over HTTPS, the web application uses the returned JWT access token to add the JWT string with a "Bearer" designation in the Authorization header of the request to the web API. The web API then validates the JWT token, and if validation is successful, returns the desired resource.

Delegated user identity with OAuth 2.0 On-Behalf-Of Draft Specification

The flow discussed below assumes that a user has been authenticated on another application (such as a native application), and their user identity has been used to acquire an access token to the first-tier web API.

1. The native application sends the access token to the first-tier web API.
2. The first-tier web API sends a request to Azure AD's token endpoint, providing its Application ID and credentials, as well as the user's access token. In addition, the request is sent with an `on_behalf_of` parameter that indicates the web API is requesting new tokens to call a downstream web API on behalf of the original user.
3. Azure AD verifies that the first-tier web API has permissions to access the second-tier web API and validates the request, returning a JWT access token and a JWT refresh token to the first-tier web API.
4. Over HTTPS, the first-tier web API then calls the second-tier web API by appending the token string in the Authorization header in the request. The first-tier web API can continue to call the second-tier web API as long as the access token and refresh tokens are valid.

Code samples

See the code samples for Daemon or Server Application to Web API scenarios. And, check back frequently as new samples are added frequently. [Server or Daemon Application to Web API](#)

App registration

- Single tenant - For both the application identity and delegated user identity cases, the daemon or server application must be registered in the same directory in Azure AD. The web API can be configured to expose a set of permissions, which are used to limit the daemon or server's access to its resources. If a delegated user identity type is being used, the server application needs to select the desired permissions from the "Permissions to Other Applications" drop-down menu in the Azure portal. This step is not required if the application identity type is being used.
- Multi-tenant - First, the daemon or server application is configured to indicate the permissions it requires to be functional. This list of required permissions is shown in a dialog when a user or administrator in the destination directory gives consent to the application, which makes it available to their organization. Some applications only require user-level permissions, which any user in the organization can consent to. Other applications require administrator-level permissions, which a user in the organization cannot consent to. Only a directory administrator can give consent to applications that require this level of permissions. When the user or administrator consents, both of the web APIs are registered in their directory.

Token expiration

When the first application uses its authorization code to get a JWT access token, it also receives a JWT refresh token. When the access token expires, the refresh token can be used to re-authenticate the user without prompting for credentials. This refresh token is then used to authenticate the user, which results in a new access token and refresh token.

Next steps

- Learn more about other [Application types and scenarios](#)
- Learn about the Azure AD [authentication basics](#)

What is authentication?

10/30/2019 • 7 minutes to read • [Edit Online](#)

Authentication is the act of challenging a party for legitimate credentials, providing the basis for creation of a security principal to be used for identity and access control. In simpler terms, it's the process of proving you are who you say you are. Authentication is sometimes shortened to AuthN.

Authorization is the act of granting an authenticated security principal permission to do something. It specifies what data you're allowed to access and what you can do with it. Authorization is sometimes shortened to AuthZ.

Microsoft identity platform simplifies authentication for application developers by providing identity as a service, with support for industry-standard protocols such as OAuth 2.0 and OpenID Connect, as well as open-source libraries for different platforms to help you start coding quickly.

There are two primary use cases in the Microsoft identity platform programming model:

- During an OAuth 2.0 authorization grant flow - when the resource owner grants authorization to the client application, allowing the client to access the resource owner's resources.
- During resource access by the client - as implemented by the resource server, using the claims values present in the access token to make access control decisions based upon them.

Authentication basics in Microsoft identity platform

Consider the most basic scenario where identity is required: a user in a web browser needs to authenticate to a web application. The following diagram shows this scenario:

Here's what you need to know about the various components shown in the diagram:

- Microsoft identity platform is the identity provider. The identity provider is responsible for verifying the identity of users and applications that exist in an organization's directory, and issues security tokens upon successful authentication of those users and applications.
- An application that wants to outsource authentication to Microsoft identity platform must be registered in Azure Active Directory (Azure AD). Azure AD registers and uniquely identifies the app in the directory.
- Developers can use the open-source Microsoft identity platform authentication libraries to make authentication easy by handling the protocol details for you. For more info, see Microsoft identity platform [v2.0 authentication libraries](#) and [v1.0 authentication libraries](#).
- Once a user has been authenticated, the application must validate the user's security token to ensure that authentication was successful. You can find quickstarts, tutorials, and code samples in a variety of languages and frameworks which show what the application must do.
 - To quickly build an app and add functionality like getting tokens, refreshing tokens, signing in a user, displaying some user info, and more, see the **Quickstarts** section of the documentation.
 - To get in-depth, scenario-based procedures for top auth developer tasks like obtaining access tokens and using them in calls to the Microsoft Graph API and other APIs, implementing sign-in with Microsoft with a traditional web browser-based app using OpenID Connect, and more, see the **Tutorials** section of the documentation.
 - To download code samples, go to [GitHub](#).
- The flow of requests and responses for the authentication process is determined by the authentication protocol that you used, such as OAuth 2.0, OpenID Connect, WS-Federation, or SAML 2.0. For more info about protocols, see the **Concepts > Authentication protocol** section of the documentation.

In the example scenario above, you can classify the apps according to these two roles:

- Apps that need to securely access resources
- Apps that play the role of the resource itself

How each flow emits tokens and codes

Depending on how your client is built, it can use one (or several) of the authentication flows supported by the Microsoft identity platform. These flows can produce a variety of tokens (id_tokens, refresh tokens, access tokens) as well as authorization codes, and require different tokens to make them work. This chart provides an overview:

FLOW	REQUIRES	ID_TOKEN	ACCESS TOKEN	REFRESH TOKEN	AUTHORIZATION CODE
Authorization code flow		x	x	x	x
Implicit flow		x	x		
Hybrid OIDC flow		x			x
Refresh token redemption	refresh token	x	x	x	
On-behalf-of flow	access token	x	x	x	
Client credentials			x (app-only)		

Tokens issued via the implicit mode have a length limitation due to being passed back to the browser via the URL (where `response_mode` is `query` or `fragment`). Some browsers have a limit on the size of the URL that can be put in the browser bar and fail when it is too long. Thus, these tokens do not have `groups` or `wids` claims.

Now that you have an overview of the basics, read on to understand the identity app model and API, how provisioning works in Microsoft identity platform, and links to detailed info about the common scenarios that Microsoft identity platform supports.

Application model

Microsoft identity platform represents applications following a specific model that's designed to fulfill two main functions:

- **Identify the app according to the authentication protocols it supports** - This involves enumerating all the identifiers, URLs, secrets, and related information that are needed at authentication time. Here, Microsoft identity platform:
 - Holds all the data required to support authentication at run time.
 - Holds all the data for deciding what resources an app might need to access and whether a given request should be fulfilled and under what circumstances.
 - Provides the infrastructure for implementing app provisioning within the app developer's tenant and to any other Azure AD tenant.
- **Handle user consent during token request time and facilitate the dynamic provisioning of apps across tenants** - Here, Microsoft identity platform:
 - Enables users and administrators to dynamically grant or deny consent for the app to access resources

on their behalf.

- Enables administrators to ultimately decide what apps are allowed to do and which users can use specific apps, and how the directory resources are accessed.

In Microsoft identity platform, an **application object** describes an application as an abstract entity. Developers work with applications. At deployment time, Microsoft identity platform uses a given application object as a blueprint to create a **service principal**, which represents a concrete instance of an application within a directory or tenant. It's the service principal that defines what the app can actually do in a specific target directory, who can use it, what resources it has access to, and so on. Microsoft identity platform creates a service principal from an application object through **consent**.

The following diagram shows a simplified Microsoft identity platform provisioning flow driven by consent. In it, two tenants exist (A and B), where tenant A owns the application, and tenant B is instantiating the application via a service principal.

In this provisioning flow:

1. A user from tenant B attempts to sign in with the app, the authorization endpoint requests a token for the application.
2. The user credentials are acquired and verified for authentication
3. The user is prompted to provide consent for the app to gain access to tenant B
4. Microsoft identity platform uses the application object in tenant A as a blueprint for creating a service principal in tenant B
5. The user receives the requested token

You can repeat this process as many times as you want for other tenants (C, D, and so on). Tenant A retains the blueprint for the app (application object). Users and admins of all the other tenants where the app is given consent retain control over what the application is allowed to do through the corresponding service principal object in each tenant. For more information, see [Application and service principal objects in Microsoft identity platform](#).

Claims in Microsoft identity platform security tokens

Security tokens (access and ID tokens) issued by Microsoft identity platform contain claims, or assertions of information about the subject that has been authenticated. Applications can use claims for various tasks, including:

- Validate the token
- Identify the subject's directory tenant
- Display user information
- Determine the subject's authorization

The claims present in any given security token are dependent upon the type of token, the type of credential used to authenticate the user, and the application configuration.

A brief description of each type of claim emitted by Microsoft identity platform is provided in the table below. For more detailed information, see the [access tokens](#) and [ID tokens](#) issued by Microsoft identity platform.

CLAIM	DESCRIPTION
Application ID	Identifies the application that is using the token.
Audience	Identifies the recipient resource the token is intended for.

CLAIM	DESCRIPTION
Application Authentication Context Class Reference	Indicates how the client was authenticated (public client vs. confidential client).
Authentication Instant	Records the date and time when the authentication occurred.
Authentication Method	Indicates how the subject of the token was authenticated (password, certificate, etc.).
First Name	Provides the given name of the user as set in Azure AD.
Groups	Contains object IDs of Azure AD groups that the user is a member of.
Identity Provider	Records the identity provider that authenticated the subject of the token.
Issued At	Records the time at which the token was issued, often used for token freshness.
Issuer	Identifies the STS that emitted the token as well as the Azure AD tenant.
Last Name	Provides the surname of the user as set in Azure AD.
Name	Provides a human readable value that identifies the subject of the token.
Object ID	Contains an immutable, unique identifier of the subject in Azure AD.
Roles	Contains friendly names of Azure AD Application Roles that the user has been granted.
Scope	Indicates the permissions granted to the client application.
Subject	Indicates the principal about which the token asserts information.
Tenant ID	Contains an immutable, unique identifier of the directory tenant that issued the token.
Token Lifetime	Defines the time interval within which a token is valid.
User Principal Name	Contains the user principal name of the subject.
Version	Contains the version number of the token.

Next steps

- Learn about the [application types and scenarios supported in Microsoft identity platform](#)

Authorize access to web applications using OpenID Connect and Azure Active Directory

9/5/2019 • 13 minutes to read • [Edit Online](#)

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.

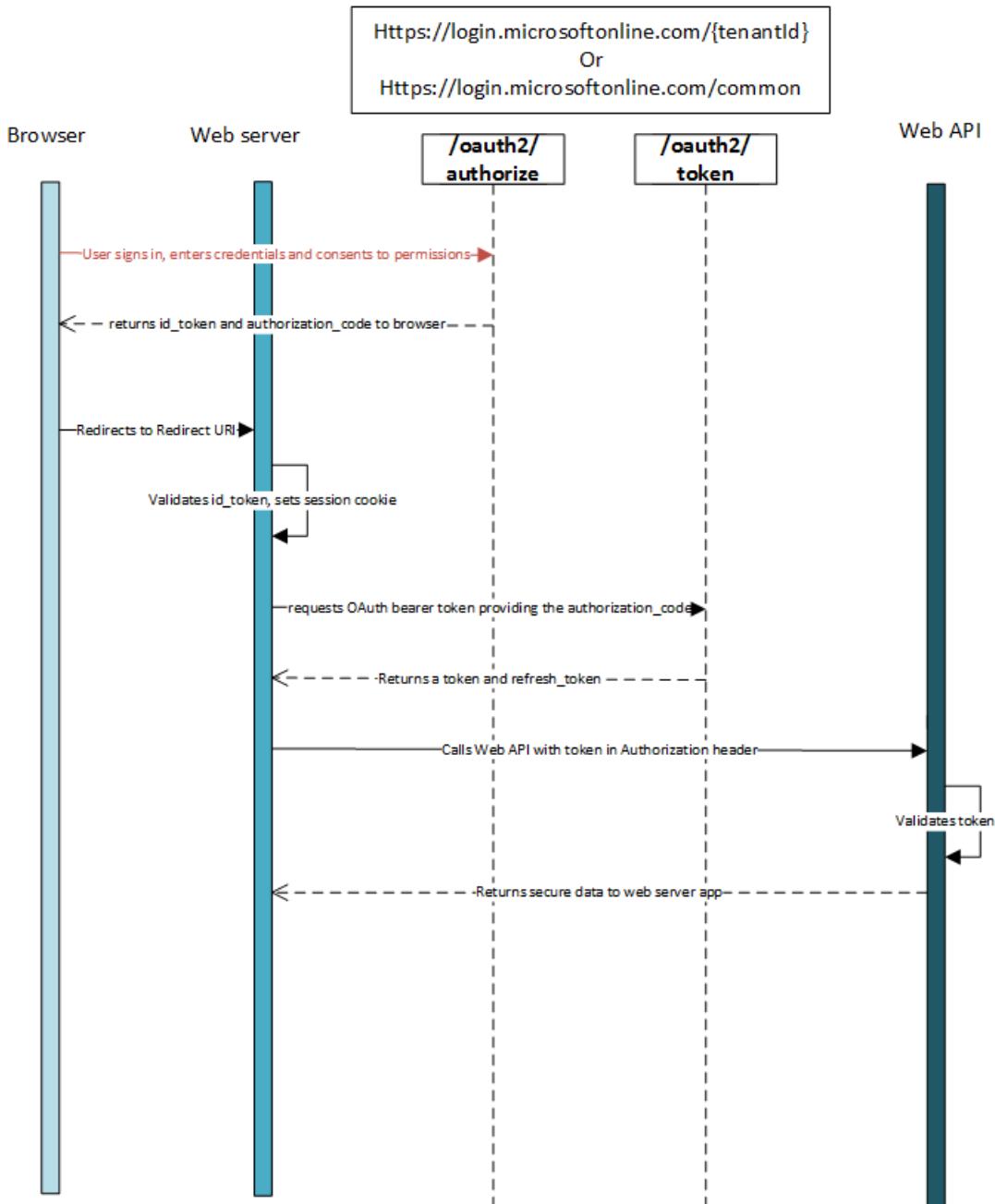
Register your application with your AD tenant

First, 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.

1. Sign in to the [Azure portal](#).
2. Choose your Azure AD tenant by selecting your account in the top right corner of the page, followed by selecting the **Switch Directory** navigation and then selecting the appropriate tenant.
 - Skip this step if you only have one Azure AD tenant under your account, or if you've already selected the appropriate Azure AD tenant.
3. In the Azure portal, search for and select **Azure Active Directory**.
4. In the **Azure Active Directory** left menu, select **App Registrations**, and then select **New registration**.
5. Follow the prompts and create a new application. It doesn't matter if it is a web application or a public client (mobile & desktop) application for this tutorial, but if you'd like specific examples for web applications or public client applications, check out our [quickstarts](#).
 - **Name** is the application name and describes your application to end users.
 - Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
 - Provide the **Redirect URI**. For web applications, this is the base URL of your app where users can sign in. For example, `http://localhost:12345`. For public client (mobile & desktop), Azure AD uses it to return token responses. Enter a value specific to your application. For example, `http://MyFirstAADApp`.
6. 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.
7. To find your application in the Azure portal, select **App registrations**, and then select **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](#). Note that providing a tenant ID rather than `common` in place of {tenant} above will result in tenant-specific URLs in the JSON object returned.

```
{
  "authorization_endpoint": "https://login.microsoftonline.com/{tenant}/oauth2/authorize",
  "token_endpoint": "https://login.microsoftonline.com/{tenant}/oauth2/token",
  "token_endpoint_auth_methods_supported":
  [
    "client_secret_post",
    "private_key_jwt",
    "client_secret_basic"
  ],
  "jwks_uri": "https://login.microsoftonline.com/common/discovery/keys"
  "userinfo_endpoint": "https://login.microsoftonline.com/{tenant}/openid/userinfo",
  ...
}
```

If your app has custom signing keys as a result of using the [claims-mapping](#) feature, you must append an `appid` query parameter containing the app ID in order to get a `jwks_uri` pointing to your app's signing key information. For example:

```
https://login.microsoftonline.com/{tenant}/.well-known/openid-configuration?appid=6731de76-14a6-49ae-97bc-6eba6914391e
```

contains a `jwks_uri` of

```
https://login.microsoftonline.com/{tenant}/discovery/keys?appid=6731de76-14a6-49ae-97bc-6eba6914391e .
```

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](#), 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%2Flocalhost%3a12345
&response_mode=form_post
&scope=openid
&state=12345
&nonce=7362CAEA-9CA5-4B43-9BA3-34D7C303EBA7
```

PARAMETER	DESCRIPTION
tenant	<p>required</p> <p>The <code>{tenant}</code> 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, <code>8eaef023-2b34-4da1-9baa-8bc8c9d6a490</code> or <code>contoso.onmicrosoft.com</code> or <code>common</code> for tenant-independent tokens</p>

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 <code>id_token</code> for OpenID Connect sign-in. It may also include other response_types, such as <code>code</code> or <code>token</code> .
scope	recommended	The OpenID Connect specification requires the scope <code>openid</code> , which translates to the "Sign you in" permission in the consent UI. This and other OIDC scopes are ignored on the v1.0 endpoint, but is still a best practice for standards-compliant clients.
nonce	required	A value included in the request, generated by the app, that is included in the resulting <code>id_token</code> 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. If missing, the user agent will be sent back to one of the redirect URLs registered for the app, at random. The maximum length is 255 bytes
response_mode	optional	Specifies the method that should be used to send the resulting authorization_code back to your app. Supported values are <code>form_post</code> for <i>HTTP form post</i> and <code>fragment</code> for <i>URL fragment</i> . For web applications, we recommend using <code>response_mode=form_post</code> to ensure the most secure transfer of tokens to your application. The default for any flow including an <code>id_token</code> is <code>fragment</code> .

PARAMETER		DESCRIPTION
state	recommended	A value included in the request that is returned in the token response. It can be a string of any content that you wish. A randomly generated unique value is typically used for preventing cross-site request forgery attacks . The state is also used to encode information about the user's state in the app before the authentication request occurred, such as the page or view they were on.
prompt	optional	Indicates the type of user interaction that is required. Currently, the only valid values are 'login', 'none', and 'consent'. <code>prompt=login</code> forces the user to enter their credentials on that request, negating single-sign on. <code>prompt=none</code> 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. <code>prompt=consent</code> 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 <code>preferred_username</code> claim.

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

Sample response

A sample response, sent to the `redirect_uri` specified in the sign-in request 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=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik1uQ19WVmNB...&state=12345
```

PARAMETER	DESCRIPTION
<code>id_token</code>	The <code>id_token</code> that the app requested. You can use the <code>id_token</code> to verify the user's identity and begin a session with the user.

PARAMETER	DESCRIPTION
state	A value included in the request that is also returned in the token response. A randomly generated unique value is typically used for preventing cross-site request forgery attacks . The state is also used to encode information about the user's state in the app before the authentication request occurred, such as the page or view they were on.

Error response

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

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

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

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

Error codes for authorization endpoint errors

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

ERROR CODE	DESCRIPTION	CLIENT ACTION
invalid_request	Protocol error, such as a missing required parameter.	Fix and resubmit the request. This is a development error, and is typically caught during initial testing.
unauthorized_client	The client application is not permitted to request an authorization code.	This usually occurs when the client application is not registered in Azure AD or is not added to the user's Azure AD tenant. The application can prompt the user with instruction for installing the application and adding it to Azure AD.
access_denied	Resource owner denied consent	The client application can notify the user that it cannot proceed unless the user consents.
unsupported_response_type	The authorization server does not support the response type in the request.	Fix and resubmit the request. This is a development error, and is typically caught during initial testing.

ERROR CODE	DESCRIPTION	CLIENT ACTION
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.

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 using the `wids` or `roles` claims.
- 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. For more information about `id_tokens` and claims, read [AAD id_tokens](#).

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%localhost%2Fmyapp%2F
```

PARAMETER	DESCRIPTION
-----------	-------------

PARAMETER		DESCRIPTION
post_logout_redirect_uri	recommended	The URL that the user should be redirected to after successful sign out. This URL must match one of the redirect URLs registered for your application in the app registration portal. If <code>post_logout_redirect_uri</code> is 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
that your application needs access to                      // The identifier of the protected resource (web API)
&state=12345                                              // Any value, provided by your app
&nonce=678910                                             // Any value, provided 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, sent to the `redirect_uri` using `response_mode=form_post`, looks like:

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

id_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik1uQ19WVmNB...&code=AwABAAAAvPM1KaPlrEqdFSBzjqfTGBcmLdgf
STLEMPGYuNHSUYBrq...&state=12345
```

PARAMETER	DESCRIPTION
<code>id_token</code>	The <code>id_token</code> that the app requested. You can use the <code>id_token</code> to verify the user's identity and begin a session with the user.
<code>code</code>	The <code>authorization_code</code> that the app requested. The app can use the authorization code to request an access token for the target resource. <code>Authorization_codes</code> are short lived, and typically expire after about 10 minutes.
<code>state</code>	If a <code>state</code> parameter is included in the request, the same value should appear in the response. The app should verify that the <code>state</code> 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
<code>error</code>	An error code string that can be used to classify types of errors that occur, and can be used to react to errors.
<code>error_description</code>	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](#).

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. To get access tokens, you can follow the steps described in the "Use the authorization code to request an access token" section of our [OAuth code flow documentation](#).

Next steps

- Learn more about the [access tokens](#).
- Learn more about the `id_token` and [claims](#).

Understanding the OAuth2 implicit grant flow in Azure Active Directory (AD)

8/15/2019 • 7 minutes to read • [Edit Online](#)

Applies to:

- Azure AD v1.0 endpoint

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. In OAuth2 implicit grant, the authorization endpoint issues access tokens directly to the client using a redirect URI that was previously supplied. 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 returned 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, and the areas you need to pay attention to are well documented (for example, [Misuse of Access Token to Impersonate Resource Owner in Implicit Flow](#) and [OAuth 2.0 Threat Model and Security Considerations](#)). 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.

Next steps

- For a complete list of developer resources, including reference information for the protocols and OAuth2 authorization grant flows support by Azure AD, refer to the [Azure AD Developer's Guide](#)
- See [How to integrate an application with Azure AD](#) for additional depth on the application integration process.

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

10/30/2019 • 21 minutes to read • [Edit Online](#)

NOTE

If you don't tell the server what resource you plan to call, then the server will not trigger the Conditional Access policies for that resource. So in order to have MFA trigger, you will need to include a resource in your URL.

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](#). It is used to perform authentication and authorization in most application types, including web apps and natively installed apps.

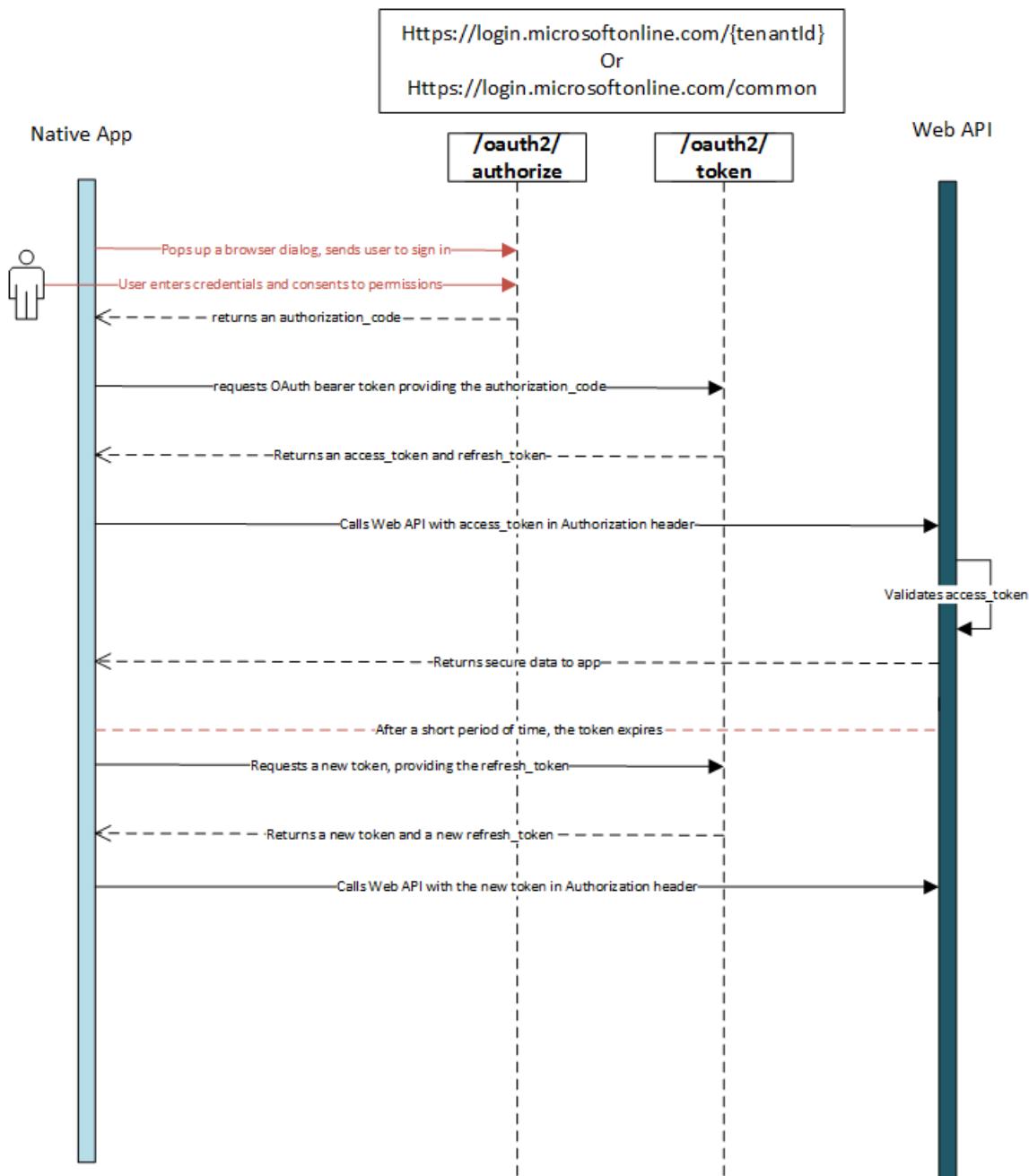
Register your application with your AD tenant

First, 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.

1. Sign in to the [Azure portal](#).
2. Choose your Azure AD tenant by selecting your account in the top right corner of the page, followed by selecting the **Switch Directory** navigation and then selecting the appropriate tenant.
 - Skip this step if you only have one Azure AD tenant under your account, or if you've already selected the appropriate Azure AD tenant.
3. In the Azure portal, search for and select **Azure Active Directory**.
4. In the **Azure Active Directory** left menu, select **App Registrations**, and then select **New registration**.
5. Follow the prompts and create a new application. It doesn't matter if it is a web application or a public client (mobile & desktop) application for this tutorial, but if you'd like specific examples for web applications or public client applications, check out our [quickstarts](#).
 - **Name** is the application name and describes your application to end users.
 - Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
 - Provide the **Redirect URI**. For web applications, this is the base URL of your app where users can sign in. For example, `http://localhost:12345`. For public client (mobile & desktop), Azure AD uses it to return token responses. Enter a value specific to your application. For example, `http://MyFirstAADApp`.
6. 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.
7. To find your application in the Azure portal, select **App registrations**, and then select **View all applications**.

OAuth 2.0 authorization flow

At a high level, the entire authorization flow for an application looks a bit like this:



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%2Flocalhost%3A12345
&response_mode=query
&resource=https%3A%2F%2Fservice.contoso.com%2F
&state=12345
```

PARAMETER		DESCRIPTION
tenant	required	<p>The <code>{tenant}</code> 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,</p> <div style="border: 1px solid #ccc; padding: 2px;"><code>8eaef023-2b34-4da1-9baa-8bc8c9d6a490</code></div> <p>or <code>contoso.onmicrosoft.com</code> or <code>common</code> for tenant-independent tokens</p>
client_id	required	<p>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.</p>
response_type	required	<p>Must include <code>code</code> for the authorization code flow.</p>
redirect_uri	recommended	<p>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</p> <div style="border: 1px solid #ccc; padding: 2px;"><code>urn:ietf:wg:oauth:2.0:oob</code></div>
response_mode	optional	<p>Specifies the method that should be used to send the resulting token back to your app. Can be <code>query</code>, <code>fragment</code>, or <code>form_post</code>. <code>query</code> 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 <code>query</code> as specified in the OpenID spec. If you're requesting just the code, you can use <code>query</code>, <code>fragment</code>, or <code>form_post</code>. <code>form_post</code> executes a POST containing the code to your redirect URI. The default is <code>query</code> for a code flow.</p>
state	recommended	<p>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.</p>

PARAMETER		DESCRIPTION
resource	recommended	<p>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.</p>
scope	ignored	<p>For v1 Azure AD apps, scopes must be statically configured in the Azure Portal under the applications Settings, Required Permissions.</p>
prompt	optional	<p>Indicate the type of user interaction that is required. Valid values are:</p> <ul style="list-style-type: none"> <i>login</i>: The user should be prompted to reauthenticate. <i>select_account</i>: 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. <i>consent</i>: User consent has been granted, but needs to be updated. The user should be prompted to consent. <i>admin_consent</i>: An administrator should be prompted to consent on behalf of all users in their organization
login_hint	optional	<p>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 <code>preferred_username</code> claim.</p>

PARAMETER		DESCRIPTION
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.
code_challenge_method	recommended	The method used to encode the <code>code_verifier</code> for the <code>code_challenge</code> parameter. Can be one of <code>plain</code> or <code>s256</code> . If excluded, <code>code_challenge</code> is assumed to be plaintext if <code>code_challenge</code> is included. Azure AAD v1.0 supports both <code>plain</code> and <code>s256</code> . For more information, see the PKCE RFC .
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>code_challenge_method</code> is included. For more information, see the PKCE RFC .

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=
AwABAAAAvPM1KaPlrEqdFSBzjqfTGBCmLdgfSTLEMPGYuNHSUYBrqqf_ZT_p5uEAEJJ_nZ3UmphWygRNy2C3jJ239gV_DBnZ2syeg95Ki-
374WHUP-i3yIhv5i-7KU2CEoPXwURQp6IVYMw-
DjAOzn7C3JCu5wpngXmbZKtJdWmiBzHpcO2aICJPu1KvJrDLDP20chJBxzVYJtkfjviLNNW717Y3ydcHDsBRKZc3GuMQanmcghXPyoDg41g8X
bwPudVh7uCmUponBQpIhbuffFP_tbV8SNzsPoFz9CLpBCZagJVXeqWoYMPe2dSsPiL09Alf_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.
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.

ERROR CODE	DESCRIPTION	CLIENT ACTION
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_DBnZ2syeg95
Ki-374WHUP-i3yIhv5i-7KU2CEoPXwURQp6IVVMw-
DjAOzn7C3JCu5wpngXmbZKtJdwmiBzHpcO2aICJPu1KvJrDLDP20chJBxzVYJtkfjviLNNW717Y3ydcHDsBRKZc3GuMQanmcghXPyoDg41g8X
bwPudVh7uCmUponBQpIhbuffFP_tbV8SNzsPoFz9CLpBCZagJVXeqWoYMPe2dSsPiL09Alf_YIe5zpi-
zY4C3aLw5g9at35eZTfNd0gBRpR5ojkMICZ6IgAA
&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	<p>required</p> <p>The <code>{tenant}</code> 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,</p> <div style="background-color: #f0f0f0; padding: 5px; display: inline-block;"> 8eaef023-2b34-4da1-9baa- 8bc8c9d6a490 </div> <p>or <code>contoso.onmicrosoft.com</code> or <code>common</code> for tenant-independent tokens</p>

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. The Application Id is displayed in the settings of the app registration.
grant_type	required	Must be <code>authorization_code</code> for the authorization code flow.
code	required	The <code>authorization_code</code> that you acquired in the previous section
redirect_uri	required	A <code>redirect_uri</code> registered on the client application.
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 <code>client_secret</code> 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 <code>resource`</code> parameters must match.
code_verifier	optional	The same <code>code_verifier</code> that was used to obtain the <code>authorization_code</code> . Required if PKCE was used in the authorization code grant request. For more information, see the PKCE RFC

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":  
    "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZnl0aEV1THdqchBSk9NOW4tQSJ9.eyJhdWQiOiJodHRwczovL3N  
    lcnZpY2UuY29udG9zby5jb20vIiwiAxNzIjoiaHR0cHM6Ly9zdHMud2luZG93cy5uZXQvN2ZlODE0NDctZGE1Ny00Mzg1LWJ1Y2ItNmR1NTdm  
    MjE0Nzd1LyIsImlhCI6MTM40DQ0MDg2MywibmJmIjoxMzg4NDQwODYzLCJ1eHAiOjEzODg0NDQ3NjMsInZlcii6IjEuMCIsInRpZCI6IjdmZ  
    TgxNDQ3LWRhNTctNDM4NS1iZWNiLTzkZTU3ZjIxNDc3ZSIsm9pZC16IjY4Mzg5YWUyLTYYzmEtNGIxOC05MWZ1LTUzZGQxMD1kNzRmNSIsIn  
    Vwbii6ImZyYW5rbUBjb250b3NvLmNbSIsInVuaXF1ZV9uYW11IjoiZnJhbmttQGNvbvnRvc28uY29tIiwiic3ViIjoiZGVocU1qQU1PRT1QV0p  
    XYkhzZnRyDjFYWJQVmwwQ2o4UUftZWSTFY50CIsImZhbwlsseV9uYW11IjoiTwlsbGVyIiwiZ21Zw5fbmFtZSI6IkZyYW5rIiwiYXBwaWQi  
    OiiyZDRkMTFhMi1mODE0LTQ2YTctODkwYS0yNzRhNzJhNzMwOWUiLCJhcHBpZGFjciI6IjAiLCJzY3Ai0iJ1c2VyX2ltcGVyc29uYXRpb24iL  
    CJhY3Ii0iIxIn0.JZw8jC0gptZxVC-  
    715sFkdnJgP3_tRjeQEPgUn28XctVe3QqmheLzw7QVZDPCyGycDWBaqy7FLpSekET_BftDkewRhyHk9FW_KeEZ0ch2c3i08NGNDb6XYGVayN  
    uSeSYk5Aw_p3ICR1UV1bqEwk-Jkzs9EEkQg4hbefqJS6yS1HoV_2EsEhpdu_wCQpxK89WP3hLYZETRJtG5kvCCEOvSHxmDE6eTHGTnEgsIk--  
    U1Pe275Dvou4gEAwLofhLDQbMSjn1V5VLsjimNBVcSRFShoxmQwBJR_b2011Y5IuD6St5zPnzruBbZYkGNurQK63TJPWmRd3mbJsGM0mf3CUQ  
  ",  
  "token_type": "Bearer",  
  "expires_in": "3600",  
  "expires_on": "1388444763",  
  "resource": "https://service.contoso.com/",  
  "refresh_token": "AwBAAAAAvPM1KaPlrEqdFSBzjqfTGAmxZGUTdM0t4B4rTfgV29ghDOHrc2B-  
  C_hHeJaJICqjZ3mY2b_YNqmf9SoAyld1PycGCB90xZzeEDg6oBz0IPfYsbDNf621pKo2Q3GGTHY1mNfwoc-  
  01rxK69hkha2CF12azM_NYhg0668yfcU14VBbiSHZyd1NVZG5QTIOcb0bu3qnLutbpadZGAxqjIbMkQ2bQS09fTrjMBtDE3D6kSMIodpCecoA  
  Non9b0LATkpitimVCrl-Nyfn3oyG4ZCWu18M9-vEou4Sq-1oMDzExgAf61noxzKniaTecM-Ve5cq6wHqYQjfv9D0z4lbceuYCAA",  
  "scope": "https%3A%2F%2Fgraph.microsoft.com%2Fmail.read",  
  "id_token":  
    "eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhdWQiOiIyZDRkMTFhMi1mODE0LTQ2YTctODkwYS0yNzRhNzJhNzMwOWUiLCJpc3Mi0iJo  
    dHRwczovL3N0cy53aW5kb3dzLm51dC83zmU4MTQ0Ny1kYTU3LTQzODUtYmVjYi02ZGU1N2YyMTQ3N2UvIiwiWF0IjoxMzg4NDQwODYzLCJuY  
    mYiojEzODg0NDA4NjMsImV4cCI6MTM40DQ0NDc2MywidmVjIjoiMS4wIiwidG1kIjoiN2ZlODE0NDctZGE1Ny00Mzg1LWJ1Y2ItNmR1NTdmMj  
    E0Nzd1Iiwb2lkIjoiNjgzODlhZTItNjJmYS00YjE4LTkxZmUtNTNkZDEw0WQ3NGY1IiwidXBuIjoiZnJhbmttQGNvbvnRvc28uY29tIiwidW5  
    pcXV1X25hbWUi0iJmcMfa21AY29udG9zby5jb20iLCJzdWIi0iJKV3ZZZENXUGhobHTMVpzJd5WVV4U2hvD3RVbTV5elBtd18talgzkhZ  
    IiwiZmFtaWx5X25hbWUi0iJnaWxsZXIiLCJnaXZ1b19uYW11IjoiRnJhbmsifQ."  
}
```

PARAMETER	DESCRIPTION
access_token	The requested access token. This is an opaque string - it depends on what the resource expects to receive, and is not intended for the client to look at. 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. For more information about Bearer tokens, see OAuth2.0 Authorization Framework: Bearer Token Usage (RFC 6750)
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.

PARAMETER	DESCRIPTION
resource	The App ID URI of the web API (secured resource).
scope	Impersonation permissions granted to the client application. The default permission is <code>user_impersonation</code> . 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.

For more information about JSON web tokens, see the [JWT IETF draft specification](#). To learn more about `id_tokens`, see the [v1.0 OpenID Connect flow](#).

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.

PARAMETER	DESCRIPTION
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.

HTTP status codes

The following table lists the HTTP status codes that the token issuance endpoint returns. In some cases, the error code is sufficient to describe the response, but if there are errors, you need to parse the accompanying JSON document and examine its error code.

HTTP CODE	DESCRIPTION
400	Default HTTP code. Used in most cases and is typically due to a malformed request. Fix and resubmit the request.
401	Authentication failed. For example, the request is missing the client_secret parameter.
403	Authorization failed. For example, the user does not have permission to access the resource.
500	An internal error has occurred at the service. Retry the request.

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. The [RFC 6750](#) specification explains how to use bearer tokens in HTTP requests to access protected resources.

Sample request

```
GET /data HTTP/1.1
Host: service.contoso.com
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2EStZn10aEV1THdqCdBsk9N0W4tQSJ9.eyJhdWQiOiJodHRwczovL3N1cnZpY2UuY29udG9zby5jb20vIiwiaXNzIjoiaHR0cHM6Ly9zdHMud2luZG93cy5uZXQvN2Z1ODE0NDctZGE1Ny00Mzg1LWJ1Y2ItNmR1NTdmMjE0Nzd1LyIsImIhdCI6MTM4ODQ0MDg2MywibmJmIjoxMzg4NDQwODYzLCJ1eHAiOjEzODg0NDQ3NjMsInZlciI6IjEuMCIsInRpZCI6IjdmZTgxNDQ3LWRhNTctNDM4NS1iZWNiLTZkZTU3ZjIxNDc3ZSIIsIm9pZCI6IjY4Mzg5YWUyLTYyZmEtNGIxOC05MWZ1LTUzZGQxMD1kNzRmNSIsInVwbii6ImZyYW5rbUBjb250b3NvLmNvbSIIsInVuaxF1Zv9uYw1lIjoiZnJhbmttQGNvbnRvc28uY29tIiwiic3ViIjoiZGV0cUlqOUlPRT1QV0pxYkhzZnRYdDJFYWJQVmwwQ2o4UUFTZWZTFY5OCIsImZhbwlseV9uYw1lIjoiTWlsbGVyIiwiZ2l2ZW5fbmFtZSI6IkZyYW5rIiwiYXBwaWQioiIyZDRkMTFhMi1mODE0LTQ2YTctODkwYS0yNzRhNzJhNzMwOUUiLCJhcHBpZGFjciI6IjAiLCJzY3AoiJ1c2Vyx2ltcGVyc29uYXRpb24iLCJhY3IiOiIxIn0.JZw8jC0gptZxVC-715sFkdnJgP3_tRjeQEPgUn28XctVe3QqmheLZw7QVZDPCyGycDWBaqy7FLpSekET_BftDkewRhyHk9FW_KeEz0ch2c3i08NGNDb6XYGVayNuSeSYk5Aw_p3ICR1UV1bqEwk-Jkzs9EEkQg4hbefqJS6yS1HoV_2EsEhp_wCQpxK89WP3hLYZETRJtG5kvCCEOvSHxmDE6eTHGTnEgsIk--U1Pe275Dvou4gEAwLofhLDQbMSjn1V5VLsjsimNBVcSRFSshoxmQwBJR_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_description="The access token is missing.",
```

Error parameters

Parameter	Description
-----------	-------------

PARAMETER	DESCRIPTION
authorization_uri	<p>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.</p> <p>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.</p>
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	<p>Returns the unique identifier of the resource. The client application can use this identifier as the value of the <code>resource</code> parameter when it requests a token for the resource.</p> <p>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</p> <p>The recommended strategy for preventing an attack is to verify that the <code>resource_id</code> matches the base of the web API URL that being accessed. For example, if https://service.contoso.com/data is being accessed, the <code>resource_id</code> can be https://service.contoso.com/. The client application must reject a <code>resource_id</code> that does not begin with the base URL unless there is a reliable alternate way to verify the id.</p>

Bearer scheme error codes

The RFC 6750 specification defines the following errors for resources that use the WWW-Authenticate header and Bearer scheme in the response.

HTTP STATUS CODE	ERROR CODE	DESCRIPTION	CLIENT ACTION
400	invalid_request	The request is not well-formed. For example, it might be missing a parameter or using the same parameter twice.	Fix the error and retry the request. This type of error should occur only during development and be detected in initial testing.
401	invalid_token	The access token is missing, invalid, or is revoked. The value of the <code>error_description</code> parameter provides additional detail.	Request a new token from the authorization server. If the new token fails, an unexpected error has occurred. Send an error message to the user and retry after random delays.

HTTP STATUS CODE	ERROR CODE	DESCRIPTION	CLIENT ACTION
403	insufficient_scope	The access token does not contain the impersonation permissions required to access the resource.	Send a new authorization request to the authorization endpoint. If the response contains the scope parameter, use the scope value in the request to the resource.
403	insufficient_access	The subject of the token does not have the permissions that are required to access the resource.	Prompt the user to use a different account or to request permissions to the specified resource.

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=0AAABAAAiL9Kn2Z27UubvWFPbm0gLWQJVzCTE9UkP3pSx1aXxUjq...
&grant_type=refresh_token
&resource=https%3A%2F%2Fservice.contoso.com%2F
&client_secret=JqQX2PN09bpM0uEihUPzyrh    // 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": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIngldCI6Ik5HVEZ2ZEstZn10aEV1THdqcHdBSk9N0W4tQSJ9.eyJhdWQiOiJodHRwczovL3NlcnPZUuY29udG9zby5jb20vIiwiaXNzIjoiaHR0CHM6Ly9zdHMud2luZG93cy5uZXQvN2Z1ODE0NDctZGE1Ny00Mzg1LWJ1Y2ItNmR1NTdmMjE0Nzd1LyIsImlhCI6MTM40DQ0MDg2MywibmJmIjoxMzg4NDQwODYzLCJleHAIojezODg0NDQ3NjMsInZlcii6IjEuMCIsInRpZCI6IjdmZTgxNDQ3LWRhNTctNDM4NS1iZWNiLTzkZTU3ZjIxNDc3ZSIisIm9pZCI6IjY4Mzg5YWUyLTYYzMeTNGIxOC05MWZ1LTUzzGQxMD1kNzRmNSIsInVwbii6ImZyYW5rbUBjb250b3NvLmNvbSISInVuaXF1ZV9uYW11IjoiZnJhbmttQGNvbnRvc28uY29tIiwick3ViIjoiZGVocU1qOU1PRT1QV0pXYkhZnRYdDJFYWJQVmwwQ2o4UUFTzWZSTFY50CIsImZhbw1se9uYW11IjoiTwlsbGVyIiwiZ21Zw5fbmftZSI6IkZyYW5rIiwiYXBwaWQiOiiyZDRkMTFhMi1mODE0LTQ2YTctODkwYS0yNzRhNzJhNzMwOWUiLCJhcHBpZGFjci6IjAiLCJzY3Ai0iJ1c2Vyx2ltcGVyc29uYXRpb24iLCJhY3Ii0iIxIn0.JZw8jC0gptZxVC-715sFkdnJgP3_tRjeQEPgUn28XctVe3QqmheLZw7QVZDPCyGycDWBaqy7FLpSekET_BftDkewRhyHk9FW_KeEz0ch2c3i08NGNDb6XYGVayNuSeSYk5Aw_p3ICR1UV1bqEwk-Jkzs9EEkQg4hbefqJS6yS1HoV_2EsEhp_wCQpxK89WPs3hLYZETRJtG5kvCCEOvSHxmDE6eTHGTnEgsIk--UlPe275Dvou4gEAwLofhLDQbMSjn1V5VLsJimNBVcSRFSshoxmQwBJR_b2011Y5iuD6St5zPnzruBbZYkGNurQK63TJPWmRd3mbJsGM0mf3CUQ",
  "refresh_token": "AwABAAAAAv YNqmf9SoAylD1PycGCB90xzZeEDg6oBzOIPfYsbDWnf621pKo2Q3GGTHY1mNfwoc-01rxK69hkha2CF12azM_NYhg0668yfcU14VBbiSHZyd1NVZG5QTI0cb0bu3qnLutbpadZGAxqjIbMkQ2bQS09fTrjMBtDE3D6kSMIodpCeoANon9b0LATkpitimVCrl PM1KaPlrEqdFSBzjqfTGAMxZGUTdM0t4B4rTfgV29ghDOHrc2B-C_hHeJaJICqjZ3mY2b_YNqmf9SoAylD1PycGCB90xzZeEDg6oBzOIPfYsbDWnf621pKo2Q3GGTHY1mNfwoc-01rxK69hkha2CF12azM_NYhg0668yfmVCrl-NyfN3oyG4ZCwu18M9-vEou4Sq-1oMDzExgAf61noxzkNiaTecM-Ve5cq6wHqYQjfV9D0z4lbceuYCAA"
}
```

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](#).

Next steps

To learn more about the Azure AD v1.0 endpoint and how to add authentication and authorization to your web applications and web APIs, see [sample applications](#).

Service-to-service calls that use delegated user identity in the On-Behalf-Of flow

8/21/2019 • 12 minutes to read • [Edit Online](#)

Applies to:

- Azure AD v1.0 endpoint

The OAuth 2.0 On-Behalf-Of (OBO) flow enables an application that invokes a service or web API to pass user authentication to another service or web API. The OBO flow propagates the delegated user identity and permissions through the request chain. For the middle-tier service to make authenticated requests to the downstream service, it must secure an access token from Azure Active Directory (Azure AD) on behalf of the user.

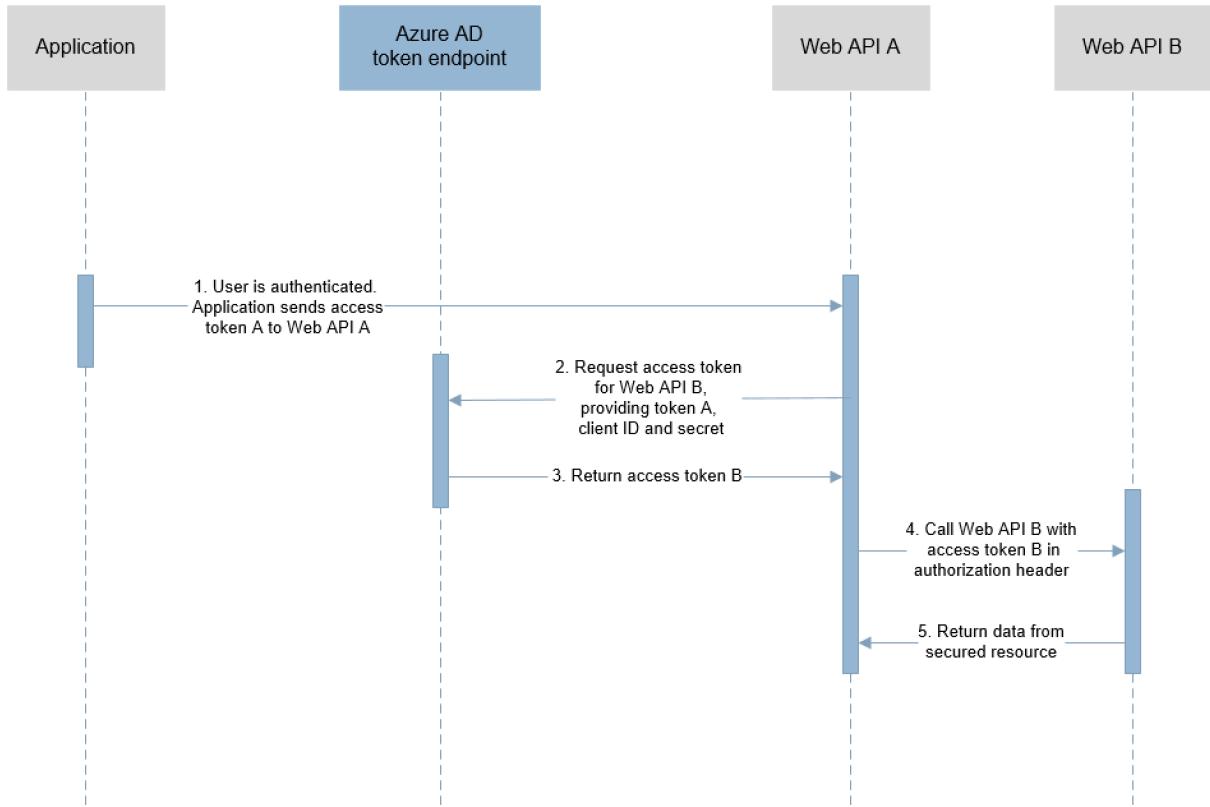
IMPORTANT

As of May 2018, an `id_token` can't be used for the On-Behalf-Of flow. Single-page apps (SPAs) must pass an access token to a middle-tier confidential client to perform OBO flows. For more detail about the clients that can perform On-Behalf-Of calls, see [limitations](#).

On-Behalf-Of flow diagram

The OBO flow starts after the user has been authenticated on an application that uses the [OAuth 2.0 authorization code grant flow](#). At that point, the application sends an access token (token A) to the middle-tier web API (API A) containing the user's claims and consent to access API A. Next, API A makes an authenticated request to the downstream web API (API B).

These steps constitute the On-Behalf-Of flow:



1. The client application makes a request to API A with the token A.
2. API A authenticates to the Azure AD token issuance endpoint and requests a token to access API B.
3. The Azure AD token issuance endpoint validates API A's credentials with token A and issues the access token for API B (token B).
4. The request to API B contains token B in the authorization header.
5. API B returns data from the secured resource.

NOTE

The audience claim in an access token used to request a token for a downstream service must be the ID of the service making the OBO request. The token also must be signed with the Azure Active Directory global signing key (which is the default for applications registered via **App registrations** in the portal).

Register the application and service in Azure AD

Register both the middle-tier service and the client application in Azure AD.

Register the middle-tier service

1. Sign in to the [Azure portal](#).
2. On the top bar, select your account and look under the **Directory** list to select an Active Directory tenant for your application.
3. Select **More Services** on the left pane and choose **Azure Active Directory**.
4. Select **App registrations** and then **New registration**.
5. Enter a friendly name for the application and select the application type.
6. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
7. Set the redirect URI to the base URL.
8. Select **Register** to create the application.

9. Generate a client secret before exiting the Azure portal.
10. In the Azure portal, choose your application and select **Certificates & secrets**.
11. Select **New client secret** and add a secret with a duration of either one year or two years.
12. When you save this page, the Azure portal displays the secret value. Copy and save the secret value in a safe location.

IMPORTANT

You need the secret to configure the application settings in your implementation. This secret value is not displayed again, and it isn't retrievable by any other means. Record it as soon as it is visible in the Azure portal.

Register the client application

1. Sign in to the [Azure portal](#).
2. On the top bar, select your account and look under the **Directory** list to select an Active Directory tenant for your application.
3. Select **More Services** on the left pane and choose **Azure Active Directory**.
4. Select **App registrations** and then **New registration**.
5. Enter a friendly name for the application and select the application type.
6. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
7. Set the redirect URI to the base URL.
8. Select **Register** to create the application.
9. Configure permissions for your application. In **API permissions**, select **Add a permission** and then **My APIs**.
10. Type the name of the middle-tier service in the text field.
11. Choose **Select Permissions** and then select **Access <service name>**.

Configure known client applications

In this scenario, the middle-tier service needs to obtain the user's consent to access the downstream API without a user interaction. The option to grant access to the downstream API must be presented up front as part of the consent step during authentication.

Follow the steps below to explicitly bind the client app's registration in Azure AD with the middle-tier service's registration. This operation merges the consent required by both the client and middle-tier into a single dialog.

1. Go to the middle-tier service registration and select **Manifest** to open the manifest editor.
2. Locate the `knownClientApplications` array property and add the client ID of the client application as an element.
3. Save the manifest by selecting **Save**.

Service-to-service access token request

To request an access token, make an HTTP POST to the tenant-specific Azure AD endpoint with the following parameters:

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

The client application is secured either by a shared secret or by 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	The type of the token request. An OBO request uses a JSON Web Token (JWT) so the value must be urn:ietf:params:oauth:grant-type:jwt-bearer .
assertion	required	The value of the access token used in the request.
client_id	required	The app ID assigned to the calling service during registration with Azure AD. To find the app ID in the Azure portal, select Active Directory , choose the directory, and then select the application name.
client_secret	required	The key registered for the calling service in Azure AD. This value should have been noted at the time of registration.
resource	required	The app ID URI of the receiving service (secured resource). To find the app ID URI in the Azure portal, select Active Directory and choose the directory. Select the application name, choose All settings , and then select Properties .
requested_token_use	required	Specifies how the request should be processed. In the On-Behalf-Of flow, the value must be on_behalf_of .
scope	required	A space separated list of scopes for the token request. For OpenID Connect, the scope openid must be specified.

Example

The following HTTP POST requests an access token for the <https://graph.windows.net> web API. The `client_id` identifies the service that requests the access token.

```
// line breaks for legibility only

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Ajwt-bearer
&client_id=625391af-c675-43e5-8e44-edd3e30ceb15
&client_secret=0Y1W%2BY3yYb3d9N8v5jvm8WrGzVZaAaHbHHcGbcgG%2BoI%3D
&resource=https%3A%2F%2Fgraph.windows.net
&assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6InowMzl6ZHNGdWl6cEJmQ1ZLMVRuMjVRSF1PMCIsImtpZCI6InowMzl6ZHNGdWl6cEJmQ1ZLMVRuMjVRSF1PMCJ9.eyJhdWQiOiJodHRwczovL2Rkb2JhbGlhbm91dGxvb2sub25taWNyb3NvZnQuY29tLzE5MjNmODYyLWU2ZGMtNDfhMy04MWRhLTgwMmJhZTAwYWY2ZCIsImlzcyI6Imh0dHBzOi8vc3RzLndpbmRvd3MubmV0LzI2MDM5Y2N1LTQ4OWQtNDAwMi04MjkzLTViMGM1MTM0ZWfjYi8iLCJpYXQiOjE0OTM0MjMxNTIsIm5iZiI6MTQ5MzQyMzE1MiwiZXhwIjoxNDkzNDY2NjUyLCJhY3Ii0iIxIiwiYwlvIjoiWTJaZ11CRFF2aT1VZEc0LzM0L3dpQndqbjhYeVp4YmR1TFhmVE1qeG8yY1N2elgreHBVQSIImFtcI6WyJwd2QiXSwiYXBwaWQiOjImZE1MDA3OS03YmViLTQxN2YtYTA2YS0zZmRjNzhjMzI1NDUiLCJhcHBpZGFjciI6IjAiLCJ1X2V4cCI6MzAyNDAwLCJmYW1pbH1fbmFtZSI6IlR1c3QiLCJnaXZ1b19uYW1lIjoiTmF2eWEiLCJpcGFkZHIIoIiXNjcuMjIwLjEuMTc3IiwiwmFtZSI6Ik5hdnlhIFRlc3QiLCJvaWQioIiXy2Q0YmNhYy1iODA4LTQyM2EtOWUyZi04MjdmYmIxYmI3MzkiLCJwbGF0ZiI6IjMiLCJzY3Ai0iJ1c2Vyx2ltcGVyc29uYXRpb24iLCJzdWIiOjEVXpYbkdkMDJIK0zRW5pbDFxdjZCakxTNU1lQy0tQ2ZpbzRxS1MzNEc4IiwidGlkIjoiMjYwMz1jY2UtNDg5ZC00MDAyLTgyOTMtNWlwyZuMzR1YWNiIiwiidW5pcXV1X25hbWUi0iJuYXZ5YUBkZG9iYWxpYW5vdXRsb29rLm9ubWljcm9zb2Z0LmNvbSISInVwbiI6Im5hdnlhQGRkb2JhbGlhbm91dGxvb2sub25taWNyb3NvZnQuY29tIiwidmVyIjoiMS4wIn0.R-Ke-X071K0r5uLwxB8g5CrcPAwRln5SccJCfEjU6IuqpqcjwCdeDnOySiVPDU_ZU5knJmzRCF8fcjFtPsaA4R7vdIEbDuOur15FXSvE8FvVSjp_490H6hBYqoSAs1n3FMfb06Z8yfCIY4tS0B2I6ahQ_x4ZWFwg1C3w5mK-_4iX81bqj95eV4RUKeFuHhQDXtWhrSgIE0YiluMvA4TnaJdLq_twXIc4_Tq_KfpkvI0040NKgU7EAMER1wZ4aDcJV2yf22gQ1sCSig6EGSTm mzDuEPsYiyd4NhidRZJP4HiQh-hePBQsgcSgYGvz9wC6n57ufYKh2wm_Ti3Q&requested_token_use=on_behalf_of&scope=openid
```

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	The type of the token request. An OBO request uses a JWT access token so the value must be urn:ietf:params:oauth:grant-type:jwt-bearer .
assertion	required	The value of the token used in the request.
client_id	required	The app ID assigned to the calling service during registration with Azure AD. To find the app ID in the Azure portal, select Active Directory , choose the directory, and then select the application name.
client_assertion_type	required	The value must be urn:ietf:params:oauth:client-assertion-type:jwt-bearer
client_assertion	required	A JSON Web Token that you create and sign with the certificate you registered as credentials for your application. See certificate credentials to learn about assertion format and about how to register your certificate.

PARAMETER		DESCRIPTION
resource	required	The app ID URI of the receiving service (secured resource). To find the app ID URI in the Azure portal, select Active Directory and choose the directory. Select the application name, choose All settings , and then select Properties .
requested_token_use	required	Specifies how the request should be processed. In the On-Behalf-Of flow, the value must be on_behalf_of .
scope	required	A space separated list of scopes for the token request. For OpenID Connect, the scope openid must be specified.

These parameters are almost the same as with the request by shared secret except that the

`client_secret` parameter is replaced by two parameters: `client_assertion_type` and `client_assertion`.

Example

The following HTTP POST requests an access token for the <https://graph.windows.net> web API with a certificate.

The `client_id` identifies the service that requests the access token.

```
// line breaks for legibility only

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Ajwt-bearer
&client_id=625391af-c675-43e5-8e44-edd3e30ceb15
&client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJSUzI1NiIsIng1dCI6Imd40HRHeXN5amNScUtqRlBuZDdSRnd2d1pJMCJ9.eyJ{a lot of characters here}M8U3bSUKKJD Eg
&resource=https%3A%2F%2Fgraph.windows.net
&assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6InowMz16ZHNGdW16cEJmQ1ZLMVRuMjVRSF1PMCI sImtpZCI6InowMz16ZHNGdW16cEJmQ1ZLMVRuMjVRSF1PMCI J9.eyJhdWQiOiJodHRwczovL2Rkb2JhbGlhbm91dGxvb2sub25taWnb3NvZnQuY29tLzE5MjNmODYyLWU2ZGMtNDFhMy04MWRhLTgwMmJhZTAwYWY2ZCIsImIzcyI6Imh0dBzOi8vc3RzLndpbmRvd3MubmV0LzI2MDM5Y2N1LTQ40WQtNDawMi04MjkzLTViMGM1MTM0ZWfjYi8iLCJpYXQiOjE00TM0MjMxNTIsIm5iZiI6MTQ5MzQyMzE1MiwiZXhwIjoxNDkzNDY2NjUyLCJhY3Ii0iIxIiwiYwlvIjoiWTJaZ1lCRFF2aT1VZEc0LzM0L3dpQndqbjhYeVp4YmR1TFhmVE1QeG8yY1N2e1greHBVQSI sImFtcI6WjJwd2QiXSwiYXBwaWQoIiJiMzE1MDA3OS03YmViLTQxN2YtYT A2YS0zZmRjNzhjMzI1NDUiLCJhcHBpZGFjciI6IjAiLCJ1X2V4cCI6MzAyNDAwLCJmYW1pbH1fbmFtZSI6I1R1c3QiLCJnaXZ1b19uYw1IjoiTmF2eWEiLCJpcGFkZHIoiIxNjcuMjIwLjEuMTC3Iiwi bmtZSI6Ik5hdnlhIFRlc3QiLCJvaWQoIiIxY2Q0YmNhYy1iODA4LTQyM2EtOWUyZi04Mjd mYmIxYmI3Mzk iLCJwbGF0ZiI6IjMiLCJzY3Ai0iJ1c2Vyx2ltcGVyc29uYXRpb24iLCJzdWIi0iJEVxpYbkdkMDJUk0zRW5pbDFxdjZCakxTNU1lQy0tQ2ZpbzRxS1MzNEc4IiwidGlkIjoiMjYwMzljY2UtNDg5ZC00MDAyLTgyOTMtNWIwYzUxMzR1YWNiIwidW5pcXV1X25hbWUi0iJuYXZ5YUBkZG9iYWpxYW5vdXRs b29rLm9ubWl jcm9zb2Z0LmNvbSI sInVwbiI6Im5hdnlhQGRkb2JhbGlhbm91dGxvb2sub25taWnb3NvZnQuY29tIiwidmVyIjoiMs4wIn0.R-Ke-X071K0r5uLwxB8g5CrcPAwRln5SccJcfEju6IUqpqjcjWcDzeDdNOySiVPDU_ZU5knJmzRCF8fcjFtPsaA4R7vdIEbDuOur15FXSvE8FvVSjP_490H6bHYqoS UAs1N3FMfb06Z8YFCIY4tS0B2I6ahQ_x4ZWFwg1C3w5mK-_4ix81bqi95eV4RUKeFuHhQDXtWhrSgIEC0YiluMvA4TnaJdLq_tWXic4_Tq_KfpkvI0040NKgU7EAMEr1wZ4aDcJV2yf22gQ1sCSig6EGSTm mzDuEPsYiyd4NhidRZJP4HiQh-hePBQsgcSgYGvz9wC6n57ufYKh2wm_Ti3Q&requested_token_use=on_behalf_of&scope=openid
```

Service-to-service access token response

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

PARAMETER	DESCRIPTION
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) .
scope	The scope of access granted in the token.
expires_in	The length of time 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 receiving service (secured resource).
access_token	The requested access token. The calling service can use this token to authenticate to the receiving service.
id_token	The requested ID token. The calling service can use this token to verify the user's identity and begin a session with the user.
refresh_token	The refresh token for the requested access token. The calling service can use this token to request another access token after the current access token expires.

Success response example

The following example shows a success response to a request for an access token for the <https://graph.windows.net> web API.

```
{
  "token_type": "Bearer",
  "scope": "User.Read",
  "expires_in": "43482",
  "ext_expires_in": "302683",
  "expires_on": "1493466951",
  "not_before": "1493423168",
  "resource": "https://graph.windows.net",

  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6InowMz16ZHNGdW16cEJmQ1ZLMVRuMjVRSFlPMCIisImtpZCI6InowMz16ZHNGdW16cEJmQ1ZLMVRuMjVRSFlPMCI9eyJhdWQiOiJodHRwczovL2dyYXB0LndpbmRvd3MubmVOIiwiiaXNzIjoiaHR0cHM6Ly9zdHMud2luZG93cy5uZXQvMjYwMz1jY2UtNDg5ZC00MDAyLTgyOTMtNWlWzUxMzR1YWNIyIsImlhCI6MTQ5MzQyMzE2OCwibmJmIjoxDKzNDIzM TY4LCJleHAIoJE00TM0NjY5NTesImFjci6IjEiLCJhaW8i0iJBu1FBM184REFBQUE1NnZGVmp0WlNjNWdBVWrY1Z0VFPyM0VvV2NvZEoveWV 1S2ZqcTZrdC9NPSIsImFtci6WyJwd2QiXSwiYXBwaWQioi2MjUzOTfHz1jNjc1LTQzzTUtoGU0NC11ZGQzzTMwY2ViMTUiLCJhcHBpZGFjc ii6IjEiLCJ1X2V4cCI6MzAyNjgzLCJmYW1pbHfbmFtZSI6I1R1c3QiLCJnaXZ1b19uYW11IjoiTmf2eWEiLCJpcGFkZHIoIixNjcuMjIwlje uMTc3IiwiibmFtZSI6Ik5hdnlhIFrlc3QiLCJvaWQiOiiXy2Q0YmNhYy1iODA4LTQyM02EtOWUyZi04MjdmYmIxYmI3Mzk1LCJwbGF0Zi6IjMiLCJjwdwlkIjoiMTAwMzNGRkZBMTJFRDdGRSIsInNjci6I1VzZXiUUmVhZCIsInN1YiI6IjNKTulaSWJ1YTc1R2hfWHDm2ZzX0JDc3kxa11ekZ KLTUvM1zD0JuM3ciLCJ0aWQioiIyNjAz0WNjZS000D1kLTQwMDItODI5My01YjbjNTEzNGVhY2IiLCJ1bm1xdWVfbmFtZSI6Im5hdnlhQGRkb 2JhbGlhb91dGxvb2s2b25taWnb3NvZnQuY29tIiwidXbuIjoibmF2eWFAZGrVymFaWFBu3V0bG9vay5vbm1pY3Jvc29mdC5jb20iLCJ1dGk i0iJ4Q3dmemhhLVAvW0pTR0x4Q0dnS0FBiwiidmViyjoiMS4wIn0.cqmUVjfVbqWsxJLUI1Z4FRx1mNQAHPL0F4EMN09r8FY9bIKE0- 0q1eTdP11Nkj_k4BmtaZsTck_mUygdMqEp9AfVYyA1HYvokcgGCW_Z6DM1VGqlIU4ssEkL9abgl1REHE1PhpwBFBBenOk9iHddD1GddTn6vJb KC3qAaNM5VarjSPu50bVvCrqKnvFixTb5bbdnSz- Qr6n6ACiEimiI1aNOPRDeKuyWBPaqcU5EAK0ef5IsVJC1yaYD1AcUYIILMLCD9ebjsy0t9pj_71vjzUSrbMdSCCdzcqez_MSNxrk1Nu9Aecu gkByp3UVUZOIyythVrj6-sVvLZKUutQ", "refresh_token": "AQABAAAAAAABnfG-mA6NTae7CdWw7QfdjKGu9-t1scy_TDEml4eLQmjJGt_nAoVu6A4oSu1KsRiz8XyQIPKQxSGfbf2FoSK- hm2K8TYzbJuswYusQpJaHUQnSqEvdaCeFuqXHBv84wjFhuanzF9dQZB_Ng5za9xK1UENrNtlq9XuLNVKzxEyeUM7JyxzdY7JiEphwImwgOYf6I I316d0Z6-H3oYsFezf4Xsjz-MOBYEov0P64uaB5nJMVdApV-NWpgk1LASfNoSPG6b7Bc02aFRZrm4kLk- xT1eKE6hSo0XU2z2t70stFJDxvNQobnvNhrAmBaHWPaC3FGwFnB0ojpZB2tzG1gLebmdRODp8kHEYAwnRK947Py12fJNKEuD0N0njmXrKxN Z_fem33LHw1Tf4kMX_GvNmblWhtBnIyG0w5emb-b54ef5AwV5_tGUeiVTCyguEc-S7G8Cz0xNj_B0iM_4bAv9iFmr9Stkltpz0- Tftg8WKmaJiC0xXj6uTf4ZkX79mJJiuuM7XP4ARiC1pkkttyg2Iym9jcZqymRkGH2Rm9sxBwC4eeZXm7M5a7TJ- 5Cq0dfuE3sBPq40RdEWMFocrAzFvP0VDR8NKHIrPr1AcUruat9DETmTNJukd1JN3041nWdZOvoJM- uKN3uz2wQ2Ld1z0Mb9_6YfMox9KTJNzRzcL52r4V_y3kB6ekaOZ9wQ3HxGBQ4zFt-2U0mSszIAA", }

  "id_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhdWQiOii2MjUzOTfHz1jNjc1LTQzzTUtoGU0NC11ZGQzzTMwY2ViMTUiLCJpc3Mi0iJodHRwczovL3N0cy53aW5kb3dzLm5ldC8yNjAz0WNjZS000D1kLTQwMDItODI5My01YjbjNTEzNGVhY2IiwiawF0IjoxNDkzNDIzMTY4LCJuYmYi0jE00TM0MjMxNjgsImV4ccI6MTQ5MzQ2Njk1MSwiYw1yIjpBInB3ZCJdLCJmYw1pbHfbmFtZSI6I1R1c3QiLCJnaXZ1b19uYW11IjoiTmf2eWEiLCJpcGFkZHIoIixNjcuMjIwljeuMTc3IiwbmFtZSI6Ik5hdnlhIFrlc3QiLCJvaWQiOiiXy2Q0YmNhYy1iODA4LTQyM2EtOWUyZi04MjdmYmIxYmI3Mzk1LCJwbGF0Zi6IjMiLCJzdWIi0iJEVxPybkdKMDJ1Uk0zRW5pbDFxdjZCakxTNu1lQy0tQ2ZpbzRxS1MzNEc4IiwidGlkIjoiMjYwMz1jY2UtNDg5ZC00MDAyLTgyOTMtNWlWzUxMzR1YWNIiwiidW5pcXV1X25hbWUi0iJuYXZ5YUBkZG9iYwXpYw5vdXRsb29rLm9ubWljcm9zb2Z0LmNbSISInVwbiI6Im5hdnlhQGRkb2JhbGlhb91dGxvb2s2b25taWnb3NvZnQuY29tIiwidXRpIjoiEN3ZnpoYS1QMFdkUU9MeENHZ0tBQSISInZlcii6IjEuMCJ9."
}
```

Error response example

The Azure AD token endpoint returns an error response when it tries to acquire an access token for a downstream API that is set with a Conditional Access policy (for example, multi-factor authentication). The middle-tier service should surface this error to the client application so that the client application can provide the user interaction to satisfy the Conditional Access policy.

```
{
  "error": "interaction_required",
  "error_description": "AADSTS50079: Due to a configuration change made by your administrator, or because you moved to a new location, you must enroll in multi-factor authentication to access 'bf8d80f9-9098-4972-b203-500f535113b1'.\r\nTrace ID: b72a68c3-0926-4b8e-bc35-3150069c2800\r\nCorrelation ID: 73d656cf-54b1-4eb2-b429-26d8165a52d7\r\nTimestamp: 2017-05-01 22:43:20Z",
  "error_codes": [50079],
  "timestamp": "2017-05-01 22:43:20Z",
  "trace_id": "b72a68c3-0926-4b8e-bc35-3150069c2800",
  "correlation_id": "73d656cf-54b1-4eb2-b429-26d8165a52d7",
  "claims": "{\"access_token\":{\"polids\":{\\\"essential\\\":true,\\\"values\\\": [\"9ab03e19-ed42-4168-b6b7-7001fb3e933a\"]}}}"
}
```

Use the access token to access the secured resource

The middle-tier service can use the acquired access token to make authenticated requests to the downstream web API by setting the token in the `Authorization` header.

Example

```
GET /me?api-version=2013-11-08 HTTP/1.1
Host: graph.windows.net
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6InowMz16ZHNGdW16cEJmQ1ZLMVRuMjVRSF1PMCsImtpZCI6InowMz16ZHNGdW16cEJmQ1ZLMVRuMjVRSF1PMCsI9...eyJhdWQiOiJodHRwczovL2dyYXBoLndpbmRvd3Mu...vMjYwMz1jY2UtNDg5ZC00MDAyLTgyOTMtNWiwYzUxMzR1YWNIyIsImlh...TM0NjY5NT...sImFjciI6IjEiLCJhaW8i0iJBu1FBmi84REFBQUE1NnZGVmp0W1NjNWdBVWwrY1Z0FpyM0VvV2NvZEoveWV1S2ZqcTZRdC9NP...sImFtciI6W...CI6MzAyNjgzLCj...CI6...6Ik5hdnlhIFRlc3Q...zNGRkZBM...z...iLCJ0a...2sub25taWNy...wv0pRT...0q1eTdp...0q1eTdp...K...TIP
When you call a SAML-protected web service from a front-end web application, you can simply call the API and initiate a normal interactive authentication flow with the user's existing session. You only need to use an OBO flow when a service-to-service call requires a SAML token to provide user context.

```

NOTE

This is a non-standard extension to the OAuth 2.0 On-Behalf-Of flow that allows an OAuth2-based application to access web service API endpoints that consume SAML tokens.

TIP

When you call a SAML-protected web service from a front-end web application, you can simply call the API and initiate a normal interactive authentication flow with the user's existing session. You only need to use an OBO flow when a service-to-service call requires a SAML token to provide user context.

Obtain a SAML token by using an OBO request with a shared secret

A service-to-service request for a SAML assertion contains the following parameters:

PARAMETER	DESCRIPTION
grant_type	required The type of the token request. For a request that uses a JWT, the value must be <code>urn:ietf:params:oauth:grant-type:jwt-bearer</code> .
assertion	required The value of the access token used in the request.

PARAMETER		DESCRIPTION
client_id	required	The app ID assigned to the calling service during registration with Azure AD. To find the app ID in the Azure portal, select Active Directory , choose the directory, and then select the application name.
client_secret	required	The key registered for the calling service in Azure AD. This value should have been noted at the time of registration.
resource	required	The app ID URI of the receiving service (secured resource). This is the resource that will be the Audience of the SAML token. To find the app ID URI in the Azure portal, select Active Directory and choose the directory. Select the application name, choose All settings , and then select Properties .
requested_token_use	required	Specifies how the request should be processed. In the On-Behalf-Of flow, the value must be on_behalf_of .
requested_token_type	required	Specifies the type of token requested. The value can be urn:ietf:params:oauth:token-type:saml2 or urn:ietf:params:oauth:token-type:saml1 depending on the requirements of the accessed resource.

The response contains a SAML token encoded in UTF8 and Base64url.

- **SubjectConfirmationData for a SAML assertion sourced from an OBO call:** If the target application requires a recipient value in **SubjectConfirmationData**, then the value must be a non-wildcard Reply URL in the resource application configuration.
- **The SubjectConfirmationData node:** The node can't contain an **InResponseTo** attribute since it's not part of a SAML response. The application receiving the SAML token must be able to accept the SAML assertion without an **InResponseTo** attribute.
- **Consent:** Consent must have been granted to receive a SAML token containing user data on an OAuth flow. For information on permissions and obtaining administrator consent, see [Permissions and consent in the Azure Active Directory v1.0 endpoint](#).

Response with SAML assertion

PARAMETER	DESCRIPTION
token_type	Indicates the token type value. The only type that Azure AD supports is Bearer . For more information about bearer tokens, see OAuth 2.0 Authorization Framework: Bearer Token Usage (RFC 6750) .
scope	The scope of access granted in the token.

PARAMETER	DESCRIPTION
expires_in	The length of time 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 receiving service (secured resource).
access_token	The parameter that returns the SAML assertion.
refresh_token	The refresh token. The calling service can use this token to request another access token after the current SAML assertion expires.

- token_type: Bearer
- expires_in: 3296
- ext_expires_in: 0
- expires_on: 1529627844
- resource: `https://api.contoso.com`
- access_token: <SAML assertion>
- issued_token_type: urn:ietf:params:oauth:token-type:saml2
- refresh_token: <Refresh token>

Client limitations

Public clients with wildcard reply URLs can't use an `id_token` for OBO flows. However, a confidential client can still redeem **access** tokens acquired through the implicit grant flow even if the public client has a wildcard redirect URI registered.

Next steps

Learn more about the OAuth 2.0 protocol and another way to perform service-to-service authentication that uses client credentials:

- [Service to service authentication using OAuth 2.0 client credentials grant in Azure AD](#)
- [OAuth 2.0 in Azure AD](#)

Service to service calls using client credentials (shared secret or certificate)

8/6/2019 • 4 minutes to read • [Edit Online](#)

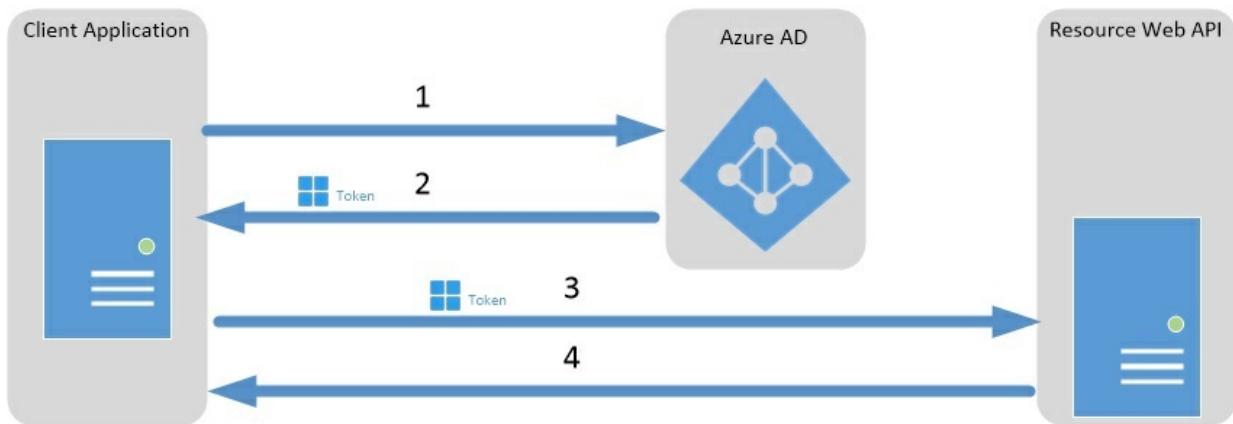
Applies to:

- Azure AD v1.0 endpoint

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](#).

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 .
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 <i>Application ID</i>
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-
e97cf07a22de&client_secret=qkDwDJ1Dfig2IpeuUZYKH1Wb8q1V0ju6sILxQQqhJ+s=&resource=https%3A%2F%2Fservice.contos
o.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 <i>Application ID</i>
client_assertion_type	required	The value must be <code>urn:ietf:params:oauth:client-assertion-type:jwt-bearer</code>
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 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 registrations , 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%2Ffc7664b4-cdd6-43e1-9365-c2e1c4e1b3bf&client_id=97e0a5b7-d745-40b6-94fe-5f77d35c6e05&client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer&client_assertion=eyJhbGciOiJSUzI1NiIsIng1dCI6Imd40HRHeXN5amNScUtqr1BuZDdSRnd2d1pJMCJ9.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) .

PARAMETER	DESCRIPTION
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": "eyJhbGciOiJSUzI1NiIsIng1dCI6IjdkRC1nZWNOZ1gxWmY3R0xrT3ZwT0IyZGNwQSIisInR5cCI6IkpxVCJ9.eyJhdWQiOiJodHRwczovL3N1cnZpY2UUy29udG9zby5jb20vIiwiaXNzIjoiaHR0cHM6Ly9zdHMud2luZG93cy5uZXQvN2ZlODE0NDctZGE1Ny00Mzg1LWJ1Y2ItNmR1NTdmMjE0Nzd1LyIsImlhcdCI6MTM4ODQ0ODI2NywibmJmIjoxMzg4NDQ4MjY3LCJleHAiOjEzODg0NTIxNjcsInZlcI6IjEuMCIsInRpZCI6IjdmZTgxNDQ3LWRhNTctNDM4NS1iZWNiLTZkZTU3ZjIxNDc3ZSIisIm9pZCI6ImE5OTE5MTYyLTkyMtctND1kYS1hZTIyLWYxMTM3YzI1Y2R1YSIsInN1Yi6ImE5OTE5MTYyLTkyMTctND1kYS1hZTIyLWYxMTM3YzI1Y2R1YSIsIm1kcCI6Imh0dHBzOi8vc3RzLndpbmRvd3MubmV0LzdmZTgxNDQ3LWRhNTctNDM4NS1iZWNiLTZkZTU3ZjIxNDc3ZS8iLCJhcHBpZCI6ImQxN2QxNWJjLWM1NzYtNDF1NS05MjdmLWRiNWYzMGRkNThmMSIsImFwcGlkYWNyIjoiMSJ9.aqtfJ7G37CpKV901Vm9sGiQhde0WMg6luYJR4wuNR2ffaQsVPPpKirM5rbcb6o5CmW10tmaAIdwDcL6i9ZT9ooIIicSRrjCYMYWHX08ip-tj-uWUiHgztI02xKdWiycItplwiHxapQm0a8Ti1CWRjJghORC1B1-fah_yWx6Cjuf4QE8xJcu-ZHX0pVZNPX22PHYV5Km-vPTq2HtIqdboKyZy3Y4y3geOrRIFE1ZYoqjqSv5q9Jgtj5ERsNQIjeFpyxW3EwPtFqMcDm4ebiAEpoEWRN4QYOMxnC90UBeG9oLA01TfmhgHLAtvJogJcYFzwngTsVo6HznsvPWy7UP3MINA",
  "token_type": "Bearer",
  "expires_in": "3599",
  "expires_on": "1388452167",
  "resource": "https://service.contoso.com/"
}
```

See also

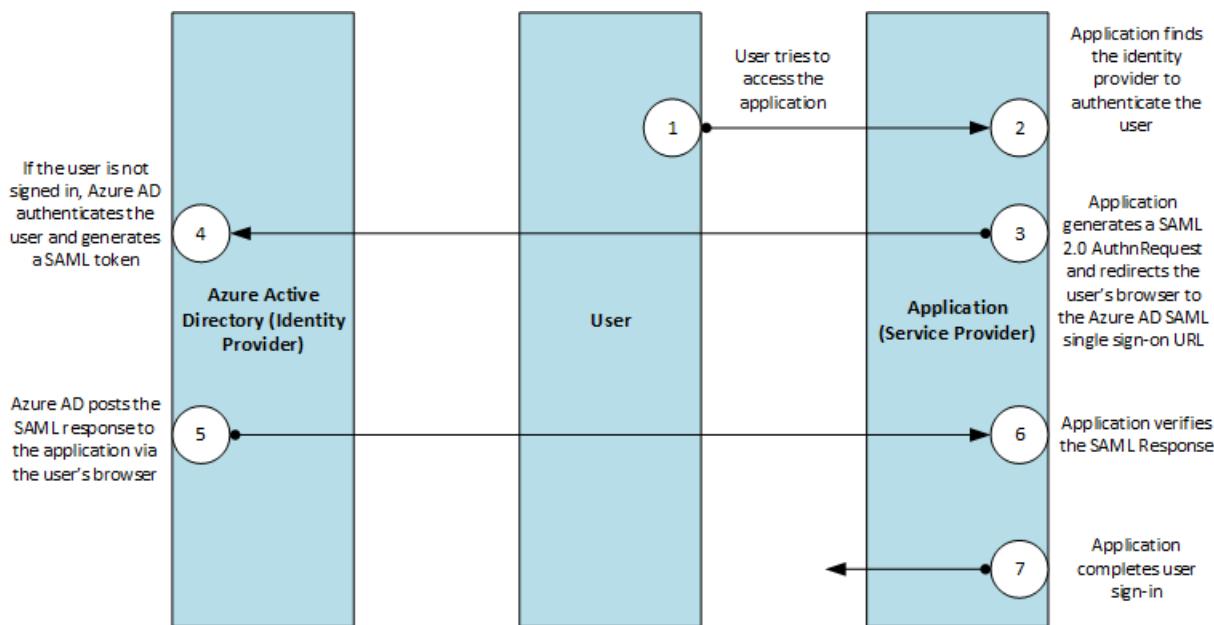
- [OAuth 2.0 in Azure AD](#)
- [Sample in C# of the service to service call with a shared secret](#) and [Sample in C# of the service to service call with a certificate](#)

Single Sign-On SAML protocol

8/6/2019 • 7 minutes to read • [Edit Online](#)

This article covers the SAML 2.0 authentication requests and responses that Azure Active Directory (Azure AD) supports for Single Sign-On.

The protocol diagram below describes the single sign-on sequence. The cloud service (the service provider) uses an HTTP Redirect binding to pass an `AuthnRequest` (authentication request) element to Azure AD (the identity provider). Azure AD then uses an HTTP post binding to post a `Response` element to the cloud service.



AuthnRequest

To request a user authentication, cloud services send an `AuthnRequest` element to Azure AD. A sample SAML 2.0 `AuthnRequest` could look like the following example:

```
<samlp:AuthnRequest
  xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
  ID="id6c1c178c166d486687be4aaf5e482730"
  Version="2.0" IssueInstant="2013-03-18T03:28:54.1839884Z"
  xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">
  <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">https://www.contoso.com</Issuer>
</samlp:AuthnRequest>
```

PARAMETER	DESCRIPTION
ID	Required Azure AD uses this attribute to populate the <code>InResponseTo</code> attribute of the returned response. ID must not begin with a number, so a common strategy is to prepend a string like "id" to the string representation of a GUID. For example, <code>id6c1c178c166d486687be4aaf5e482730</code> is a valid ID.

PARAMETER		DESCRIPTION
Version	Required	This parameter should be set to 2.0 .
IssueInstant	Required	This is a DateTime string with a UTC value and round-trip format ("o") . Azure AD expects a DateTime value of this type, but doesn't evaluate or use the value.
AssertionConsumerServiceUrl	Optional	If provided, this parameter must match the <code>RedirectUri</code> of the cloud service in Azure AD.
ForceAuthn	Optional	This is a boolean value. If true, it means that the user will be forced to re-authenticate, even if they have a valid session with Azure AD.
IsPassive	Optional	This is a boolean value that specifies whether Azure AD should authenticate the user silently, without user interaction, using the session cookie if one exists. If this is true, Azure AD will attempt to authenticate the user using the session cookie.

All other `AuthnRequest` attributes, such as Consent, Destination, AssertionConsumerServiceIndex, AttributeConsumerServiceIndex, and ProviderName are **ignored**.

Azure AD also ignores the `Conditions` element in `AuthnRequest`.

Issuer

The `Issuer` element in an `AuthnRequest` must exactly match one of the **ServicePrincipalNames** in the cloud service in Azure AD. Typically, this is set to the **App ID URI** that is specified during application registration.

A SAML excerpt containing the `Issuer` element looks like the following sample:

```
<Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">https://www.contoso.com</Issuer>
```

NameIDPolicy

This element requests a particular name ID format in the response and is optional in `AuthnRequest` elements sent to Azure AD.

A `NameIDPolicy` element looks like the following sample:

```
<NameIDPolicy Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent"/>
```

If `NameIDPolicy` is provided, you can include its optional `Format` attribute. The `Format` attribute can have only one of the following values; any other value results in an error.

- `urn:oasis:names:tc:SAML:2.0:nameid-format:persistent` : Azure Active Directory issues the NameID claim as a pairwise identifier.
- `urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress` : Azure Active Directory issues the NameID claim in e-mail address format.

- `urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified` : This value permits Azure Active Directory to select the claim format. Azure Active Directory issues the NameID as a pairwise identifier.
- `urn:oasis:names:tc:SAML:2.0:nameid-format:transient` : Azure Active Directory issues the NameID claim as a randomly generated value that is unique to the current SSO operation. This means that the value is temporary and cannot be used to identify the authenticating user.

Azure AD ignores the `AllowCreate` attribute.

RequestAuthnContext

The `RequestedAuthnContext` element specifies the desired authentication methods. It is optional in `AuthnRequest` elements sent to Azure AD. Azure AD supports `AuthnContextClassRef` values such as `urn:oasis:names:tc:SAML:2.0:ac:classes>Password`.

Scoping

The `scoping` element, which includes a list of identity providers, is optional in `AuthnRequest` elements sent to Azure AD.

If provided, don't include the `ProxyCount` attribute, `IDPListOption` or `RequesterID` element, as they aren't supported.

Signature

Don't include a `Signature` element in `AuthnRequest` elements, as Azure AD does not support signed authentication requests.

Subject

Azure AD ignores the `Subject` element of `AuthnRequest` elements.

Response

When a requested sign-on completes successfully, Azure AD posts a response to the cloud service. A response to a successful sign-on attempt looks like the following sample:

```

<samlp:Response ID="_a4958bfd-e107-4e67-b06d-0d85ade2e76a" Version="2.0" IssueInstant="2013-03-18T07:38:15.144Z" Destination="https://contoso.com/identity/inboundsso.aspx"
InResponseTo="id758d0ef385634593a77bdf7e632984b6" xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">
  <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion"> https://login.microsoftonline.com/82869000-6ad1-48f0-8171-272ed18796e9/</Issuer>
  <ds:Signature xmlns:ds="https://www.w3.org/2000/09/xmldsig#">
    ...
  </ds:Signature>
  <samlp:Status>
    <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success" />
  </samlp:Status>
  <Assertion ID="_bf9c623d-cc20-407a-9a59-c2d0aee84d12" IssueInstant="2013-03-18T07:38:15.144Z" Version="2.0" xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
    <Issuer>https://login.microsoftonline.com/82869000-6ad1-48f0-8171-272ed18796e9/</Issuer>
    <ds:Signature xmlns:ds="https://www.w3.org/2000/09/xmldsig#">
      ...
    </ds:Signature>
    <Subject>
      <NameID>Uz2Pqz1X7pxe4XLWxV9KJQ+n59d573SepSAkuYKSde8=</NameID>
      <SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
        <SubjectConfirmationData InResponseTo="id758d0ef385634593a77bdf7e632984b6" NotOnOrAfter="2013-03-18T07:43:15.144Z" Recipient="https://contoso.com/identity/inboundsso.aspx" />
      </SubjectConfirmation>
    </Subject>
    <Conditions NotBefore="2013-03-18T07:38:15.128Z" NotOnOrAfter="2013-03-18T08:48:15.128Z">
      <AudienceRestriction>
        <Audience>https://www.contoso.com</Audience>
      </AudienceRestriction>
    </Conditions>
    <AttributeStatement>
      <Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name">
        <AttributeValue>testuser@contoso.com</AttributeValue>
      </Attribute>
      <Attribute Name="http://schemas.microsoft.com/identity/claims/objectidentifier">
        <AttributeValue>3F2504E0-4F89-11D3-9A0C-0305E82C3301</AttributeValue>
      </Attribute>
      ...
    </AttributeStatement>
    <AuthnStatement AuthnInstant="2013-03-18T07:33:56.000Z" SessionIndex="_bf9c623d-cc20-407a-9a59-c2d0aee84d12">
      <AuthnContext>
        <AuthnContextClassRef> urn:oasis:names:tc:SAML:2.0:ac:classes:Password</AuthnContextClassRef>
      </AuthnContext>
    </AuthnStatement>
  </Assertion>
</samlp:Response>

```

Response

The `Response` element includes the result of the authorization request. Azure AD sets the `ID`, `Version` and `IssueInstant` values in the `Response` element. It also sets the following attributes:

- `Destination`: When sign-on completes successfully, this is set to the `RedirectUri` of the service provider (cloud service).
- `InResponseTo`: This is set to the `ID` attribute of the `AuthnRequest` element that initiated the response.

Issuer

Azure AD sets the `Issuer` element to `https://login.microsoftonline.com/<TenantIDGUID>/` where `<TenantIDGUID>` is the tenant ID of the Azure AD tenant.

For example, a response with `Issuer` element could look like the following sample:

```
<Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion"> https://login.microsoftonline.com/82869000-6ad1-48f0-8171-272ed18796e9/</Issuer>
```

Status

The `status` element conveys the success or failure of sign-on. It includes the `StatusCode` element, which contains a code or a set of nested codes that represents the status of the request. It also includes the `StatusMessage` element, which contains custom error messages that are generated during the sign-on process.

The following sample is a SAML response to an unsuccessful sign-on attempt.

```
<samlp:Response ID="_f0961a83-d071-4be5-a18c-9ae7b22987a4" Version="2.0" IssueInstant="2013-03-18T08:49:24.405Z" InResponseTo="iddce91f96e56747b5ace6d2e2aa9d4f8c" xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">
  <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">https://sts.windows.net/82869000-6ad1-48f0-8171-272ed18796e9/</Issuer>
  <samlp:Status>
    <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Requester">
      <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:RequestUnsupported" />
    </samlp:StatusCode>
    <samlp:StatusMessage>AADSTS75006: An error occurred while processing a SAML2 Authentication request. AADSTS90011: The SAML authentication request property 'NameIdentifierPolicy/SPNameQualifier' is not supported. Trace ID: 66febed4-e737-49ff-ac23-464ba090d57c Timestamp: 2013-03-18 08:49:24Z</samlp:StatusMessage>
  </samlp:Status>
```

Assertion

In addition to the `ID`, `IssueInstant` and `Version`, Azure AD sets the following elements in the `Assertion` element of the response.

Issuer

This is set to `https://sts.windows.net/<TenantIDGUID>/` where `<TenantIDGUID>` is the Tenant ID of the Azure AD tenant.

```
<Issuer>https://login.microsoftonline.com/82869000-6ad1-48f0-8171-272ed18796e9/</Issuer>
```

Signature

Azure AD signs the assertion in response to a successful sign-on. The `Signature` element contains a digital signature that the cloud service can use to authenticate the source to verify the integrity of the assertion.

To generate this digital signature, Azure AD uses the signing key in the `IDPSSODescriptor` element of its metadata document.

```
<ds:Signature xmlns:ds="https://www.w3.org/2000/09/xmldsig#">
  digital_signature_here
</ds:Signature>
```

Subject

This specifies the principal that is the subject of the statements in the assertion. It contains a `NameID` element, which represents the authenticated user. The `NameID` value is a targeted identifier that is directed only to the service provider that is the audience for the token. It is persistent - it can be revoked, but is never reassigned. It is also opaque, in that it does not reveal anything about the user and cannot be used as an identifier for attribute queries.

The `Method` attribute of the `SubjectConfirmation` element is always set to `urn:oasis:names:tc:SAML:2.0:cm:bearer`.

```

<Subject>
  <NameID>Uz2Pqz1X7pxe4XLWxV9KJQ+n59d573SepSAkuYKSde8=</NameID>
  <SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
    <SubjectConfirmationData InResponseTo="id758d0ef385634593a77bdf7e632984b6" NotOnOrAfter="2013-03-18T07:43:15.144Z" Recipient="https://contoso.com/identity/inboundsso.aspx" />
  </SubjectConfirmation>
</Subject>

```

Conditions

This element specifies conditions that define the acceptable use of SAML assertions.

```

<Conditions NotBefore="2013-03-18T07:38:15.128Z" NotOnOrAfter="2013-03-18T08:48:15.128Z">
  <AudienceRestriction>
    <Audience>https://www.contoso.com</Audience>
  </AudienceRestriction>
</Conditions>

```

The `NotBefore` and `NotOnOrAfter` attributes specify the interval during which the assertion is valid.

- The value of the `NotBefore` attribute is equal to or slightly (less than a second) later than the value of `IssueInstant` attribute of the `Assertion` element. Azure AD does not account for any time difference between itself and the cloud service (service provider), and does not add any buffer to this time.
- The value of the `NotOnOrAfter` attribute is 70 minutes later than the value of the `NotBefore` attribute.

Audience

This contains a URI that identifies an intended audience. Azure AD sets the value of this element to the value of `Issuer` element of the `AuthnRequest` that initiated the sign-on. To evaluate the `Audience` value, use the value of the `App ID URI` that was specified during application registration.

```

<AudienceRestriction>
  <Audience>https://www.contoso.com</Audience>
</AudienceRestriction>

```

Like the `Issuer` value, the `Audience` value must exactly match one of the service principal names that represents the cloud service in Azure AD. However, if the value of the `Issuer` element is not a URL value, the `Audience` value in the response is the `Issuer` value prefixed with `spn:`.

AttributeStatement

This contains claims about the subject or user. The following excerpt contains a sample `AttributeStatement` element. The ellipsis indicates that the element can include multiple attributes and attribute values.

```

<AttributeStatement>
  <Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name">
    <AttributeValue>testuser@contoso.com</AttributeValue>
  </Attribute>
  <Attribute Name="http://schemas.microsoft.com/identity/claims/objectidentifier">
    <AttributeValue>3F2504E0-4F89-11D3-9A0C-0305E82C3301</AttributeValue>
  </Attribute>
  ...
</AttributeStatement>

```

- **Name Claim** - The value of the `Name` attribute (`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name`) is the user principal name of the authenticated user, such as `testuser@managedtenant.com`.
- **ObjectIdentifier Claim** - The value of the `ObjectIdentifier` attribute (`http://schemas.microsoft.com/identity/claims/objectidentifier`) is the `objectId` of the directory object that

represents the authenticated user in Azure AD. `objectId` is an immutable, globally unique, and reuse safe identifier of the authenticated user.

AuthnStatement

This element asserts that the assertion subject was authenticated by a particular means at a particular time.

- The `AuthnInstant` attribute specifies the time at which the user authenticated with Azure AD.
- The `AuthnContext` element specifies the authentication context used to authenticate the user.

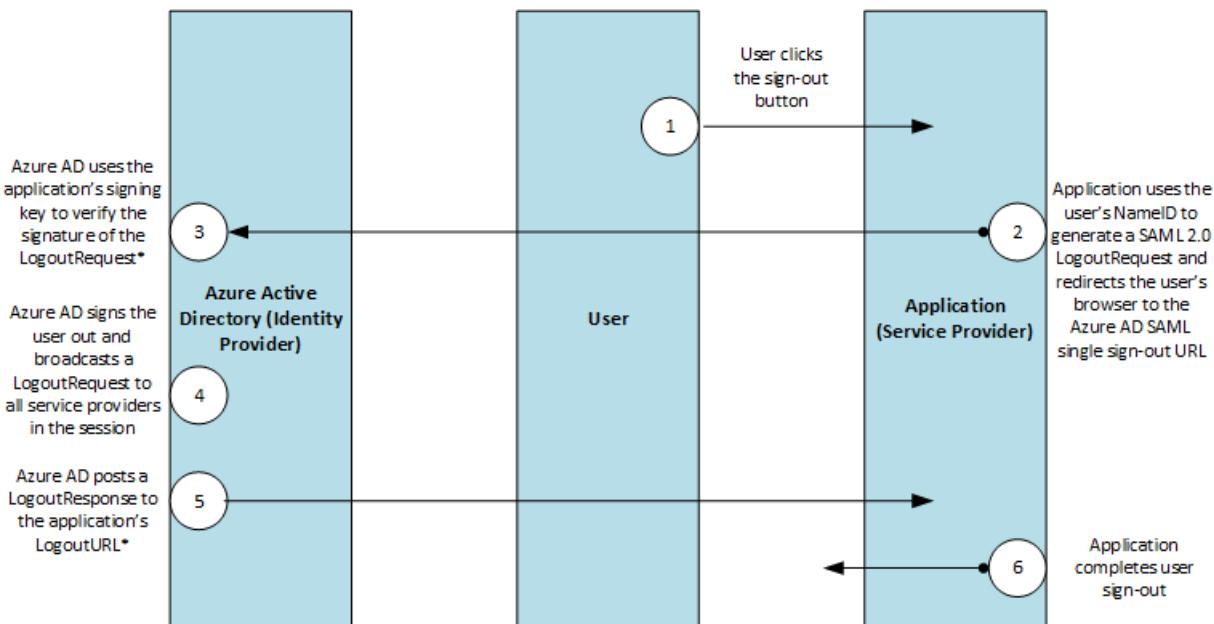
```
<AuthnStatement AuthnInstant="2013-03-18T07:33:56.000Z" SessionIndex="_bf9c623d-cc20-407a-9a59-c2d0aeee84d12">
  <AuthnContext>
    <AuthnContextClassRef> urn:oasis:names:tc:SAML:2.0:ac:classes:Password</AuthnContextClassRef>
  </AuthnContext>
</AuthnStatement>
```

Single Sign-Out SAML Protocol

8/6/2019 • 2 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) supports the SAML 2.0 web browser single sign-out profile. For single sign-out to work correctly, the **LogoutURL** for the application must be explicitly registered with Azure AD during application registration. Azure AD uses the LogoutURL to redirect users after they're signed out.

The following diagram shows the workflow of the Azure AD single sign-out process.



* Azure AD gets the signing key and LogoutURL of the application from the application metadata

LogoutRequest

The cloud service sends a `LogoutRequest` message to Azure AD to indicate that a session has been terminated. The following excerpt shows a sample `LogoutRequest` element.

```
<samlp:LogoutRequest xmlns="urn:oasis:names:tc:SAML:2.0:metadata" ID="idaa6ebe6839094fe4abc4ebd5281ec780" Version="2.0" IssueInstant="2013-03-28T07:10:49.600482Z" xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">
  <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">https://www.workaad.com</Issuer>
  <NameID xmlns="urn:oasis:names:tc:SAML:2.0:assertion"> Uz2Pqz1X7pxe4XLWxV9KJQ+n59d573SepSAkuYKSde8=</NameID>
</samlp:LogoutRequest>
```

LogoutRequest

The `LogoutRequest` element sent to Azure AD requires the following attributes:

- `ID` - This identifies the sign-out request. The value of `ID` should not begin with a number. The typical practice is to append `id` to the string representation of a GUID.
- `Version` - Set the value of this element to **2.0**. This value is required.
- `IssueInstant` - This is a `DateTime` string with a Coordinate Universal Time (UTC) value and [round-trip format \("o"\)](#). Azure AD expects a value of this type, but doesn't enforce it.

Issuer

The `Issuer` element in a `LogoutRequest` must exactly match one of the **ServicePrincipalNames** in the cloud service in Azure AD. Typically, this is set to the **App ID URI** that is specified during application registration.

NameID

The value of the `NameID` element must exactly match the `NameID` of the user that is being signed out.

LogoutResponse

Azure AD sends a `LogoutResponse` in response to a `LogoutRequest` element. The following excerpt shows a sample `LogoutResponse`.

```
<samlp:LogoutResponse ID="_f0961a83-d071-4be5-a18c-9ae7b22987a4" Version="2.0" IssueInstant="2013-03-18T08:49:24.405Z" InResponseTo="iddce91f96e56747b5ace6d2e2aa9d4f8c"
  xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">
  <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">https://sts.windows.net/82869000-6ad1-48f0-8171-272ed18796e9/</Issuer>
  <samlp>Status>
    <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success" />
  </samlp>Status>
</samlp:LogoutResponse>
```

LogoutResponse

Azure AD sets the `ID`, `Version` and `IssueInstant` values in the `LogoutResponse` element. It also sets the `InResponseTo` element to the value of the `ID` attribute of the `LogoutRequest` that elicited the response.

Issuer

Azure AD sets this value to `https://login.microsoftonline.com/<TenantIdGUID>/` where `<TenantIdGUID>` is the tenant ID of the Azure AD tenant.

To evaluate the value of the `Issuer` element, use the value of the **App ID URI** provided during application registration.

Status

Azure AD uses the `statusCode` element in the `Status` element to indicate the success or failure of sign-out. When the sign-out attempt fails, the `statusCode` element can also contain custom error messages.

How Azure AD uses the SAML protocol

8/6/2019 • 2 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) uses the SAML 2.0 protocol to enable applications to provide a single sign-on experience to their users. The [Single Sign-On](#) and [Single Sign-Out](#) SAML profiles of Azure AD explain how SAML assertions, protocols, and bindings are used in the identity provider service.

SAML Protocol requires the identity provider (Azure AD) and the service provider (the application) to exchange information about themselves.

When an application is registered with Azure AD, the app developer registers federation-related information with Azure AD. This information includes the **Redirect URI** and **Metadata URI** of the application.

Azure AD uses the cloud service's **Metadata URI** to retrieve the signing key and the logout URI. Customer can open the app in **Azure AD -> App Registration** and then in **Settings -> Properties**, they can update the Logout URL. This way Azure AD can send the response to the correct URL.

Azure Active Directory exposes tenant-specific and common (tenant-independent) single sign-on and single sign-out endpoints. These URLs represent addressable locations -- they are not just identifiers -- so you can go to the endpoint to read the metadata.

- The tenant-specific endpoint is located at

`https://login.microsoftonline.com/<TenantDomainName>/FederationMetadata/2007-06/FederationMetadata.xml`.

The `<TenantDomainName>` placeholder represents a registered domain name or TenantID GUID of an Azure AD tenant. For example, the federation metadata of the contoso.com tenant is at:

<https://login.microsoftonline.com/contoso.com/FederationMetadata/2007-06/FederationMetadata.xml>

- The tenant-independent endpoint is located at

`https://login.microsoftonline.com/common/FederationMetadata/2007-06/FederationMetadata.xml`. In this endpoint address, **common** appears instead of a tenant domain name or ID.

For information about the federation metadata documents that Azure AD publishes, see [Federation Metadata](#).

Federation metadata

8/6/2019 • 4 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) publishes a federation metadata document for services that is configured to accept the security tokens that Azure AD issues. The federation metadata document format is described in the [Web Services Federation Language \(WS-Federation\) Version 1.2](#), which extends [Metadata for the OASIS Security Assertion Markup Language \(SAML\) v2.0](#).

Tenant-specific and Tenant-independent metadata endpoints

Azure AD publishes tenant-specific and tenant-independent endpoints.

Tenant-specific endpoints are designed for a particular tenant. The tenant-specific federation metadata includes information about the tenant, including tenant-specific issuer and endpoint information. Applications that restrict access to a single tenant use tenant-specific endpoints.

Tenant-independent endpoints provide information that is common to all Azure AD tenants. This information applies to tenants hosted at login.microsoftonline.com and is shared across tenants. Tenant-independent endpoints are recommended for multi-tenant applications, since they are not associated with any particular tenant.

Federation metadata endpoints

Azure AD publishes federation metadata at

`https://login.microsoftonline.com/<TenantDomainName>/FederationMetadata/2007-06/FederationMetadata.xml`.

For **tenant-specific endpoints**, the `<TenantDomainName>` can be one of the following types:

- A registered domain name of an Azure AD tenant, such as: `contoso.onmicrosoft.com`.
- The immutable tenant ID of the domain, such as `72f988bf-86f1-41af-91ab-2d7cd011db45`.

For **tenant-independent endpoints**, the `<TenantDomainName>` is `common`. This document lists only the Federation Metadata elements that are common to all Azure AD tenants that are hosted at login.microsoftonline.com.

For example, a tenant-specific endpoint might be

`https://login.microsoftonline.com/contoso.onmicrosoft.com/FederationMetadata/2007-06/FederationMetadata.xml`.

The tenant-independent endpoint is <https://login.microsoftonline.com/common/FederationMetadata/2007-06/FederationMetadata.xml>. You can view the federation metadata document by typing this URL in a browser.

Contents of federation Metadata

The following section provides information needed by services that consume the tokens issued by Azure AD.

Entity ID

The `<EntityDescriptor>` element contains an `<EntityID>` attribute. The value of the `<EntityID>` attribute represents the issuer, that is, the security token service (STS) that issued the token. It is important to validate the issuer when you receive a token.

The following metadata shows a sample tenant-specific `<EntityDescriptor>` element with an `<EntityID>` element.

```

<EntityDescriptor
  xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
  ID="_b827a749-cfcf-46b3-ab8b-9f6d14a1294b"
  entityID="https://sts.windows.net/72f988bf-86f1-41af-91ab-2d7cd011db45"/>

```

You can replace the tenant ID in the tenant-independent endpoint with your tenant ID to create a tenant-specific `EntityID` value. The resulting value will be the same as the token issuer. The strategy allows a multi-tenant application to validate the issuer for a given tenant.

The following metadata shows a sample tenant-independent `EntityID` element. Please note, that the `{tenant}` is a literal, not a placeholder.

```

<EntityDescriptor
  xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
  ID="_0e5bd9d0-49ef-4258-bc15-21ce143b61bd"
  entityID="https://sts.windows.net/{tenant}"/>

```

Token signing certificates

When a service receives a token that is issued by an Azure AD tenant, the signature of the token must be validated with a signing key that is published in the federation metadata document. The federation metadata includes the public portion of the certificates that the tenants use for token signing. The certificate raw bytes appear in the `KeyDescriptor` element. The token signing certificate is valid for signing only when the value of the `use` attribute is `signing`.

A federation metadata document published by Azure AD can have multiple signing keys, such as when Azure AD is preparing to update the signing certificate. When a federation metadata document includes more than one certificate, a service that is validating the tokens should support all certificates in the document.

The following metadata shows a sample `KeyDescriptor` element with a signing key.

```

<KeyDescriptor use="signing">
  <KeyInfo xmlns="https://www.w3.org/2000/09/xmldsig#">
    <X509Data>
      <X509Certificate>
        MIIDPjCCAiqgAwIBAgIQVWmXY/+9RqFA/0G9kFulHDAJBgUrDgMCHQUAMC0xKzApBgNVBAMTImFjY291bnRzLmFjY2Vzc2NvbnRyb2wud2luZG93cy5uZXQwHhcNMTIwNjA3MDcwMDAwWhcNMTQwNjA3MDcwMDAwWjAtMSswKQYDVQDDEyJhY2NvdW50cy5hY2N1c3Njb250cm9sLndpbmRvd3MubmV0MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEArCz8Sn3GGXmikH2MdTeGY1D711EORX/1VXpr+ecGgqfUWF8MPB07XkYuJ54DAuYT318+2XrzMj0tqkT94VkJmxv6dFGhG8YZ8vNMPd4tdj9c01pvWQdqXtL1tFRpD/P6UMEigfN0c9oWDg9U7Ilymgei0UXtf1gtcQbc5sSQu0S4vr9YJp2gLFIGK11Iqg4XSGdcI0QWLLkkC6cBukhVnd6BCYbLjTYy3fNs4DzNdemJlxGl8sLexFytBF6YApvSdus3nFXaMCtBGx16HzkK9ne3lobAwL2o79bP4imEGqg+ibvyNmbrwFGnQrBc1jTF9LyQX9q+louxVfHs6ZiVwIDAQABo2IwYDBeBgNVHQEEVzBvgBCxDsLd8xkfOLKm4Q/SzjtoS8wLTEmcKA1UEAxMiYWNjb3VudHMuYWNjZXNzY29udHJvbC53aw5kb3dzLm51dIIQVWmXY/+9RqFA/0G9kFulHDAJBgUrDgMCHQUAA4IBAQAkJtxxm/ErgyS1Nk69+1odTMP80y6L0H17z7XGG3w4TqvTUSWaxD4hSFJ0e7mHLQLD7oV/erACxwSzn2pMoZ89MBDj0MQA+e6QzGB7jmSzPTNmQgMLA8fWCfqPrz6zgH+1F1gNp8hJY57kfeVPBiyyuBmlTEBsBlzo1Y9dd/55qqfQk6cgSeCbHcy/RU/iep0+UsRM1SgPNNmqhj5gmN2AFVCN96zF694LwuPae5CeR2ZcVknexOWHYjFM0MgUSw0ubnG10h9AJgGyhNGcjQqu9vd1xkupFgaN+f7P3p3EVN5csBgs5H94jEcQZT7EKeTiZ6bTrpDAnrr8tDCy8ng
      </X509Certificate>
    </X509Data>
  </KeyInfo>
</KeyDescriptor>

```

The `KeyDescriptor` element appears in two places in the federation metadata document; in the WS-Federation-specific section and the SAML-specific section. The certificates published in both sections will be the same.

In the WS-Federation-specific section, a WS-Federation metadata reader would read the certificates from a `RoleDescriptor` element with the `SecurityTokenServiceType` type.

The following metadata shows a sample `RoleDescriptor` element.

```
<RoleDescriptor xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:fed="https://docs.oasis-open.org/wsfed/federation/200706"
xsi:type="fed:SecurityTokenServiceType" protocolSupportEnumeration="https://docs.oasis-open.org/wsfed/federation/200706">
```

In the SAML-specific section, a WS-Federation metadata reader would read the certificates from a `IDPSSODescriptor` element.

The following metadata shows a sample `IDPSSODescriptor` element.

```
<IDPSSODescriptor protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
```

There are no differences in the format of tenant-specific and tenant-independent certificates.

WS-Federation endpoint URL

The federation metadata includes the URL that Azure AD uses for single sign-in and single sign-out in WS-Federation protocol. This endpoint appears in the `PassiveRequestorEndpoint` element.

The following metadata shows a sample `PassiveRequestorEndpoint` element for a tenant-specific endpoint.

```
<fed:PassiveRequestorEndpoint>
<EndpointReference xmlns="https://www.w3.org/2005/08/addressing">
<Address>
https://login.microsoftonline.com/72f988bf-86f1-41af-91ab-2d7cd011db45/wsfed
</Address>
</EndpointReference>
</fed:PassiveRequestorEndpoint>
```

For the tenant-independent endpoint, the WS-Federation URL appears in the WS-Federation endpoint, as shown in the following sample.

```
<fed:PassiveRequestorEndpoint>
<EndpointReference xmlns="https://www.w3.org/2005/08/addressing">
<Address>
https://login.microsoftonline.com/common/wsfed
</Address>
</EndpointReference>
</fed:PassiveRequestorEndpoint>
```

SAML protocol endpoint URL

The federation metadata includes the URL that Azure AD uses for single sign-in and single sign-out in SAML 2.0 protocol. These endpoints appear in the `IDPSSODescriptor` element.

The sign-in and sign-out URLs appear in the `SingleSignOnService` and `SingleLogoutService` elements.

The following metadata shows a sample `PassiveRequestorEndpoint` for a tenant-specific endpoint.

```
<IDPSSODescriptor protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
...
<SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="https://login.microsoftonline.com/contoso.onmicrosoft.com/saml2" />
<SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="https://login.microsoftonline.com/contoso.onmicrosoft.com /saml2" />
</IDPSSODescriptor>
```

Similarly the endpoints for the common SAML 2.0 protocol endpoints are published in the tenant-independent

federation metadata, as shown in the following sample.

```
<IDPSSODescriptor protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
...
<SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="https://login.microsoftonline.com/common/saml2" />
<SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect"
Location="https://login.microsoftonline.com/common/saml2" />
</IDPSSODescriptor>
```

Signing key rollover in Azure Active Directory

8/6/2019 • 13 minutes to read • [Edit Online](#)

This article discusses what you need to know about the public keys that are used in Azure Active Directory (Azure AD) to sign security tokens. It is important to note that these keys roll over on a periodic basis and, in an emergency, could be rolled over immediately. All applications that use Azure AD should be able to programmatically handle the key rollover process or establish a periodic manual rollover process. Continue reading to understand how the keys work, how to assess the impact of the rollover to your application and how to update your application or establish a periodic manual rollover process to handle key rollover if necessary.

Overview of signing keys in Azure AD

Azure AD uses public-key cryptography built on industry standards to establish trust between itself and the applications that use it. In practical terms, this works in the following way: Azure AD uses a signing key that consists of a public and private key pair. When a user signs in to an application that uses Azure AD for authentication, Azure AD creates a security token that contains information about the user. This token is signed by Azure AD using its private key before it is sent back to the application. To verify that the token is valid and originated from Azure AD, the application must validate the token's signature using the public key exposed by Azure AD that is contained in the tenant's [OpenID Connect discovery document](#) or SAML/WS-Fed [federation metadata document](#).

For security purposes, Azure AD's signing key rolls on a periodic basis and, in the case of an emergency, could be rolled over immediately. Any application that integrates with Azure AD should be prepared to handle a key rollover event no matter how frequently it may occur. If it doesn't, and your application attempts to use an expired key to verify the signature on a token, the sign-in request will fail.

There is always more than one valid key available in the OpenID Connect discovery document and the federation metadata document. Your application should be prepared to use any of the keys specified in the document, since one key may be rolled soon, another may be its replacement, and so forth.

How to assess if your application will be affected and what to do about it

How your application handles key rollover depends on variables such as the type of application or what identity protocol and library was used. The sections below assess whether the most common types of applications are impacted by the key rollover and provide guidance on how to update the application to support automatic rollover or manually update the key.

- [Native client applications accessing resources](#)
- [Web applications / APIs accessing resources](#)
- [Web applications / APIs protecting resources and built using Azure App Services](#)
- [Web applications / APIs protecting resources using .NET OWIN OpenID Connect, WS-Fed or WindowsAzureActiveDirectoryBearerAuthentication middleware](#)
- [Web applications / APIs protecting resources using .NET Core OpenID Connect or JwtBearerAuthentication middleware](#)
- [Web applications / APIs protecting resources using Node.js passport-azure-ad module](#)
- [Web applications / APIs protecting resources and created with Visual Studio 2015 or later](#)
- [Web applications protecting resources and created with Visual Studio 2013](#)
- [Web APIs protecting resources and created with Visual Studio 2013](#)

- Web applications protecting resources and created with Visual Studio 2012
- Web applications protecting resources and created with Visual Studio 2010, 2008 or using Windows Identity Foundation
- Web applications / APIs protecting resources using any other libraries or manually implementing any of the supported protocols

This guidance is **not** applicable for:

- Applications added from Azure AD Application Gallery (including Custom) have separate guidance with regards to signing keys. [More information](#).
- On-premises applications published via application proxy don't have to worry about signing keys.

Native client applications accessing resources

Applications that are only accessing resources (i.e Microsoft Graph, KeyVault, Outlook API, and other Microsoft APIs) generally only obtain a token and pass it along to the resource owner. Given that they are not protecting any resources, they do not inspect the token and therefore do not need to ensure it is properly signed.

Native client applications, whether desktop or mobile, fall into this category and are thus not impacted by the rollover.

Web applications / APIs accessing resources

Applications that are only accessing resources (i.e Microsoft Graph, KeyVault, Outlook API, and other Microsoft APIs) generally only obtain a token and pass it along to the resource owner. Given that they are not protecting any resources, they do not inspect the token and therefore do not need to ensure it is properly signed.

Web applications and web APIs that are using the app-only flow (client credentials / client certificate), fall into this category and are thus not impacted by the rollover.

Web applications / APIs protecting resources and built using Azure App Services

Azure App Services' Authentication / Authorization (EasyAuth) functionality already has the necessary logic to handle key rollover automatically.

Web applications / APIs protecting resources using .NET OWIN OpenID Connect, WS-Fed or WindowsAzureActiveDirectoryBearerAuthentication middleware

If your application is using the .NET OWIN OpenID Connect, WS-Fed or WindowsAzureActiveDirectoryBearerAuthentication middleware, it already has the necessary logic to handle key rollover automatically.

You can confirm that your application is using any of these by looking for any of the following snippets in your application's Startup.cs or Startup.Auth.cs

```
app.UseOpenIdConnectAuthentication(
    new OpenIdConnectAuthenticationOptions
    {
        // ...
    });

```

```
app.UseWsFederationAuthentication(
    new WsFederationAuthenticationOptions
    {
        // ...
    });

```

```
app.UseWindowsAzureActiveDirectoryBearerAuthentication(
    new WindowsAzureActiveDirectoryBearerAuthenticationOptions
    {
        // ...
    });

```

Web applications / APIs protecting resources using .NET Core OpenID Connect or JwtBearerAuthentication middleware

If your application is using the .NET Core OWIN OpenID Connect or JwtBearerAuthentication middleware, it already has the necessary logic to handle key rollover automatically.

You can confirm that your application is using any of these by looking for any of the following snippets in your application's Startup.cs or Startup.Auth.cs

```
app.UseOpenIdConnectAuthentication(
    new OpenIdConnectAuthenticationOptions
    {
        // ...
    });

```

```
app.UseJwtBearerAuthentication(
    new JwtBearerAuthenticationOptions
    {
        // ...
    });

```

Web applications / APIs protecting resources using Node.js passport-azure-ad module

If your application is using the Nodejs passport-ad module, it already has the necessary logic to handle key rollover automatically.

You can confirm that your application passport-ad by searching for the following snippet in your application's app.js

```
var OIDCStrategy = require('passport-azure-ad').OIDCStrategy;

passport.use(new OIDCStrategy({
    //...
}));

```

Web applications / APIs protecting resources and created with Visual Studio 2015 or later

If your application was built using a web application template in Visual Studio 2015 or later and you selected **Work Or School Accounts** from the **Change Authentication** menu, it already has the necessary logic to handle key rollover automatically. This logic, embedded in the OWIN OpenID Connect middleware, retrieves and caches the keys from the OpenID Connect discovery document and periodically refreshes them.

If you added authentication to your solution manually, your application might not have the necessary key rollover logic. You will need to write it yourself, or follow the steps in [Web applications / APIs using any other libraries or manually implementing any of the supported protocols](#).

Web applications protecting resources and created with Visual Studio 2013

If your application was built using a web application template in Visual Studio 2013 and you selected **Organizational Accounts** from the **Change Authentication** menu, it already has the necessary logic to handle key rollover automatically. This logic stores your organization's unique identifier and the signing key information in two database tables associated with the project. You can find the connection string for the database in the

project's Web.config file.

If you added authentication to your solution manually, your application might not have the necessary key rollover logic. You will need to write it yourself, or follow the steps in [Web applications / APIs using any other libraries or manually implementing any of the supported protocols..](#)

The following steps will help you verify that the logic is working properly in your application.

1. In Visual Studio 2013, open the solution, and then click on the **Server Explorer** tab on the right window.
2. Expand **Data Connections**, **DefaultConnection**, and then **Tables**. Locate the **IssuingAuthorityKeys** table, right-click it, and then click **Show Table Data**.
3. In the **IssuingAuthorityKeys** table, there will be at least one row, which corresponds to the thumbprint value for the key. Delete any rows in the table.
4. Right-click the **Tenants** table, and then click **Show Table Data**.
5. In the **Tenants** table, there will be at least one row, which corresponds to a unique directory tenant identifier. Delete any rows in the table. If you don't delete the rows in both the **Tenants** table and **IssuingAuthorityKeys** table, you will get an error at runtime.
6. Build and run the application. After you have logged in to your account, you can stop the application.
7. Return to the **Server Explorer** and look at the values in the **IssuingAuthorityKeys** and **Tenants** table. You'll notice that they have been automatically repopulated with the appropriate information from the federation metadata document.

Web APIs protecting resources and created with Visual Studio 2013

If you created a web API application in Visual Studio 2013 using the Web API template, and then selected **Organizational Accounts** from the **Change Authentication** menu, you already have the necessary logic in your application.

If you manually configured authentication, follow the instructions below to learn how to configure your Web API to automatically update its key information.

The following code snippet demonstrates how to get the latest keys from the federation metadata document, and then use the [JWT Token Handler](#) to validate the token. The code snippet assumes that you will use your own caching mechanism for persisting the key to validate future tokens from Azure AD, whether it be in a database, configuration file, or elsewhere.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IdentityModel.Tokens;
using System.Configuration;
using System.Security.Cryptography.X509Certificates;
using System.Xml;
using System.IdentityModel.Metadata;
using System.ServiceModel.Security;
using System.Threading;

namespace JWTValidation
{
    public class JWTValidator
    {
        private string MetadataAddress = "[Your Federation Metadata document address goes here]";

        // Validates the JWT Token that's part of the Authorization header in an HTTP request.
        public void ValidateJwtToken(string token)
        {
            JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler()
            {
                // Do not disable for production code
            }
        }
    }
}
```


Web applications protecting resources and created with Visual Studio 2012

If your application was built in Visual Studio 2012, you probably used the Identity and Access Tool to configure your application. It's also likely that you are using the [Validating Issuer Name Registry \(VINR\)](#). The VINR is responsible for maintaining information about trusted identity providers (Azure AD) and the keys used to validate tokens issued by them. The VINR also makes it easy to automatically update the key information stored in a Web.config file by downloading the latest federation metadata document associated with your directory, checking if the configuration is out of date with the latest document, and updating the application to use the new key as necessary.

If you created your application using any of the code samples or walkthrough documentation provided by Microsoft, the key rollover logic is already included in your project. You will notice that the code below already exists in your project. If your application does not already have this logic, follow the steps below to add it and to verify that it's working correctly.

1. In **Solution Explorer**, add a reference to the **System.IdentityModel** assembly for the appropriate project.
2. Open the **Global.asax.cs** file and add the following using directives:

```
using System.Configuration;
using System.IdentityModel.Tokens;
```

3. Add the following method to the **Global.asax.cs** file:

```
protected void RefreshValidationSettings()
{
    string configPath = AppDomain.CurrentDomain.BaseDirectory + "\\\" + "Web.config";
    string metadataAddress =
        ConfigurationManager.AppSettings["ida:FederationMetadataLocation"];
    ValidatingIssuerNameRegistry.WriteToConfig(metadataAddress, configPath);
}
```

4. Invoke the **RefreshValidationSettings()** method in the **Application_Start()** method in **Global.asax.cs** as shown:

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    ...
    RefreshValidationSettings();
}
```

Once you have followed these steps, your application's Web.config will be updated with the latest information from the federation metadata document, including the latest keys. This update will occur every time your application pool recycles in IIS; by default IIS is set to recycle applications every 29 hours.

Follow the steps below to verify that the key rollover logic is working.

1. After you have verified that your application is using the code above, open the **Web.config** file and navigate to the **<issuerNameRegistry>** block, specifically looking for the following few lines:

```
<issuerNameRegistry type="System.IdentityModel.Tokens.ValidatingIssuerNameRegistry,
System.IdentityModel.Tokens.ValidatingIssuerNameRegistry">
    <authority name="https://sts.windows.net/ec4187af-07da-4f01-b18f-64c2f5abceca/">
        <keys>
            <add thumbprint="3A38FA984E8560F19AADC9F86FE9594BB6AD049B" />
        </keys>
    </authority>
</issuerNameRegistry>
```

2. In the <add thumbprint=""> setting, change the thumbprint value by replacing any character with a different one. Save the **Web.config** file.
3. Build the application, and then run it. If you can complete the sign-in process, your application is successfully updating the key by downloading the required information from your directory's federation metadata document. If you are having issues signing in, ensure the changes in your application are correct by reading the [Adding Sign-On to Your Web Application Using Azure AD](#) article, or downloading and inspecting the following code sample: [Multi-Tenant Cloud Application for Azure Active Directory](#).

Web applications protecting resources and created with Visual Studio 2008 or 2010 and Windows Identity Foundation (WIF) v1.0 for .NET 3.5

If you built an application on WIF v1.0, there is no provided mechanism to automatically refresh your application's configuration to use a new key.

- *Easiest way* Use the FedUtil tooling included in the WIF SDK, which can retrieve the latest metadata document and update your configuration.
- Update your application to .NET 4.5, which includes the newest version of WIF located in the System namespace. You can then use the [Validating Issuer Name Registry \(VINR\)](#) to perform automatic updates of the application's configuration.
- Perform a manual rollover as per the instructions at the end of this guidance document.

Instructions to use the FedUtil to update your configuration:

1. Verify that you have the WIF v1.0 SDK installed on your development machine for Visual Studio 2008 or 2010. You can [download it from here](#) if you have not yet installed it.
2. In Visual Studio, open the solution, and then right-click the applicable project and select **Update federation metadata**. If this option is not available, FedUtil and/or the WIF v1.0 SDK has not been installed.
3. From the prompt, select **Update** to begin updating your federation metadata. If you have access to the server environment where the application is hosted, you can optionally use FedUtil's [automatic metadata update scheduler](#).
4. Click **Finish** to complete the update process.

Web applications / APIs protecting resources using any other libraries or manually implementing any of the supported protocols

If you are using some other library or manually implemented any of the supported protocols, you'll need to review the library or your implementation to ensure that the key is being retrieved from either the OpenID Connect discovery document or the federation metadata document. One way to check for this is to do a search in your code or the library's code for any calls out to either the OpenID discovery document or the federation metadata document.

If the key is being stored somewhere or hardcoded in your application, you can manually retrieve the key and update it accordingly by performing a manual rollover as per the instructions at the end of this guidance document. **It is strongly encouraged that you enhance your application to support automatic rollover** using any of the approaches outline in this article to avoid future disruptions and overhead if Azure AD increases its rollover cadence or has an emergency out-of-band rollover.

How to test your application to determine if it will be affected

You can validate whether your application supports automatic key rollover by downloading the scripts and following the instructions in [this GitHub repository](#).

How to perform a manual rollover if your application does not support automatic rollover

If your application does **not** support automatic rollover, you will need to establish a process that periodically

monitors Azure AD's signing keys and performs a manual rollover accordingly. [This GitHub repository](#) contains scripts and instructions on how to do this.

Microsoft identity platform ID tokens

10/17/2019 • 7 minutes to read • [Edit Online](#)

`id_tokens` are sent to the client application as part of an [OpenID Connect](#) flow. They can be sent along side or instead of an access token, and are used by the client to authenticate the user.

Using the id_token

ID Tokens should be used to validate that a user is who they claim to be and get additional useful information about them - it shouldn't be used for authorization in place of an [access token](#). The claims it provides can be used for UX inside your application, as keys in a database, and providing access to the client application. When creating keys for a database, `idp` should not be used because it messes up guest scenarios. Keying should be done on `sub` alone (which is always unique), with `tid` used for routing if need be. If you need to share data across services, `oid + sub + tid` will work since multiple services all get the same `oid`.

Claims in an id_token

`id_tokens` for a Microsoft identity are [JWTs](#), meaning they consist of a header, payload, and signature portion. You can use the header and signature to verify the authenticity of the token, while the payload contains the information about the user requested by your client. Except where noted, all claims listed here appear in both v1.0 and v2.0 tokens.

v1.0

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6IjdfWnVmMXR2a3dMeF1hSFMzcTZsVWpVWUlHdyIsImtpZCI6IjdfWnVmM
XR2a3dMeF1hSFMzcTZsVWpVWUlHdyJ9.eyJhdWQiOiJiMTRhNzUwNS05NmU5LTQ5MjctOTF1OC0wNjAxZDBmYz1jYWElLCJpc3MiO
iJodHRwczovL3N0cy53aW5kb3dzLm51dC9mYTE1ZDY5Mi1l0WM3LTQ0NjAtYTc0My0yOWYyOTU2ZmQ0MjkvIiwiawF0IjoxNTM2Mj
c1MTi0LCJuYmYi0jE1MzYyNzUxMjQsImV4cCI6MTUzNjI3OTAyNCwiYwlvIjoiQVhRQWkvOE1BQUFBcXhzdUIrUjREMpJGUxFPRVR
PNFlkWGJMRD1rWjh4Z1hhZGVBTBRMk5rTlQ1aXpmZzN1d2JXU1hodVNTajZVVDVoeTJENldxQXBCNwpLQTzaZ1o5ay9TVTI3dVY5
Y2V0WGZMT3RwTrnR0Z2s1RGNCdGsrTExzdhovSmcrZ11sbXY5Y1VVFhscGhUYzZDODZkbWoxRkn3PT0iLCJhbXIiOlsicnNhIl0sI
mVtYWlsIjoiYWJ1bG1Abwljcm9zb2Z0LmNvbSIsImZhbWlseV9uYW11IjoiTGluY29sbiIsImdpdmVuX25hbWUiOijBYmUiLCJpZH
AiOjodHRwczovL3N0cy53aW5kb3dzLm51dC83MmY50DhiZi04NmYxLTQxYWYtOTFhYi0yZDdjZDAxMWR1NDcvIiwiiaXBhZGRyIjo
iMTMxljEwNy4yMjiuMjIiLCJuYW11IjoiYWJ1bGk1LCJub25jZSI6IjEyMzUyMyIsIm9pZCI6IjA10DMzYjZilWFhMWQtNDJkNC05
ZWMyLTFiMmJiOTE5NDQzOCIsInJoIjoiSSIIsInN1YiI6IjVfsjlyU3NzOC1qdnRFswN1NnV1Uk5MOHhYYjhMRjRGc2dfS29vQzJSS
1EiLCJ0aWQoIjMYTE1ZDY5Mi1l0WM3LTQ0NjAtYTc0My0yOWYyOTU2ZmQ0Mjk1LCJ1bm1xdWVfbmFtZSI6IkFiZUxpQG1pY3Jvc2
9mdC5jb20iLCJ1dGkiOijMeGVFNDZhcvRrT3BHU2ZUbG40RUFBiwidmVYIjoiMS4wIn0=.UJQrCA6qn2bXq57qzGX_-
D3HcPHqBMOKDPx4su1yKRLNErVD8xkxJLNlVRdASHqEcpvDctbdHccu6DPpkq5f0ibcaQFhejQNcABidJCTz0Bb2AbdUCTqAzdt9p
dgQvMBnVH1xk3SCM6d4BbT4BkLLj10ZLasX7vRknaSjE_C5DI7Fg4WrZPwOhII1dB0HEZ_qpNaYXEiy-
o94UJ94zCr07GgrqMsFYQqFR7kn-mm68AjvLcgwSFZvyR_yIK75S_K37vC3QryQ7cNoafDe9upql_6pB2ybMVlgWPs_DmbJ8g0om-
sPlwyn74Cc1tw3ze-Xptw_2uVdPgWyqfuWAfq6Q
```

View this v1.0 sample token in [jwt.ms](#).

v2.0

```

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzIiNiIsImtpZCI6IjFMVE16YWtpaG1SbGFf0HoyQkVKVlh1V01xbyJ9.eyJ2ZXIiOiIyLjAi
LCJpc3MiOiJodHRwczovL2xvZ2luLm1pY3Jvc29mdG9ubGluzS5jb20vOTEyMjA0MQQtNm2Ny00YzViLWIxMTItMzZhMza0YjY2Z
GFkl3YyLjAiLCJzdWIiOiJBQUFBQUFBQUFBQUFBQUFBQUFJa3pxR1ZyU2FTYUZieTc4MmjIdGFRIiwiYXvkIjoiNmNiMDQwMT
gtYTNmNS00NmE3LWI50TUtOTQwYzc4ZjVhZWYzIiwiZXhwIjoxNTM2MzYxNDExLCJpYXQiOjE1MzYyNzQ3MTEsIm5iZiI6MTUzNjI
3NDcxMSwibmFtZSI6IkFizSBMaw5jb2xuIiwiChJlZmVycmVkX3VzZXJuYW1IjoiQWJ1TG1AbWljcm9zb2Z0LmNvbSISIm9pZCI6
IjAwMDAwMDAwLTawMDAwMC02NmYzLTmZmJ1Y2E3ZWE4MSIsInRpZCI6IjkxMjIwNDBkLTZjnJctNGM1Yi1iMTEyLTm2YTmWn
GI2NmRhZCIsIm5vbmn1IjoiMTIzNTIzIiwiYwlvIjoiRGYyVVZYTDFpeCFsTUNXTVNPSkJjRmF0emNHZnZGR2hqS3Y4cTVnMHg3Mz
JkUjVNQjVCaXN2R1FPN11XQnlqZDhpUURMcSF1R2JRGFexA1bw5PcmNkcUh1lwNuBhR1cfftUnA2QU1aOGpZIn0.1AFWW-
Ck5nRowS11tm7GzzvDwUkqvhsQpm55TQsmVo9Y59cLhRXpvB8n-
55HCr9Z6G_31_UbeUkoz612I2j_Sm9FFShSDDjoaLqr54CreGIJvjtmS3EkK9a7SJbbcpL1MpUtlfygow39tFjY7EVNW9p1wUvRrT
gVkj7YLprvfw-CIqw3gHC-T7IK_m_xkr08INERBtaecwhTeN4chPC4W3jdmw_1Ixzc48YoQ0dB1L-
ImX98Egypfrlbm0IBL5spFzL6JDZIRRJou8vecJvj1mq-IUhGt0Macxx8jdxYLP-
KUU2d9MbNKpCKJuZ7p8gwTL5B7N1Udh_dmSviPWrw

```

View this v2.0 sample token in [jwt.ms](#).

Header claims

CLAIM	FORMAT	DESCRIPTION
<code>typ</code>	String - always "JWT"	Indicates that the token is a JWT.
<code>alg</code>	String	Indicates the algorithm that was used to sign the token. Example: "RS256"
<code>kid</code>	String	Thumbprint for the public key used to sign this token. Emitted in both v1.0 and v2.0 <code>id_tokens</code> .
<code>x5t</code>	String	The same (in use and value) as <code>kid</code> . However, this is a legacy claim emitted only in v1.0 <code>id_tokens</code> for compatibility purposes.

Payload claims

This list shows the claims that are in most `id_tokens` by default (except where noted). However, your app can use [optional claims](#) to request additional claims in the `id_token`. These can range from the `groups` claim to information about the user's name.

CLAIM	FORMAT	DESCRIPTION
<code>aud</code>	String, an App ID URI	Identifies the intended recipient of the token. In <code>id_tokens</code> , the audience is your app's Application ID, assigned to your app in the Azure portal. Your app should validate this value, and reject the token if the value does not match.

CLAIM	FORMAT	DESCRIPTION
<code>iss</code>	String, an STS URI	<p>Identifies the security token service (STS) that constructs and returns the token, and the Azure AD tenant in which the user was authenticated. If the token was issued by the v2.0 endpoint, the URI will end in <code>/v2.0</code>. The GUID that indicates that the user is a consumer user from a Microsoft account is <code>9188040d-6c67-4c5b-b112-36a304b66dad</code>. Your app should use the GUID portion of the claim to restrict the set of tenants that can sign in to the app, if applicable.</p>
<code>iat</code>	int, a UNIX timestamp	"Issued At" indicates when the authentication for this token occurred.
<code>idp</code>	String, usually an STS URI	<p>Records the identity provider that authenticated the subject of the token. This value is identical to the value of the Issuer claim unless the user account not in the same tenant as the issuer - guests, for instance. If the claim isn't present, it means that the value of <code>iss</code> can be used instead. For personal accounts being used in an organizational context (for instance, a personal account invited to an Azure AD tenant), the <code>idp</code> claim may be 'live.com' or an STS URI containing the Microsoft account tenant <code>9188040d-6c67-4c5b-b112-36a304b66dad</code>.</p>
<code>nbf</code>	int, a UNIX timestamp	The "nbf" (not before) claim identifies the time before which the JWT MUST NOT be accepted for processing.
<code>exp</code>	int, a UNIX timestamp	The "exp" (expiration time) claim identifies the expiration time on or after which the JWT MUST NOT be accepted for processing. It's important to note that a resource may reject the token before this time as well - if, for example, a change in authentication is required or a token revocation has been detected.

CLAIM	FORMAT	DESCRIPTION
c_hash	String	The code hash is included in ID tokens only when the ID token is issued with an OAuth 2.0 authorization code. It can be used to validate the authenticity of an authorization code. For details about performing this validation, see the OpenID Connect specification .
at_hash	String	The access token hash is included in ID tokens only when the ID token is issued with an OAuth 2.0 access token. It can be used to validate the authenticity of an access token. For details about performing this validation, see the OpenID Connect specification .
aio	Opaque String	An internal claim used by Azure AD to record data for token reuse. Should be ignored.
preferred_username	String	The primary username that represents the user. It could be an email address, phone number, or a generic username without a specified format. Its value is mutable and might change over time. Since it is mutable, this value must not be used to make authorization decisions. The <code>profile</code> scope is required to receive this claim.
email	String	The <code>email</code> claim is present by default for guest accounts that have an email address. Your app can request the email claim for managed users (those from the same tenant as the resource) using the email optional claim . On the v2.0 endpoint, your app can also request the <code>email</code> OpenID Connect scope - you don't need to request both the optional claim and the scope to get the claim. The email claim only supports addressable mail from the user's profile information.
name	String	The <code>name</code> claim provides a human-readable value that identifies the subject of the token. The value isn't guaranteed to be unique, it is mutable, and it's designed to be used only for display purposes. The <code>profile</code> scope is required to receive this claim.

CLAIM	FORMAT	DESCRIPTION
<code>nonce</code>	String	The nonce matches the parameter included in the original /authorize request to the IDP. If it does not match, your application should reject the token.
<code>oid</code>	String, a GUID	The immutable identifier for an object in the Microsoft identity system, in this case, a user account. This ID uniquely identifies the user across applications - two different applications signing in the same user will receive the same value in the <code>oid</code> claim. The Microsoft Graph will return this ID as the <code>id</code> property for a given user account. Because the <code>oid</code> allows multiple apps to correlate users, the <code>profile</code> scope is required to receive this claim. Note that if a single user exists in multiple tenants, the user will contain a different object ID in each tenant - they're considered different accounts, even though the user logs into each account with the same credentials. The <code>oid</code> claim is a GUID and cannot be reused.
<code>roles</code>	Array of strings	The set of roles that were assigned to the user who is logging in.
<code>rh</code>	Opaque String	An internal claim used by Azure to revalidate tokens. Should be ignored.
<code>sub</code>	String, a GUID	The principal about which the token asserts information, such as the user of an app. This value is immutable and cannot be reassigned or reused. The subject is a pairwise identifier - it is unique to a particular application ID. If a single user signs into two different apps using two different client IDs, those apps will receive two different values for the subject claim. This may or may not be wanted depending on your architecture and privacy requirements.

CLAIM	FORMAT	DESCRIPTION
<code>tid</code>	String, a GUID	A GUID that represents the Azure AD tenant that the user is from. For work and school accounts, the GUID is the immutable tenant ID of the organization that the user belongs to. For personal accounts, the value is 9188040d-6c67-4c5b-b112-36a304b66dad . The <code>profile</code> scope is required to receive this claim.
<code>unique_name</code>	String	Provides a human readable value that identifies the subject of the token. This value isn't guaranteed to be unique within a tenant and should be used only for display purposes. Only issued in v1.0 <code>id_tokens</code> .
<code>uti</code>	Opaque String	An internal claim used by Azure to revalidate tokens. Should be ignored.
<code>ver</code>	String, either 1.0 or 2.0	Indicates the version of the <code>id_token</code> .

Validating an id_token

Validating an `id_token` is similar to the first step of [validating an access token](#) - your client should validate that the correct issuer has sent back the token and that it hasn't been tampered with. Because `id_tokens` are always a JWT, many libraries exist to validate these tokens - we recommend you use one of these rather than doing it yourself.

To manually validate the token, see the steps details in [validating an access token](#). After validating the signature on the token, the following claims should be validated in the `id_token` (these may also be done by your token validation library):

- Timestamps: the `iat`, `nbf`, and `exp` timestamps should all fall before or after the current time, as appropriate.
- Audience: the `aud` claim should match the app ID for your application.
- Nonce: the `nonce` claim in the payload must match the nonce parameter passed into the `/authorize` endpoint during the initial request.

Next steps

- Learn about [access tokens](#)
- Customize the claims in your `id_token` using [optional claims](#).

Microsoft identity platform access tokens

11/12/2019 • 20 minutes to read • [Edit Online](#)

Access tokens enable clients to securely call APIs protected by Azure. Microsoft identity platform access tokens are **JWTs**, Base64 encoded JSON objects signed by Azure. Clients should treat access tokens as opaque strings, as the contents of the token are intended for the resource only. For validation and debugging purposes, developers can decode JWTs using a site like jwt.ms. Your client can get an access token from either the v1.0 endpoint or the v2.0 endpoint using a variety of protocols.

When your client requests an access token, Azure AD also returns some metadata about the access token for your app's consumption. This information includes the expiry time of the access token and the scopes for which it's valid. This data allows your app to do intelligent caching of access tokens without having to parse the access token itself.

If your application is a resource (web API) that clients can request access to, access tokens provide helpful information for use in authentication and authorization, such as the user, client, issuer, permissions, and more.

See the following sections to learn how a resource can validate and use the claims inside an access token.

IMPORTANT

Access tokens are created based on the *audience* of the token, meaning the application that owns the scopes in the token. This is how a resource setting `accessTokenAcceptedVersion` in the app manifest to `2` allows a client calling the v1.0 endpoint to receive a v2.0 access token. Similarly, this is why changing the access token optional claims for your client do not change the access token received when a token is requested for `user.read`, which is owned by the MS Graph resource.

For the same reason, while testing your client application with a personal account (such as hotmail.com or outlook.com), you may find that the access token received by your client is an opaque string. This is because the resource being accessed has requested legacy MSA (Microsoft account) tickets that are encrypted and can't be understood by the client.

Sample tokens

v1.0 and v2.0 tokens look similar and contain many of the same claims. An example of each is provided here.

v1.0

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Imk2bEdrM0ZaenhsY1ViMkMzbkVRN3N5SEpsWSISImtpZCI6Imk2bEdrM0ZaenhsY1ViMkMzbkVRN3N5SEpsWSJ9.eyJhdWQiOiIjJzFkYTlkNC1mZjc3LTrjM2UtYTawNs04NDbjM2Y4MzA3NDUiLCjpc3Mi0iJodHrwczovL3N0cy53aW5kb3dzLm51dC9mYTE1ZDY5Mi11i0WM3LTQ0NjAtYTC0My0yOWYtOTUyMjIyOS8iLCJpYXqi0jE1MzczyMzMxMDYsIm5i1i6MTUzNzIzMzEwNiwiXhwIjoxNTM3MjM3MDA2LCJhY3i0iIxIiwiYlwIjoiQvhRQWkvOE1BLQUBFBRm0rRS9RVFcrz0uVnhMaLdkdzhLkzYxQudy091TU1GnVmViYU1q1n1hP0lubQzKzdtk95RCT0d1p5R24yVmFUL2teS1h3NE1JahJnR1zXnkJu0HdMwg9UMuxrSvorRnpRvMtKUFBUmSwNETjWHTBtE6NUERTL0Rpq0PrnRTiyM1rb1u20v5hR1u1hVu9uc01hd1rrRpt0i1CJhbX1i01s1d2h1l0s0mfwgc1kjioiNzVkyMn32YtMTBhMy0002TzSLtg1ZmQtOGMxMj1c1NDRmtDj1yIiYXBwaWRhY3i0iIwIiwiZw1hawI0iJyBmVmAuBtaWlyNb3znQuY29tIiwiZmFtaWx5z5hbwI0i1jMw5j2bu2xiUiw2l2zZw5fbmTzSi6Kf1zsAoTVNGVCKiLcJpZhAoi1odHrwczovL3N0cy53aW5kb3dzLm51dC93SmM50Dh1z04NmxLyTQxWyt0FhYi0zDjjZDxmMjIyNdcviIiwiXbhZGrYijoiMjIyLjIyM4mJyMjIiCJyUw11joiYwJbGk1CjwaiQoIiWmJjyM2I2Y1hYTfkLtzQdzt0WvjMc0xYjyJyKxj0TQ0Mzgj1CjyaC16IkkiLCjzY3ai0j1c2VxY2t3GVy29uYXRpB24iLCjzdW1i0ijsM19yb1tUVJyMj1jVUXTO1pmSwlhBxcE9pTx01sdNaQuNvMud1WeEiLCj0aWq1o1jMyte1ZDY5Mi11i0WM3LTQ0NjAtYTC0My0yOWYtOTU2ZmQ0Mj1kLCj1bm1xdwVfbmftZS16MifzWxpQG1pY3Jvc29mdC5jb20iLCJ1dGk1o1jGvnHFeF1YSTMwLvr1awltdVvRkFbiwiidmVjyjoiMS4wIn0.D3H6pMuTQnoJA
Gq6AHd

View this v1.0 token in [JWT.ms](#)

v2.0

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Imk2bEdrM0ZaenhsY1ViMkMzbkVRN3N5SePsWSJ9.eyJhdWQiOiIzTc0MTcyIi1zTU2LTQ4NDMtOWZmNC11NjZhMzliYjEyzTmLcJpc3Mi0iJodHrwczovL2xvZ2ulm1pY3Jvc29mdG9ubGlzS5j2b20vNzJm0TqGyM0tODZmMS00MWFmLTkxYwItMmQ3Y2QwMTFkYjQ3L3YljlAiLCjPXYQiOjE1MzcMzEwNDgsIm5iZi16MuZnZiMta0CwiZhwIjoxNTM3MjM00TQ4LCjhaw8i0jBWFfBaS48UFBUQF0QWfTaG8zQ2hNaWY2S09udHSRqjd1QnE0L0RjrY1F6amNRK3hQXkvQzNqRGF0R3Y2DZ3T1kVkdSz2h0Um53jsFT2Nbkk5aY2p2a295ckZ4Q3R0dJmzTqUmlvT0ZKNGDQd0ldw9DwYcxdu9U0VdIyMj1yZoh3TFBZUS91Zjc5UngrMetjAwpkcm1Wnj15Y3R66VE9PS1sImF6C6G1jZ1NzQxjZ1lWJN1TYtNdg0My05ZmY0lWU2NmzezW0j1Mj1My1sImF6C6Gfjci16i1jai1C3uW11joiQWj1lExpbnVbg4iLCjvawQoIi020TAyMj1zS1m1zFhlTRkNTYtWYjkMs03ZTmRn2Qz0GU0NzQlCicjWcmVzXjyZhdFrxN1m5hbw10iJyhVmUsaBtaWnbY3NzNzQyU29tIwiicmgio1j1iwi2CnWjoiYnjNzXnZx2F1zL1Cj2dWio1jz1sPwMfieZhdGZPvBz3VbzG1anJLJUtm1ZrtMj1yWdUy1cioReXzRtkriwidGlikj1oInzJm0Tg4Ym0tODZmMS00MWFmLTkxYwItMm03Y2QwMTFkYj03Ii1wdxRbIi0jz1fP0nFYTFBqMGVRYTevUy1JWZBOS1sInzLc1i6j1uMCJ9.pi4N-w-3u59drBLbfCt

View this v2.0 token in [JWT.ms](#).

Claims in access tokens

JWTs are split into three pieces:

- **Header** - Provides information about how to [validate the token](#) including information about the type of token and how it was signed.
 - **Payload** - Contains all of the important data about the user or app that is attempting to call your service.
 - **Signature** - Is the raw material used to validate the token.

Each piece is separated by a period (.) and separately Base64 encoded.

Claims are present only if a value exists to fill it. So, your app shouldn't take a dependency on a claim being present. Examples include `pwd_exp` (not every tenant requires passwords to expire) or `family_name` (client credential flows are on behalf of applications, which don't have names). Claims used for access token validation will always be present.

NOTE

Some claims are used to help Azure AD secure tokens in case of reuse. These are marked as not being for public consumption in the description as "Opaque". These claims may or may not appear in a token, and new ones may be added without notice.

Header claims

CLAIM	FORMAT	DESCRIPTION
<code>typ</code>	String - always "JWT"	Indicates that the token is a JWT.
<code>nonce</code>	String	A unique identifier used to protect against token replay attacks. Your resource can record this value to protect against replays.
<code>alg</code>	String	Indicates the algorithm that was used to sign the token, for example, "RS256"
<code>kid</code>	String	Specifies the thumbprint for the public key that's used to sign this token. Emitted in both v1.0 and v2.0 access tokens.
<code>x5t</code>	String	Functions the same (in use and value) as <code>kid</code> . <code>x5t</code> is a legacy claim emitted only in v1.0 access tokens for compatibility purposes.

Payload claims

CLAIM	FORMAT	DESCRIPTION
<code>aud</code>	String, an App ID URI	Identifies the intended recipient of the token. In id tokens, the audience is your app's Application ID, assigned to your app in the Azure portal. Your app should validate this value and reject the token if the value does not match.
<code>iss</code>	String, an STS URI	Identifies the security token service (STS) that constructs and returns the token, and the Azure AD tenant in which the user was authenticated. If the token issued is a v2.0 token (see the <code>ver</code> claim), the URI will end in <code>/v2.0</code> . The GUID that indicates that the user is a consumer user from a Microsoft account is <code>9188040d-6c67-4c5b-b112-36a304b66dad</code> . Your app should use the GUID portion of the claim to restrict the set of tenants that can sign in to the app, if applicable.
<code>idp</code>	String, usually an STS URI	Records the identity provider that authenticated the subject of the token. This value is identical to the value of the Issuer claim unless the user account not in the same tenant as the issuer - guests, for instance. If the claim isn't present, it means that the value of <code>iss</code> can be used instead. For personal accounts being used in an organizational context (for instance, a personal account invited to an Azure AD tenant), the <code>idp</code> claim may be 'live.com' or an STS URI containing the Microsoft account tenant <code>9188040d-6c67-4c5b-b112-36a304b66dad</code> .
<code>iat</code>	int, a UNIX timestamp	"Issued At" indicates when the authentication for this token occurred.
<code>nbf</code>	int, a UNIX timestamp	The "nbf" (not before) claim identifies the time before which the JWT must not be accepted for processing.

CLAIM	FORMAT	DESCRIPTION
<code>exp</code>	int, a UNIX timestamp	The "exp" (expiration time) claim identifies the expiration time on or after which the JWT must not be accepted for processing. It's important to note that a resource may reject the token before this time as well, such as when a change in authentication is required or a token revocation has been detected.
<code>aio</code>	Opaque String	An internal claim used by Azure AD to record data for token reuse. Resources should not use this claim.
<code>acr</code>	String, a "0" or "1"	Only present in v1.0 tokens. The "Authentication context class" claim. A value of "0" indicates the end-user authentication did not meet the requirements of ISO/IEC 29115.
<code>amr</code>	JSON array of strings	Only present in v1.0 tokens. Identifies how the subject of the token was authenticated. See the amr claim section for more details.
<code>appid</code>	String, a GUID	Only present in v1.0 tokens. The application ID of the client using the token. The application can act as itself or on behalf of a user. The application ID typically represents an application object, but it can also represent a service principal object in Azure AD.
<code>appidacr</code>	"0", "1", or "2"	Only present in v1.0 tokens. Indicates how the client was authenticated. For a public client, the value is "0". If client ID and client secret are used, the value is "1". If a client certificate was used for authentication, the value is "2".
<code>azp</code>	String, a GUID	Only present in v2.0 tokens, a replacement for <code>appid</code> . The application ID of the client using the token. The application can act as itself or on behalf of a user. The application ID typically represents an application object, but it can also represent a service principal object in Azure AD.
<code>azpacr</code>	"0", "1", or "2"	Only present in v2.0 tokens, a replacement for <code>appidacr</code> . Indicates how the client was authenticated. For a public client, the value is "0". If client ID and client secret are used, the value is "1". If a client certificate was used for authentication, the value is "2".
<code>preferred_username</code>	String	The primary username that represents the user. It could be an email address, phone number, or a generic username without a specified format. Its value is mutable and might change over time. Since it is mutable, this value must not be used to make authorization decisions. It can be used for username hints though. The <code>profile</code> scope is required in order to receive this claim.
<code>name</code>	String	Provides a human-readable value that identifies the subject of the token. The value is not guaranteed to be unique, it is mutable, and it's designed to be used only for display purposes. The <code>profile</code> scope is required in order to receive this claim.
<code>scp</code>	String, a space separated list of scopes	The set of scopes exposed by your application for which the client application has requested (and received) consent. Your app should verify that these scopes are valid ones exposed by your app, and make authorization decisions based on the value of these scopes. Only included for user tokens .

CLAIM	FORMAT	DESCRIPTION
<code>roles</code>	Array of strings, a list of permissions	The set of permissions exposed by your application that the requesting application or user has been given permission to call. For application tokens , this is used during the client-credentials flow in place of user scopes. For user tokens this is populated with the roles the user was assigned to on the target application.
<code>wids</code>	Array of RoleTemplateID GUIDs	Denotes the tenant-wide roles assigned to this user, from the section of roles present in the admin roles page . This claim is configured on a per-application basis, through the <code>groupMembershipClaims</code> property of the application manifest . Setting it to "All" or "DirectoryRole" is required. May not be present in tokens obtained through the implicit flow due to token length concerns.
<code>groups</code>	JSON array of GUIDs	<p>Provides object IDs that represent the subject's group memberships. These values are unique (see Object ID) and can be safely used for managing access, such as enforcing authorization to access a resource. The groups included in the groups claim are configured on a per-application basis, through the <code>groupMembershipClaims</code> property of the application manifest. A value of null will exclude all groups, a value of "SecurityGroup" will include only Active Directory Security Group memberships, and a value of "All" will include both Security Groups and Office 365 Distribution Lists.</p> <p>See the <code>hasgroups</code> claim below for details on using the <code>groups</code> claim with the implicit grant. For other flows, if the number of groups the user is in goes over a limit (150 for SAML, 200 for JWT), then an overage claim will be added to the claim sources pointing at the AAD Graph endpoint containing the list of groups for the user.</p>
<code>hasgroups</code>	Boolean	If present, always <code>true</code> , denoting the user is in at least one group. Used in place of the <code>groups</code> claim for JWTs in implicit grant flows if the full groups claim would extend the URI fragment beyond the URL length limits (currently 6 or more groups). Indicates that the client should use the Graph to determine the user's groups (https://graph.windows.net/{tenantID}/users/{userID}/memberOf).
<code>groups:src1</code>	JSON object	<p>For token requests that are not length limited (see <code>hasgroups</code> above) but still too large for the token, a link to the full groups list for the user will be included. For JWTs as a distributed claim, for SAML as a new claim in place of the <code>groups</code> claim.</p> <p>Example JWT Value:</p> <pre>"groups":"src1" "_claim_sources : "src1" : { "endpoint" : "https://graph.windows.net/{tenantID}/users/{userID}/memberOf" }</pre>

CLAIM	FORMAT	DESCRIPTION
<code>sub</code>	String, a GUID	The principal about which the token asserts information, such as the user of an app. This value is immutable and cannot be reassigned or reused. It can be used to perform authorization checks safely, such as when the token is used to access a resource, and can be used as a key in database tables. Because the subject is always present in the tokens that Azure AD issues, we recommend using this value in a general-purpose authorization system. The subject is, however, a pairwise identifier - it is unique to a particular application ID. Therefore, if a single user signs into two different apps using two different client IDs, those apps will receive two different values for the subject claim. This may or may not be desired depending on your architecture and privacy requirements. See also the <code>oid</code> claim (which does remain the same across apps within a tenant).
<code>oid</code>	String, a GUID	The immutable identifier for an object in the Microsoft identity platform, in this case, a user account. It can also be used to perform authorization checks safely and as a key in database tables. This ID uniquely identifies the user across applications - two different applications signing in the same user will receive the same value in the <code>oid</code> claim. Thus, <code>oid</code> can be used when making queries to Microsoft online services, such as the Microsoft Graph. The Microsoft Graph will return this ID as the <code>id</code> property for a given user account . Because the <code>oid</code> allows multiple apps to correlate users, the <code>profile</code> scope is required in order to receive this claim. Note that if a single user exists in multiple tenants, the user will contain a different object ID in each tenant - they are considered different accounts, even though the user logs into each account with the same credentials.
<code>tid</code>	String, a GUID	Represents the Azure AD tenant that the user is from. For work and school accounts, the GUID is the immutable tenant ID of the organization that the user belongs to. For personal accounts, the value is <code>9188040d-6c67-4c5b-b112-36a304b66dad</code> . The <code>profile</code> scope is required in order to receive this claim.
<code>unique_name</code>	String	Only present in v1.0 tokens. Provides a human readable value that identifies the subject of the token. This value is not guaranteed to be unique within a tenant and should be used only for display purposes.
<code>uti</code>	Opaque String	An internal claim used by Azure to revalidate tokens. Resources shouldn't use this claim.
<code>rh</code>	Opaque String	An internal claim used by Azure to revalidate tokens. Resources should not use this claim.
<code>ver</code>	String, either <code>1.0</code> or <code>2.0</code>	Indicates the version of the access token.

NOTE

Groups coverage claim

To ensure that the token size doesn't exceed HTTP header size limits, Azure AD limits the number of object IDs that it includes in the groups claim. If a user is member of more groups than the coverage limit (150 for SAML tokens, 200 for JWT tokens), then Azure AD does not emit the groups claim in the token. Instead, it includes an overage claim in the token that indicates to the application to query the Graph API to retrieve the user's group membership.

```

{
  ...
  "_claim_names": {
    "groups": "src1"
  },
  {
    "_claim_sources": {
      "src1": {
        "endpoint": "[Graph Url to get this user's group membership from]"
      }
    }
  }
  ...
}

```

You can use the `BulkCreateGroups.ps1` provided in the [App Creation Scripts](#) folder to help test overage scenarios.

v1.0 basic claims

The following claims will be included in v1.0 tokens if applicable, but aren't included in v2.0 tokens by default. If you're using v2.0 and need one of these claims, request them using [optional claims](#).

CLAIM	FORMAT	DESCRIPTION
<code>ipaddr</code>	String	The IP address the user authenticated from.
<code>onprem_sid</code>	String, in SID format	In cases where the user has an on-premises authentication, this claim provides their SID. You can use <code>onprem_sid</code> for authorization in legacy applications.
<code>pwd_exp</code>	int, a UNIX timestamp	Indicates when the user's password expires.
<code>pwd_url</code>	String	A URL where users can be sent to reset their password.
<code>in_corp</code>	boolean	Signals if the client is logging in from the corporate network. If they aren't, the claim isn't included.
<code>nickname</code>	String	An additional name for the user, separate from first or last name.
<code>family_name</code>	String	Provides the last name, surname, or family name of the user as defined on the user object.
<code>given_name</code>	String	Provides the first or given name of the user, as set on the user object.
<code>upn</code>	String	The username of the user. May be a phone number, email address, or unformatted string. Should only be used for display purposes and providing username hints in reauthentication scenarios.

The `amr` claim

Microsoft identities can authenticate in different ways, which may be relevant to your application. The `amr` claim is an array that can contain multiple items, such as `["mfa", "rsa", "pwd"]`, for an authentication that used both a password and the Authenticator app.

VALUE	DESCRIPTION
<code>pwd</code>	Password authentication, either a user's Microsoft password or an app's client secret.
<code>rsa</code>	Authentication was based on the proof of an RSA key, for example with the Microsoft Authenticator app . This includes if authentication was done by a self-signed JWT with a service owned X509 certificate.
<code>otp</code>	One-time passcode using an email or a text message.
<code>fed</code>	A federated authentication assertion (such as JWT or SAML) was used.

VALUE	DESCRIPTION
wia	Windows Integrated Authentication
mfa	Multi-factor authentication was used. When this is present the other authentication methods will also be included.
ngcmfa	Equivalent to <code>mfa</code> , used for provisioning of certain advanced credential types.
wiaormfa	The user used Windows or an MFA credential to authenticate.
none	No authentication was done.

Validating tokens

To validate an `id_token` or an `access_token`, your app should validate both the token's signature and the claims. To validate access tokens, your app should also validate the issuer, the audience, and the signing tokens. These need to be validated against the values in the OpenID discovery document. For example, the tenant-independent version of the document is located at <https://login.microsoftonline.com/common/.well-known/openid-configuration>.

The Azure AD middleware has built-in capabilities for validating access tokens, and you can browse through our [samples](#) to find one in the language of your choice. For more information on how to explicitly validate a JWT token, see the [manual JWT validation sample](#).

We provide libraries and code samples that show how to easily handle token validation. The below information is provided for those who wish to understand the underlying process. There are also several third-party open-source libraries available for JWT validation - there is at least one option for almost every platform and language out there. For more information about Azure AD authentication libraries and code samples, see [v1.0 authentication libraries](#) and [v2.0 authentication libraries](#).

Validating the signature

A JWT contains three segments, which are separated by the `.` character. The first segment is known as the **header**, the second as the **body**, and the third as the **signature**. The signature segment can be used to validate the authenticity of the token so that it can be trusted by your app.

Tokens issued by Azure AD are signed using industry standard asymmetric encryption algorithms, such as RSA 256. The header of the JWT contains information about the key and encryption method used to sign the token:

```
{
  "typ": "JWT",
  "alg": "RS256",
  "x5t": "iBjL1Rcqzhiy4fpIXxdZqohM2Yk",
  "kid": "iBjL1Rcqzhiy4fpIXxdZqohM2Yk"
}
```

The `alg` claim indicates the algorithm that was used to sign the token, while the `kid` claim indicates the particular public key that was used to validate the token.

At any given point in time, Azure AD may sign an `id_token` using any one of a certain set of public-private key pairs. Azure AD rotates the possible set of keys on a periodic basis, so your app should be written to handle those key changes automatically. A reasonable frequency to check for updates to the public keys used by Azure AD is every 24 hours.

You can acquire the signing key data necessary to validate the signature by using the [OpenID Connect metadata document](#) located at:

```
https://login.microsoftonline.com/common/v2.0/.well-known/openid-configuration
```

TIP

Try this [URL](#) in a browser!

This metadata document:

- Is a JSON object containing several useful pieces of information, such as the location of the various endpoints required for doing OpenID Connect authentication.
- Includes a `jwks_uri`, which gives the location of the set of public keys used to sign tokens. The JSON Web Key (JWK) located at the `jwks_uri` contains all of the public key information in use at that particular moment in time. The JWK format is described in [RFC 7517](#). Your app can use the `kid` claim in the JWT header to select which public key in this document has been used to sign a particular token. It can then do signature validation using the correct public key and the indicated algorithm.

NOTE

The v1.0 endpoint returns both the `x5t` and `kid` claims, while the v2.0 endpoint responds with only the `kid` claim. Going forward, we recommend using the `kid` claim to validate your token.

Doing signature validation is outside the scope of this document - there are many open source libraries available for helping you do so if necessary. However, the Microsoft Identity platform has one token signing extension to the standards - custom signing keys.

If your app has custom signing keys as a result of using the [claims-mapping](#) feature, you must append an `appid` query parameter containing the app ID to get a `jwks_uri` pointing to your app's signing key information, which should be used for validation. For example: <https://login.microsoftonline.com/{tenant}/.well-known/openid-configuration?appid=6731de76-14a6-49ae-97bc-6eba6914391e> contains a `jwks_uri` of <https://login.microsoftonline.com/{tenant}/discovery/keys?appid=6731de76-14a6-49ae-97bc-6eba6914391e>.

Claims based authorization

Your application's business logic will dictate this step, some common authorization methods are laid out below.

- Check the `scp` or `roles` claim to verify that all present scopes match those exposed by your API, and allow the client to do the requested action.
- Ensure the calling client is allowed to call your API using the `appid` claim.
- Validate the authentication status of the calling client using `appidacr` - it shouldn't be 0 if public clients aren't allowed to call your API.
- Check against a list of past `nonce` claims to verify the token isn't being replayed.
- Check that the `tid` matches a tenant that is allowed to call your API.
- Use the `acr` claim to verify the user has performed MFA. This should be enforced using [Conditional Access](#).
- If you've requested the `roles` or `groups` claims in the access token, verify that the user is in the group allowed to do this action.
 - For tokens retrieved using the implicit flow, you'll likely need to query the [Microsoft Graph](#) for this data, as it's often too large to fit in the token.

User and application tokens

Your application may receive tokens on behalf of a user (the usual flow) or directly from an application (through the [client credentials flow](#)). These app-only tokens indicate that this call is coming from an application and does not have a user backing it. These tokens are handled largely the same, with some differences:

- App-only tokens will not have a `scp` claim, and may instead have a `roles` claim. This is where application permission (as opposed to delegated permissions) will be recorded. For more information about delegated and application permissions, see permission and consent in [v1.0](#) and [v2.0](#).
- Many human-specific claims will be missing, such as `name` or `upn`.
- The `sub` and `oid` claims will be the same.

Token revocation

Refresh tokens can be invalidated or revoked at any time, for different reasons. These fall into two main categories: timeouts and revocations.

Token timeouts

- `MaxInactiveTime`: If the refresh token hasn't been used within the time dictated by the `MaxInactiveTime`, the Refresh Token will no longer be valid.
- `MaxSessionAge`: If `MaxAgeSessionMultiFactor` or `MaxAgeSessionSingleFactor` have been set to something other than their default (`Until-revoked`), then reauthentication will be required after the time set in the `MaxAgeSession*` elapses.
- Examples:
 - The tenant has a `MaxInactiveTime` of five days, and the user went on vacation for a week, and so Azure AD hasn't seen a new token request from the user in 7 days. The next time the user requests a new token, they'll find their Refresh Token has been revoked, and they must enter their credentials again.
 - A sensitive application has a `MaxAgeSessionSingleFactor` of one day. If a user logs in on Monday, and on Tuesday (after 25 hours have elapsed), they'll be required to reauthenticate.

Revocation

	PASSWORD-BASED COOKIE	PASSWORD-BASED TOKEN	NON-PASSWORD-BASED COOKIE	NON-PASSWORD-BASED TOKEN	CONFIDENTIAL CLIENT TOKEN
Password expires	Stays alive	Stays alive	Stays alive	Stays alive	Stays alive
Password changed by user	Revoked	Revoked	Stays alive	Stays alive	Stays alive

	PASSWORD-BASED COOKIE	PASSWORD-BASED TOKEN	NON-PASSWORD-BASED COOKIE	NON-PASSWORD-BASED TOKEN	CONFIDENTIAL CLIENT TOKEN
User does SSPR	Revoked	Revoked	Stays alive	Stays alive	Stays alive
Admin resets password	Revoked	Revoked	Stays alive	Stays alive	Stays alive
User revokes their refresh tokens via PowerShell	Revoked	Revoked	Revoked	Revoked	Revoked
Admin revokes all refresh tokens for the tenant via PowerShell	Revoked	Revoked	Revoked	Revoked	Revoked
Single sign-out on web	Revoked	Stays alive	Revoked	Stays alive	Stays alive

NOTE

A "Non-password based" login is one where the user didn't type in a password to get it. For example, using your face with Windows Hello, a FIDO2 key, or a PIN.

Primary Refresh Tokens (PRT) on Windows 10 are segregated based on the credential. For example, Windows Hello and password have their respective PRTs, isolated from one another. When a user signs-in with a Hello credential (PIN or biometrics) and then changes the password, the password based PRT obtained previously will be revoked. Signing back in with a password invalidates the old PRT and requests a new one.

Refresh tokens aren't invalidated or revoked when used to fetch a new access token and refresh token.

Next steps

- Learn about [id_tokens](#) in Azure AD.
- Learn about permission and consent in [v1.0](#) and [v2.0](#).

Certificate credentials for application authentication

11/4/2019 • 3 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) allows an application to use its own credentials for authentication, for example, in the OAuth 2.0 Client Credentials Grant flow ([v1.0](#), [v2.0](#)) and the On-Behalf-Of flow ([v1.0](#), [v2.0](#)).

One form of credential that an application can use for authentication is a JSON Web Token(JWT) assertion signed with a certificate that the application owns.

Assertion format

To compute the assertion, you can use one of the many [JSON Web Token](#) libraries in the language of your choice. The information carried by the token are as follows:

PARAMETER	REMARK
<code>alg</code>	Should be RS256
<code>typ</code>	Should be JWT
<code>x5t</code>	Should be the X.509 Certificate SHA-1 thumbprint

Claims (payload)

PARAMETER	REMARKS
<code>aud</code>	Audience: Should be https://login.microsoftonline.com/*tenant_id*/oauth2/token
<code>exp</code>	Expiration date: the date when the token expires. The time is represented as the number of seconds from January 1, 1970 (1970-01-01T0:0:0Z) UTC until the time the token validity expires.
<code>iss</code>	Issuer: should be the client_id (Application ID of the client service)
<code>jti</code>	GUID: the JWT ID
<code>nbf</code>	Not Before: the date before which the token cannot be used. The time is represented as the number of seconds from January 1, 1970 (1970-01-01T0:0:0Z) UTC until the time the token was issued.
<code>sub</code>	Subject: As for <code>iss</code> , should be the client_id (Application ID of the client service)

Signature

The signature is computed applying the certificate as described in the [JSON Web Token RFC7519 specification](#)

Example of a decoded JWT assertion

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "x5t": "gx8tGysyjcRqKjFPnd7RFwwZI0"  
}  
.  
{  
  "aud": "https://login.microsoftonline.com/contoso.onmicrosoft.com/oauth2/token",  
  "exp": 1484593341,  
  "iss": "97e0a5b7-d745-40b6-94fe-5f77d35c6e05",  
  "jti": "22b3bb26-e046-42df-9c96-65dbd72c1c81",  
  "nbf": 1484592741,  
  "sub": "97e0a5b7-d745-40b6-94fe-5f77d35c6e05"  
}  
.  
"Gh95kHCOEGq5E_ArMBbDXhwKR577scxYaoJ1P{a lot of characters here}KKJDEg"
```

Example of an encoded JWT assertion

The following string is an example of encoded assertion. If you look carefully, you notice three sections separated by dots (.):

- The first section encodes the header
- The second section encodes the payload
- The last section is the signature computed with the certificates from the content of the first two sections

```
"eyJhbGciOiJSUzI1NiIsIng1dCI6Imd40HRHeXN5amNScUtqR1BuZDdSRnd2d1pJMCJ9.eyJhdWQiOiJodHRwczpcL1wvbG9naW4ubWljcm9zb2Z0b25saW51LmNvbVwam1wcm1ldXJob3RtYWlsLm9ubWljcm9zb2Z0LmNvbVwvb2F1dGgyXC90b2t1biIsImV4cCI6MTQ4NDU5MzM0MSwi aXNzIjoiOTd1MGE1YjctZDc0NS00MGI2LTk0ZmUtNWY3N2QzNWM2ZTA1IiwianRpIjoiMjJiMjYtZTA0Ni00MmRmLT1j0TYtNjVkJmQ3M mMxYzgxIiwibmJmIjoxNDg0NTkyNzQxLCJzdWIiOiI5N2UwYTViNy1kNzQ1LTQuYjYtOTRmZS01Zjc3ZDM1Yzz1MDUiifQ. Gh95kHCOEGq5E_ArMBbDXhwKR577scxYaoJ1P{a lot of characters here}KKJDEg"
```

Register your certificate with Azure AD

You can associate the certificate credential with the client application in Azure AD through the Azure portal using any of the following methods:

Uploading the certificate file

In the Azure app registration for the client application:

1. Select **Certificates & secrets**.
2. Click on **Upload certificate** and select the certificate file to upload.
3. Click **Add**. Once the certificate is uploaded, the thumbprint, start date, and expiration values are displayed.

Updating the application manifest

Having hold of a certificate, you need to compute:

- `$base64Thumbprint`, which is the base64 encoding of the certificate hash
- `$base64Value`, which is the base64 encoding of the certificate raw data

You also need to provide a GUID to identify the key in the application manifest (`$keyId`).

In the Azure app registration for the client application:

1. Select **Manifest** to open the application manifest.
2. Replace the *keyCredentials* property with your new certificate information using the following schema.

```
"keyCredentials": [  
    {  
        "customKeyIdentifier": "$base64Thumbprint",  
        "KeyId": "$keyid",  
        "type": "AsymmetricX509Cert",  
        "usage": "Verify",  
        "value": "$base64Value"  
    }  
]
```

3. Save the edits to the application manifest and then upload the manifest to Azure AD.

The `keyCredentials` property is multi-valued, so you may upload multiple certificates for richer key management.

Code sample

NOTE

You must calculate the X5T header by using the certificate's hash and converting it to a base64 string. In C# it would look something similar to that of: `System.Convert.ToBase64String(cert.GetCertHash());`

The code sample on [Authenticating to Azure AD in daemon apps with certificates](#) shows how an application uses its own credentials for authentication. It also shows how you can [create a self-signed certificate](#) using the `New-SelfSignedCertificate` Powershell command. You can also take advantage and use the [app creation scripts](#) to create the certificates, compute the thumbprint, and so on.

Azure AD SAML token reference

8/6/2019 • 5 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) emits several types of security tokens in the processing of each authentication flow. This document describes the format, security characteristics, and contents of each type of token.

Claims in SAML tokens

NAME	EQUIVALENT JWT CLAIM	DESCRIPTION	EXAMPLE
Audience	<code>aud</code>	The intended recipient of the token. The application that receives the token must verify that the audience value is correct and reject any tokens intended for a different audience.	<pre><AudienceRestriction> <Audience> https://contoso.com </Audience> </AudienceRestriction></pre>
Authentication Instant		Records the date and time when authentication occurred.	<pre><AuthnStatement AuthnInstant="2011-12-29T05:35:22.000Z"></pre>
Authentication Method	<code>amr</code>	Identifies how the subject of the token was authenticated.	<pre><AuthnContextClassRef> http://schemas.microsoft.com/ws/2008/06/identity/cla </AuthnContextClassRef></pre>
First Name	<code>given_name</code>	Provides the first or "given" name of the user, as set on the Azure AD user object.	<pre><Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity <AttributeValue>Frank<AttributeValue></pre>
Groups	<code>groups</code>	Provides object IDs that represent the subject's group memberships. These values are unique (see Object ID) and can be safely used for managing access, such as enforcing authorization to access a resource. The groups included in the groups claim are configured on a per-application basis, through the "groupMembershipClaims" property of the application manifest. A value of null will exclude all groups, a value of "SecurityGroup" will include only Active Directory Security Group memberships, and a value of "All" will include both Security Groups and Office 365 Distribution Lists.	<pre><Attribute Name="http://schemas.microsoft.com/ws/2008/06/identi <AttributeValue>07dd8a60-bf6d-4e17- 8844-230b77145381<AttributeValue></pre>
Groups Overage Indicator	<code>groups:src1</code>	For token requests that are not length limited (see <code>hasgroups</code> above) but still too large for the token, a link to the full groups list for the user will be included. For SAML this is added as a new claim in place of the <code>groups</code> claim.	<pre><Attribute Name=" http://schemas.microsoft.com/claims/groups.link"> <AttributeValue>https://graph.windows.net/{tenantID}</pre>
Identity Provider	<code>idp</code>	Records the identity provider that authenticated the subject of the token. This value is identical to the value of the Issuer claim unless the user account is in a different tenant than the issuer.	<pre><Attribute Name=" http://schemas.microsoft.com/identity/claims/identit <AttributeValue>https://sts.windows.net/cbb1a5ac- f33b-45fa-9bf5-f37db0fed422/<AttributeValue></pre>
IssuedAt	<code>iat</code>	Stores the time at which the token was issued. It is often used to measure token freshness.	<pre><Assertion ID="_d5ec7a9b-8d8f-4b44-8c94- 9812612142be" IssueInstant="2014-01- 06T20:20:23.085Z" Version="2.0" xmlns="urn:oasis:names:tc:SAML:2.0:assertion"></pre>
Issuer	<code>iss</code>	Identifies the security token service (STS) that constructs and returns the token. In the tokens that Azure AD returns, the issuer is <code>sts.windows.net</code> . The GUID in the Issuer claim value is the tenant ID of the Azure AD directory. The tenant ID is an immutable and reliable identifier of the directory.	<pre><Issuer>https://sts.windows.net/cbb1a5ac- f33b-45fa-9bf5-f37db0fed422/<Issuer></pre>
Last Name	<code>family_name</code>	Provides the last name, surname, or family name of the user as defined in the Azure AD user object.	<pre><Attribute Name=" http://schemas.xmlsoap.org/ws/2005/05/identity/claim <AttributeValue>Miller<AttributeValue></pre>
Name	<code>unique_name</code>	Provides a human readable value that identifies the subject of the token. This value is not guaranteed to be unique within a tenant and is designed to be used only for display purposes.	<pre><Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity <AttributeValue>frankm@contoso.com<AttributeValue></pre>

NAME	EQUIVALENT JWT CLAIM	DESCRIPTION	EXAMPLE
Object ID	<code>oid</code>	Contains a unique identifier of an object in Azure AD. This value is immutable and cannot be reassigned or reused. Use the object ID to identify an object in queries to Azure AD.	<Attribute Name="http://schemas.microsoft.com/identity/claims/oid"/> <AttributeValue>528b2ac2-aa9c-45e1-88d4-959b53bc7dd0<AttributeValue>
Roles	<code>roles</code>	Represents all application roles that the subject has been granted both directly and indirectly through group membership and can be used to enforce role-based access control. Application roles are defined on a per-application basis, through the <code>appRoles</code> property of the application manifest. The <code>value</code> property of each application role is the value that appears in the roles claim.	<Attribute Name="http://schemas.microsoft.com/ws/2008/06/identity/claims/roles"/>
Subject	<code>sub</code>	Identifies the principal about which the token asserts information, such as the user of an application. This value is immutable and cannot be reassigned or reused, so it can be used to perform authorization checks safely. Because the subject is always present in the tokens the Azure AD issues, we recommend using this value in a general purpose authorization system. <code>SubjectConfirmation</code> is not a claim. It describes how the subject of the token is verified. <code>Bearer</code> indicates that the subject is confirmed by their possession of the token.	<Subject> <SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer"/> </Subject>
Tenant ID	<code>tid</code>	An immutable, non-reusable identifier that identifies the directory tenant that issued the token. You can use this value to access tenant-specific directory resources in a multi-tenant application. For example, you can use this value to identify the tenant in a call to the Graph API.	<Attribute Name="http://schemas.microsoft.com/identity/claims/tid"/> <AttributeValue>cbb1a5ac-f33b-45fa-9bf5-f37db0fed422<AttributeValue>
Token Lifetime	<code>nbf</code> , <code>exp</code>	Defines the time interval within which a token is valid. The service that validates the token should verify that the current date is within the token lifetime, else it should reject the token. The service might allow for up to five minutes beyond the token lifetime range to account for any differences in clock time ("time skew") between Azure AD and the service.	<Conditions> NotBefore="2013-03-18T21:32:51.261Z" NotOnOrAfter="2013-03-18T22:32:51.261Z" >

Sample SAML Token

This is a sample of a typical SAML token.

```

<?xml version="1.0" encoding="UTF-8"?>
<t:RequestSecurityTokenResponse xmlns:t="http://schemas.xmlsoap.org/ws/2005/02/trust">
  <t:Lifetime>
    <wsu:Created xmlns:wsu="https://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">2014-12-24T05:15:47.060Z</wsu:Created>
    <wsu:Expires xmlns:wsu="https://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">2014-12-24T06:15:47.060Z</wsu:Expires>
  </t:Lifetime>
  <wsp:AppliesTo xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <EndpointReference xmlns="https://www.w3.org/2005/08/addressing">
      <Address>https://contoso.onmicrosoft.com/MyWebApp</Address>
    </EndpointReference>
  </wsp:AppliesTo>
  <t:RequestedSecurityToken>
    <Assertion xmlns="urn:oasis:names:tc:SAML:2.0:assertion" ID="_3ef08993-846b-41de-99df-b7f3ff77671b" IssueInstant="2014-12-24T05:20:47.060Z" Version="2.0">
      <Issuer>https://sts.windows.net/b9411234-09af-49c2-b0c3-653adc1f376e/</Issuer>
      <ds:Signature xmlns:ds="https://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm="https://www.w3.org/2001/10/xml-exc-c14n#" />
          <ds:SignatureMethod Algorithm="https://www.w3.org/2001/04/xmldsig-more#rsa-sha256" />
          <ds:Reference URI="#_3ef08993-846b-41de-99df-b7f3ff77671b">
            <ds:Transforms>
              <ds:Transform Algorithm="https://www.w3.org/2000/09/xmldsig#enveloped-signature" />
              <ds:Transform Algorithm="https://www.w3.org/2001/10/xml-exc-c14n#" />
            </ds:Transforms>
            <ds:DigestMethod Algorithm="https://www.w3.org/2001/04/xmlenc#sha256" />
            <ds:DigestValue>V1J5P80U1pD24hEyGuAxrbtgR0VghCqI32UkER/nDY=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>j+zPF6mti8Rq4Kyw2NU2nnu0pbJu1z5bR/zDaKa07FCTdmjUzAvIVF8pspVR6CbzCYM3HOAmLhuWmBKAk6qQUbmKsw+XlmF/pB/ivJFdgZSLrt1Bs1P/WBV3t04x6frW4FcIDzh8KhctzJzfSSwGCFyW95er7WxJ10nU41d7jsHRDiboxPgP75jQu2ZER7wOYzr6ff+ha+/Aj3UMw+8ZtC+wCJC3yyENHDAnp2RfgdElJa168enn668fk8pBDjKDGbNB06qBgFPaBT65YvE/tkEmrUxdWkmUKv3y7JWzUYNMd9oUlut93UTyTAIGOs5fVP92fK2NeMVJW7Xg==</ds:SignatureValue>
        <KeyInfo xmlns="https://www.w3.org/2000/09/xmldsig#">
          <X509Data>
            <ds:SignatureValue>MIIDPjCCAabcAwIBAgIQsRiM0jheFZhKk49YD0k1TAjBgUrDgMCHQUAMC0xKzApBgNVBAMTImFjY291bnRzLmFjY2VzC2NvbnRyB2wud2luZG93cy5uZxQwHhtNMTQwMTAxMDcwMDAwJjAtMSswKQYDVQDQEYJHv2Nvdw50cy5h2N1c3Njb250cm9sLndpbmRvd3MuubmV0MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMII8CgKCAQEAKSCWg6q9iYxvJE2NIhSy01KvqoWCO2GFipgH0sTSAs5FahQosk9ZNTztX0ywS/AhsBe0PgYYgFYJL6/EgzVuwrktxr9e31um194flgy/Abxwo9yAduf4dcHPTpCWR1dnDR+0n/4Py1wEuuhQn1r4hpzjFKFLbZnBt77ACSiYx+IH4K4p+NaVEi5wQtSsj0heFzHck49YDOSk1TAjBgurDgCHQUAA41BAQCJ34Prpy7KEC42F5bUaBLQhQ1PNTA1uMDbdnV GKcmSp8M65b8h0Nw1IjGGGy/unXx6iy8pPJX4DyprFTutd2882RwfGE05t4Cw+zZg7djh/JH/ODYRMorxFEW+8uKmXMKmx2wyXMkvfipbTy5LnAU8Jvs2Lg4rOBcXWLAIarZ</X509Certificate>
          <X509Data>
            <KeyInfo>
              </KeyInfo>
            </X509Data>
          </KeyInfo>
        </ds:SignatureValue>
      </ds:Signature>
    </Assertion>
  </t:RequestedSecurityToken>
</t:RequestSecurityTokenResponse>

```

```

<Subject>
  <NameID Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent">m_H3naDei2LNxUmEcWd0BZ1Ni_jVET1pMLR6iQSuYmo</NameID>
  <SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer" />
</Subject>
<Conditions NotBefore="2014-12-24T05:15:47.060Z" NotOnOrAfter="2014-12-24T06:15:47.060Z">
  <AudienceRestriction>
    <Audience>https://contoso.onmicrosoft.com/MyWebApp</Audience>
  </AudienceRestriction>
</Conditions>
<AttributeStatement>
  <Attribute Name="http://schemas.microsoft.com/identity/claims/objectidentifier">
    <AttributeValue>a1addde8-e4f9-4571-ad93-3059e3750d23</AttributeValue>
  </Attribute>
  <Attribute Name="http://schemas.microsoft.com/identity/claims/tenantid">
    <AttributeValue>b9411234-09af-49c2-b0c3-653adc1f376e</AttributeValue>
  </Attribute>
  <Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name">
    <AttributeValue>sample.admin@contoso.onmicrosoft.com</AttributeValue>
  </Attribute>
  <Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname">
    <AttributeValue>Admin</AttributeValue>
  </Attribute>
  <Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname">
    <AttributeValue>Sample</AttributeValue>
  </Attribute>
  <Attribute Name="http://schemas.microsoft.com/ws/2008/06/identity/claims/groups">
    <AttributeValue>5581e43f-6096-41d4-8ffa-04e560bab39d</AttributeValue>
    <AttributeValue>07d8a89-bf6d-4e81-8844-230b77145381</AttributeValue>
    <AttributeValue>0e129f4g-6b0a-4944-982d-f776000632af</AttributeValue>
    <AttributeValue>3ee07328-52ef-4739-a89b-109708c22fb5</AttributeValue>
    <AttributeValue>329k14b3-1851-4b94-947f-9a4dacb595f4</AttributeValue>
    <AttributeValue>6e32c650-9b0a-4491-b429-6c60d2ca9a42</AttributeValue>
    <AttributeValue>f3a169a7-9a58-4ebf-9d47-b70029v07424</AttributeValue>
    <AttributeValue>8e2c86b2-b1ad-476d-9574-544d155aa6ff</AttributeValue>
    <AttributeValue>1bf80264-ff24-4866-b22c-6212e5b9a47</AttributeValue>
    <AttributeValue>4075f9c3-072d-4c32-b542-03e8bc678f3ec</AttributeValue>
    <AttributeValue>76f80527-f2cd-46f4-8c52-8jvd8bc749b1</AttributeValue>
    <AttributeValue>0ba31460-44d0-42b5-b90c-47b3fc48e35c</AttributeValue>
    <AttributeValue>edd41703-8652-4948-94a7-2d917bba7667</AttributeValue>
  </Attribute>
  <Attribute Name="http://schemas.microsoft.com/identity/claims/identityprovider">
    <AttributeValue>https://sts.windows.net/b9411234-09af-49c2-b0c3-653adc1f376e</AttributeValue>
  </Attribute>
</AttributeStatement>
<AuthnStatement AuthnInstant="2014-12-23T18:51:11.000Z">
  <AuthnContext>
    <AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password</AuthnContextClassRef>
  </AuthnContext>
</AuthnStatement>
</Assertion>
<t:RequestedSecurityToken>
  <t:RequestedAttachedReference>
    <SecurityTokenReference xmlns="https://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd" xmlns:d3p1="https://docs.oasis-open.org/wss/oasis-wss-wssecurity-secext-1.1.xsd" d3p1:TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0">
      <KeyIdentifier ValueType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLID">_3ef08993-846b-41de-99df-b7f3ff77671b</KeyIdentifier>
    </SecurityTokenReference>
  </t:RequestedAttachedReference>
  <t:RequestedUnattachedReference>
    <SecurityTokenReference xmlns="https://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd" xmlns:d3p1="https://docs.oasis-open.org/wss/oasis-wss-wssecurity-secext-1.1.xsd" d3p1:TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0">
      <KeyIdentifier ValueType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLID">_3ef08993-846b-41de-99df-b7f3ff77671b</KeyIdentifier>
    </SecurityTokenReference>
  </t:RequestedUnattachedReference>
  <t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0</t:TokenType>
  <t:RequestType>http://schemas.xmlsoap.org/ws/2005/02/trust/Issue</t:RequestType>
  <t:KeyType>http://schemas.xmlsoap.org/ws/2005/05/identity/NoProofKey</t:KeyType>
</t:RequestSecurityTokenResponse>

```

Related content

- See the Azure AD Graph [Policy operations](#) and the [Policy entity](#), to learn more about managing token lifetime policy via the Azure AD Graph API.
- For more information and samples on managing policies via PowerShell cmdlets, including samples, see [Configurable token lifetimes in Azure AD](#).
- Add [custom and optional claims](#) to the tokens for your application.
- Use [Single Sign On \(SSO\) with SAML](#).
- Use the [Azure Single Sign Out SAML Protocol](#)

Application and service principal objects in Azure Active Directory

10/23/2019 • 4 minutes to read • [Edit Online](#)

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.

Application registration

When you register an Azure AD application in the [Azure portal](#), two objects are created in your Azure AD tenant:

- An application object, and
- A service principal object

Application object

An Azure AD application is defined by its one and only application object, which resides in the Azure AD tenant where the application was registered, known as the application's "home" tenant. 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 AD 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 AD tenant. This enables core features such as authentication of the user/application during sign-in, and authorization during resource access.

When an application is given permission to access resources in a tenant (upon registration or [consent](#)), a service principal object is created. The Microsoft Graph [ServicePrincipal entity](#) defines the schema for a service principal object's properties.

Application and service principal relationship

Consider the application object as the *global* representation of your application for use across all tenants, and the service principal as 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 therefore has a 1:1 relationship with the software application, and a 1:many relationships with its corresponding service principal object(s).

A service principal must be created in each tenant where the application is used, enabling it to establish an identity for sign-in and/or access to resources being secured by the tenant. A single-tenant application has only one service principal (in its home tenant), created and consented for use during application registration. A multi-tenant Web application/API also has a service principal created in each tenant where a user from that tenant has consented to its use.

NOTE

Any changes you make to your application object, are also reflected in its service principal object in the application's home tenant only (the tenant where it was registered). For multi-tenant applications, changes to the application object are not reflected in any consumer tenants' service principal objects, until the access is removed through the [Application Access Panel](#) and granted again.

Also note that native applications are registered as multi-tenant by default.

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.

Next steps

- You can use the [Microsoft Graph Explorer](#) to query both the application and service principal objects.
- You can access an application's application object using the Microsoft Graph API, the [Azure portal's](#) application manifest editor, or [Azure AD PowerShell cmdlets](#), as represented by its OData [Application entity](#).
- You can access an application's service principal object through the Microsoft Graph API or [Azure AD PowerShell cmdlets](#), as represented by its OData [ServicePrincipal entity](#).

How and why applications are added to Azure AD

10/23/2019 • 7 minutes to read • [Edit Online](#)

There are two representations of applications in Azure AD:

- **Application objects** - Although there are [exceptions](#), application objects can be considered the definition of an application.
- **Service principals** - Can be considered an instance of an application. Service principals generally reference an application object, and one application object can be referenced by multiple service principals across directories.

What are application objects and where do they come from?

You can manage [application objects](#) in the Azure portal through the [App Registrations](#) experience. Application objects describe the application to Azure AD and can be considered the definition of the application, allowing the service to know how to issue tokens to the application based on its settings. The application object will only exist in its home directory, even if it's a multi-tenant application supporting service principals in other directories. The application object may include any of the following (as well as additional information not mentioned here):

- Name, logo, and publisher
- Redirect URIs
- Secrets (symmetric and/or asymmetric keys used to authenticate the application)
- API dependencies (OAuth)
- Published APIs/resources/scopes (OAuth)
- App roles (RBAC)
- SSO metadata and configuration
- User provisioning metadata and configuration
- Proxy metadata and configuration

Application objects can be created through multiple pathways, including:

- Application registrations in the Azure portal
- Creating a new application using Visual Studio and configuring it to use Azure AD authentication
- When an admin adds an application from the app gallery (which will also create a service principal)
- Using the Microsoft Graph API, Azure AD Graph API, or PowerShell to create a new application
- Many others including various developer experiences in Azure and in API explorer experiences across developer centers

What are service principals and where do they come from?

You can manage [service principals](#) in the Azure portal through the [Enterprise Applications](#) experience. Service principals are what govern an application connecting to Azure AD and can be considered the instance of the application in your directory. For any given application, it can have at most one application object (which is registered in a "home" directory) and one or more service principal objects representing instances of the application in every directory in which it acts.

The service principal can include:

- A reference back to an application object through the application ID property
- Records of local user and group application-role assignments
- Records of local user and admin permissions granted to the application

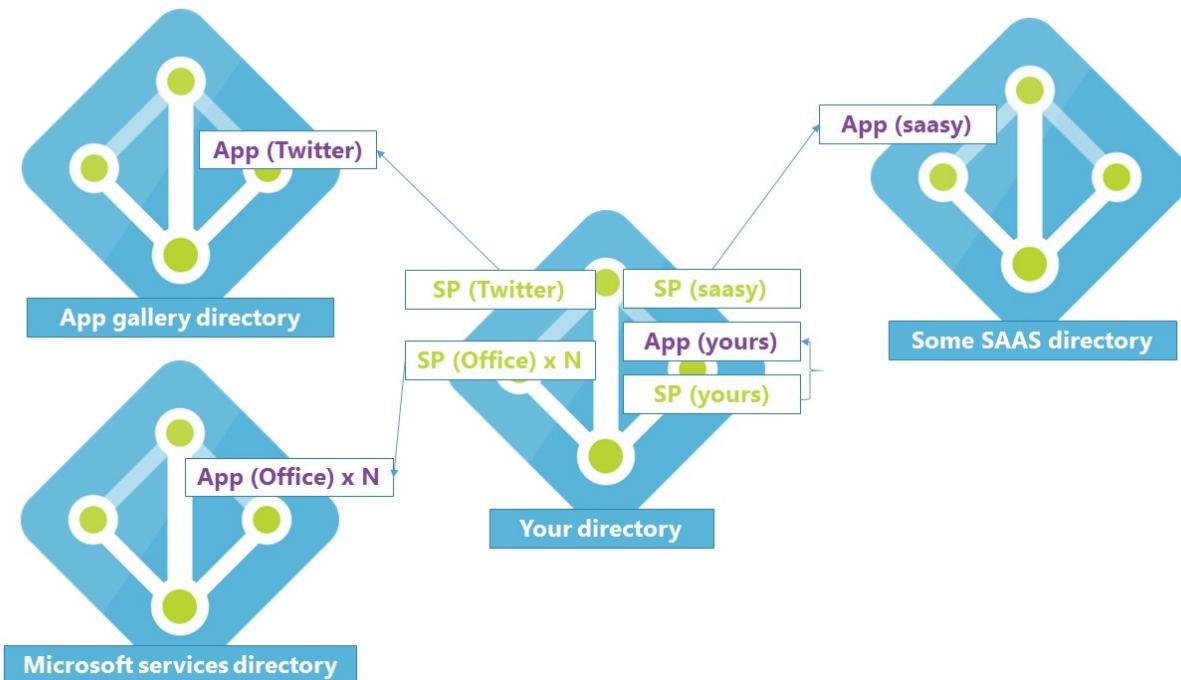
- For example: permission for the application to access a particular user's email
- Records of local policies including Conditional Access policy
- Records of alternate local settings for an application
 - Claims transformation rules
 - Attribute mappings (User provisioning)
 - Directory-specific app roles (if the application supports custom roles)
 - Directory-specific name or logo

Like application objects, service principals can also be created through multiple pathways including:

- When users sign in to a third-party application integrated with Azure AD
 - During sign-in, users are asked to give permission to the application to access their profile and other permissions. The first person to give consent causes a service principal that represents the application to be added to the directory.
- When users sign in to Microsoft online services like [Office 365](#)
 - When you subscribe to Office 365 or begin a trial, one or more service principals are created in the directory representing the various services that are used to deliver all of the functionality associated with Office 365.
 - Some Office 365 services like SharePoint create service principals on an ongoing basis to allow secure communication between components including workflows.
- When an admin adds an application from the app gallery (this will also create an underlying app object)
- Add an application to use the [Azure AD Application Proxy](#)
- Connect an application for single sign on using SAML or password single sign-on (SSO)
- Programmatically via the Azure AD Graph API or PowerShell

How are application objects and service principals related to each other?

An application has one application object in its home directory that is referenced by one or more service principals in each of the directories where it operates (including the application's home directory).



In the preceding diagram, Microsoft maintains two directories internally (shown on the left) that it uses to publish applications:

- One for Microsoft Apps (Microsoft services directory)
- One for pre-integrated third-party applications (App gallery directory)

Application publishers/vendors who integrate with Azure AD are required to have a publishing directory (shown on the right as "Some SaaS Directory").

Applications that you add yourself (represented as **App (yours)** in the diagram) include:

- Apps you developed (integrated with Azure AD)
- Apps you connected for single-sign-on
- Apps you published using the Azure AD application proxy

Notes and exceptions

- Not all service principals point back to an application object. When Azure AD was originally built the services provided to applications were more limited and the service principal was sufficient for establishing an application identity. The original service principal was closer in shape to the Windows Server Active Directory service account. For this reason, it's still possible to create service principals through different pathways, such as using Azure AD PowerShell, without first creating an application object. The Azure AD Graph API requires an application object before creating a service principal.
- Not all of the information described above is currently exposed programmatically. The following are only available in the UI:
 - Claims transformation rules
 - Attribute mappings (User provisioning)
- For more detailed information on the service principal and application objects, see the Azure AD Graph REST API reference documentation:
 - [Application](#)
 - [Service Principal](#)

Why do applications integrate with Azure AD?

Applications are added to Azure AD to leverage one or more of the services it provides including:

- Application authentication and authorization
- User authentication and authorization
- SSO using federation or password
- User provisioning and synchronization
- Role-based access control - Use the directory to define application roles to perform role-based authorization checks in an application
- OAuth authorization services - Used by Office 365 and other Microsoft applications to authorize access to APIs/resources
- Application publishing and proxy - Publish an application from a private network to the internet

Who has permission to add applications to my Azure AD instance?

While there are some tasks that only global administrators can do (such as adding applications from the app gallery and configuring an application to use the Application Proxy) by default all users in your directory have rights to register application objects that they are developing and discretion over which applications they share/give access to their organizational data through consent. If a person is the first user in your directory to sign in to an application and grant consent, that will create a service principal in your tenant; otherwise, the consent grant information will be stored on the existing service principal.

Allowing users to register and consent to applications might initially sound concerning, but keep the following in

mind:

- Applications have been able to leverage Windows Server Active Directory for user authentication for many years without requiring the application to be registered or recorded in the directory. Now the organization will have improved visibility to exactly how many applications are using the directory and for what purpose.
- Delegating these responsibilities to users negates the need for an admin-driven application registration and publishing process. With Active Directory Federation Services (ADFS) it was likely that an admin had to add an application as a relying party on behalf of their developers. Now developers can self-service.
- Users signing in to applications using their organization accounts for business purposes is a good thing. If they subsequently leave the organization they will automatically lose access to their account in the application they were using.
- Having a record of what data was shared with which application is a good thing. Data is more transportable than ever and it's useful to have a clear record of who shared what data with which applications.
- API owners who use Azure AD for OAuth decide exactly what permissions users are able to grant to applications and which permissions require an admin to agree to. Only admins can consent to larger scopes and more significant permissions, while user consent is scoped to the users' own data and capabilities.
- When a user adds or allows an application to access their data, the event can be audited so you can view the Audit Reports within the Azure portal to determine how an application was added to the directory.

If you still want to prevent users in your directory from registering applications and from signing in to applications without administrator approval, there are two settings that you can change to turn off those capabilities:

- To prevent users from consenting to applications on their own behalf:
 1. In the Azure portal, go to the [User settings](#) section under Enterprise applications.
 2. Change **Users can consent to apps accessing company data on their behalf** to **No**.

NOTE

If you decide to turn off user consent, an admin will be required to consent to any new application a user needs to use.

- To prevent users from registering their own applications:
 1. In the Azure portal, go to the [User settings](#) section under Azure Active Directory
 2. Change **Users can register applications** to **No**.

NOTE

Microsoft itself uses the default configuration with users able to register applications and consent to applications on their own behalf.

Tenancy in Azure Active Directory

10/23/2019 • 2 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) organizes objects like users and apps into groups called *tenants*. Tenants allow an administrator to set policies on the users within the organization and the apps that the organization owns to meet their security and operational policies.

Who can sign in to your app?

When it comes to developing apps, developers can choose to configure their app to be either single-tenant or multi-tenant during app registration in the [Azure portal](#).

- Single-tenant apps are only available in the tenant they were registered in, also known as their home tenant.
- Multi-tenant apps are available to users in both their home tenant and other tenants.

In the Azure portal, you can configure your app to be single-tenant or multi-tenant by setting the audience as follows.

AUDIENCE	SINGLE/MULTI-TENANT	WHO CAN SIGN IN
Accounts in this directory only	Single tenant	All user and guest accounts in your directory can use your application or API. <i>Use this option if your target audience is internal to your organization.</i>
Accounts in any Azure AD directory	Multi-tenant	All users and guests with a work or school account from Microsoft can use your application or API. This includes schools and businesses that use Office 365. <i>Use this option if your target audience is business or educational customers.</i>
Accounts in any Azure AD directory and personal Microsoft accounts (such as Skype, Xbox, Outlook.com)	Multi-tenant	All users with a work or school, or personal Microsoft account can use your application or API. It includes schools and businesses that use Office 365 as well as personal accounts that are used to sign in to services like Xbox and Skype. <i>Use this option to target the widest set of Microsoft accounts.</i>

Best practices for multi-tenant apps

Building great multi-tenant apps can be challenging because of the number of different policies that IT administrators can set in their tenants. If you choose to build a multi-tenant app, follow these best practices:

- Test your app in a tenant that has configured [Conditional Access policies](#).
- Follow the principle of least user access to ensure that your app only requests permissions it actually needs. Avoid requesting permissions that require admin consent as this may prevent users from acquiring your app at all in some organizations.

- Provide appropriate names and descriptions for any permissions you expose as part of your app. This helps users and admins know what they are agreeing to when they attempt to use your app's APIs. For more information, see the best practices section in the [permissions guide](#).

Next steps

- [How to convert an app to be multi-tenant](#)

Permissions and consent in the Azure Active Directory v1.0 endpoint

10/29/2019 • 6 minutes to read • [Edit Online](#)

Applies to:

- Azure AD v1.0 endpoint

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 to by administrators](#) because they are typically powerful and allow access to data across user-boundaries, or data that would otherwise be restricted to administrators. Users who are defined as owners of the resource application (i.e. the API which publishes the permissions) are also allowed to grant application permissions for the APIs they own.

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.

NOTE

You can view the permissions that an Azure AD Application or Service Principal exposes using the Azure portal, or PowerShell. Try this script to view the permissions exposed by Microsoft Graph.

```
Connect-AzureAD

# Get OAuth2 Permissions/delegated permissions
(Get-AzureADServicePrincipal -filter "DisplayName eq 'Microsoft Graph'").OAuth2Permissions

# Get App roles/application permissions
(Get-AzureADServicePrincipal -filter "DisplayName eq 'Microsoft Graph'").AppRoles
```

PROPERTY NAME	DESCRIPTION	EXAMPLE
<code>ID</code>	Is a GUID value that uniquely identifies this permission.	570282fd-fa5c-430d-a7fd-fc8dc98a9dca
<code>.IsEnabled</code>	Indicates whether this permission is available for use.	true
<code>Type</code>	Indicates whether this permission requires user consent or admin consent.	User
<code>AdminConsentDescription</code>	Is a description that's shown to administrators during the admin consent experiences	Allows the app to read email in user mailboxes.
<code>AdminConsentDisplayName</code>	Is the friendly name that's shown to administrators during the admin consent experience.	Read user mail
<code>UserConsentDescription</code>	Is a description that's shown to users during a user consent experience.	Allows the app to read email in your mailbox.
<code>UserConsentDisplayName</code>	Is the friendly name that's shown to users during a user consent experience.	Read your mail
<code>Value</code>	Is the string that's used to identify the permission during OAuth 2.0 authorize flows. <code>Value</code> 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.

Learn more about registering an Azure AD app that requests access to a static set of APIs.

- **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. [Learn more about how to grant admin consent](#).

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

For example:

- Mail.Read - Allows users to read mail.
- Mail.ReadWrite - Allows users to read or write mail.
- Mail.ReadWrite.All - Allows an administrator or user to access all mail in the organization.

Azure Active Directory consent framework

10/23/2019 • 3 minutes to read • [Edit Online](#)

The Azure Active Directory (Azure AD) consent framework makes it easy to develop multi-tenant web and native client applications. These applications allow sign-in by user accounts from an Azure AD tenant that's different from the one where the application is registered. They may also need to access web APIs such as the Microsoft Graph API (to access Azure AD, Intune, and services in Office 365) and other Microsoft services' APIs, in addition to your own web APIs.

The framework is based on a user or an administrator giving consent to an application that asks to be registered in their directory, which may involve accessing directory data. For example, if a web client application needs to read calendar information about the user from Office 365, that user is required to consent to the client application first. After consent is given, the client application will be able to call the Microsoft Graph API on behalf of the user, and use the calendar information as needed. The [Microsoft Graph API](#) provides access to data in Office 365 (like calendars and messages from Exchange, sites and lists from SharePoint, documents from OneDrive, notebooks from OneNote, tasks from Planner, and workbooks from Excel), as well as users and groups from Azure AD and other data objects from more Microsoft cloud services.

The consent framework is built on OAuth 2.0 and its various flows, such as authorization code grant and client credentials grant, using public or confidential clients. By using OAuth 2.0, Azure AD makes it possible to build many different types of client applications--such as on a phone, tablet, server, or a web application--and gain access to the required resources.

For more info about using the consent framework with OAuth2.0 authorization grants, see [Authorize access to web applications using OAuth 2.0 and Azure AD](#) and [Authentication scenarios for Azure AD](#). For info about getting authorized access to Office 365 through Microsoft Graph, see [App authentication with Microsoft Graph](#).

Consent experience - an example

The following steps show you how the consent experience works for both the application developer and the user.

1. Assume you have a web client application that needs to request specific permissions to access a resource/API. You'll learn how to do this configuration in the next section, but essentially the Azure portal is used to declare permission requests at configuration time. Like other configuration settings, they become part of the application's Azure AD registration:

Fourth Coffee Web App - API permissions

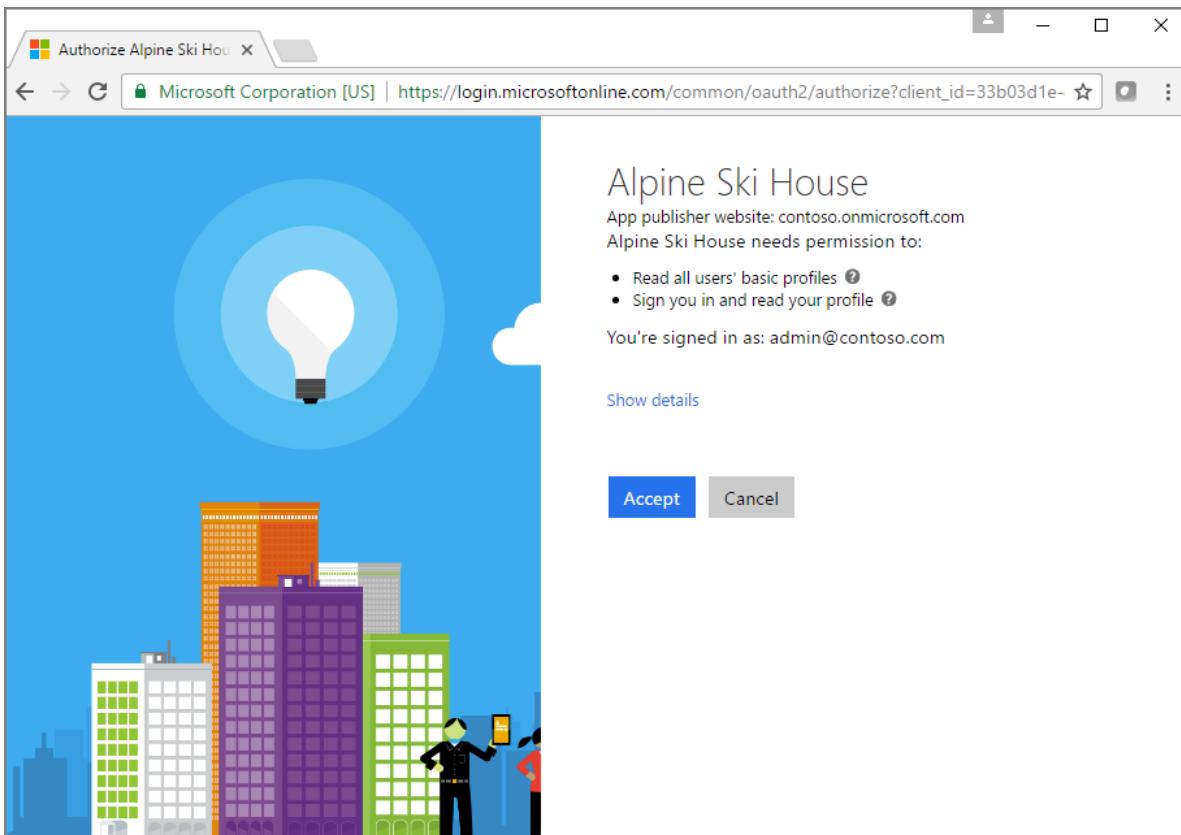
Overview Quickstart Manage API permissions Add a permission API / PERMISSIONS NAME TYPE DESCRIPTION ADMIN CONSENT REQUIRED Microsoft Graph (1) User.Read Delegated Sign in and read user profile - These are the permissions that this application requests statically. You may also request user consentable permissions dynamically through code. See best practices for requesting permissions Grant consent Grant admin consent for Fourth Coffee

- Consider that your application's permissions have been updated, the application is running, and a user is about to use it for the first time. First, the application needs to obtain an authorization code from Azure AD's `/authorize` endpoint. The authorization code can then be used to acquire a new access and refresh token.
- If the user is not already authenticated, Azure AD's `/authorize` endpoint prompts the user to sign in.

The screenshot shows the Microsoft Azure Active Directory sign-in page. The URL in the browser is `https://login.microsoftonline.com`. The page has a blue header with the Microsoft logo and the text "Sign in to your account". Below the header, there is a large graphic of a lightbulb inside a blue circle, with buildings in the background. The text "Alpine Ski House" is displayed above the sign-in form. The sign-in form includes fields for "Email or phone" and "Password", a "Keep me signed in" checkbox, and a "Sign in" button. Below the form, there is a link "Can't access your account?". At the bottom, there is copyright information "© 2017 Microsoft" and links for "Terms of use", "Privacy & Cookies", and the Microsoft logo.

- After the user has signed in, Azure AD will determine if the user needs to be shown a consent page. This determination is based on whether the user (or their organization's administrator) has already granted the application consent. If consent has not already been granted, Azure AD prompts the user for consent and

displays the required permissions it needs to function. The set of permissions that are displayed in the consent dialog match the ones selected in the **Delegated permissions** in the Azure portal.



5. After the user grants consent, an authorization code is returned to your application, which is redeemed to acquire an access token and refresh token. For more information about this flow, see [Web API app type](#).
6. As an administrator, you can also consent to an application's delegated permissions on behalf of all the users in your tenant. Administrative consent prevents the consent dialog from appearing for every user in the tenant, and can be done in the [Azure portal](#) by users with the administrator role. To learn which administrator roles can consent to delegated permissions, see [Administrator role permissions in Azure AD](#).

To consent to an app's delegated permissions

- a. Go to the **API permissions** page for your application
- b. Click on the **Grant admin consent** button.

A screenshot of the Azure portal showing the "Fourth Coffee Web App - API permissions" page. The left sidebar shows navigation options like Overview, Quickstart, Manage, Branding, Authentication, Certificates & secrets, API permissions (which is selected and highlighted with a red box), Expose an API, Owners, Manifest, Support + Troubleshooting, Troubleshooting, and New support request. The main content area has a header "API permissions" with a note: "Permissions have changed. Users and/or admins will have to consent even if they have already done so previously." It lists "API / PERMISSIONS NAME", "TYPE", "DESCRIPTION", and "ADMIN CONSENT REQUIRED" columns. A row for "Microsoft Graph (1)" shows "User.Read" as Delegated, "Sign in and read user profile", and "No". Below this, a note says "These are the permissions that this application requests statically. You may also request user consenable permissions dynamically through code. See best practices for requesting permissions". At the bottom is a section titled "Grant consent" with the note "As an administrator, you can grant consent on behalf of all users in this directory. Granting admin consent for all users means that end users will not be shown a consent screen when using the application." A red box highlights the "Grant admin consent for Fourth Coffee" button at the bottom.

IMPORTANT

Granting explicit consent using the **Grant permissions** button is currently required for single-page applications (SPA) that use ADAL.js. Otherwise, the application fails when the access token is requested.

Next steps

- See [how to convert an app to be multi-tenant](#)
- For more depth, learn [how consent is supported at the OAuth 2.0 protocol layer during the authorization code grant flow](#).

Understanding Azure AD application consent experiences

10/23/2019 • 4 minutes to read • [Edit Online](#)

Learn more about the Azure Active Directory (Azure AD) application consent user experience. So you can intelligently manage applications for your organization and/or develop applications with a more seamless consent experience.

Consent and permissions

Consent is the process of a user granting authorization to an application to access protected resources on their behalf. An admin or user can be asked for consent to allow access to their organization/individual data.

The actual user experience of granting consent will differ depending on policies set on the user's tenant, the user's scope of authority (or role), and the type of [permissions](#) being requested by the client application. This means that application developers and tenant admins have some control over the consent experience. Admins have the flexibility of setting and disabling policies on a tenant or app to control the consent experience in their tenant. Application developers can dictate what types of permissions are being requested and if they want to guide users through the user consent flow or the admin consent flow.

- **User consent flow** is when an application developer directs users to the authorization endpoint with the intent to record consent for only the current user.
- **Admin consent flow** is when an application developer directs users to the admin consent endpoint with the intent to record consent for the entire tenant. To ensure the admin consent flow works properly, application developers must list all permissions in the `RequiredResourceAccess` property in the application manifest. For more info, see [Application manifest](#).

Building blocks of the consent prompt

The consent prompt is designed to ensure users have enough information to determine if they trust the client application to access protected resources on their behalf. Understanding the building blocks will help users granting consent make more informed decisions and it will help developers build better user experiences.

The following diagram and table provide information about the building blocks of the consent prompt.



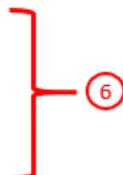
① kelly@contoso.com

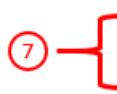
② Permissions requested

③  Contoso Test App ④
zawad.co ⑤

This app would like to:

- ✓ Read and write your files
- ✓ Read your calendar
- ✗ Sign you in and read your profile



⑦  Allows you to sign in to the app with your organizational account and let the app read your profile. It also allows the app to read basic company information.

Accepting these permissions means that you allow this app to use your data as specified in their [terms of service](#) and [privacy statement](#). You can change these permissions at <https://myapps.microsoft.com>.

Only accept if you trust the publisher and if you selected this app from a store or website you trust. Ask your admin if you're not sure. Microsoft is not involved in licensing this app to you. [Hide details](#)

Cancel

Accept

#	COMPONENT	PURPOSE
1	User identifier	This identifier represents the user that the client application is requesting to access protected resources on behalf of.
2	Title	The title changes based on whether the users are going through the user or admin consent flow. In user consent flow, the title will be "Permissions requested" while in the admin consent flow the title will have an additional line "Accept for your organization".

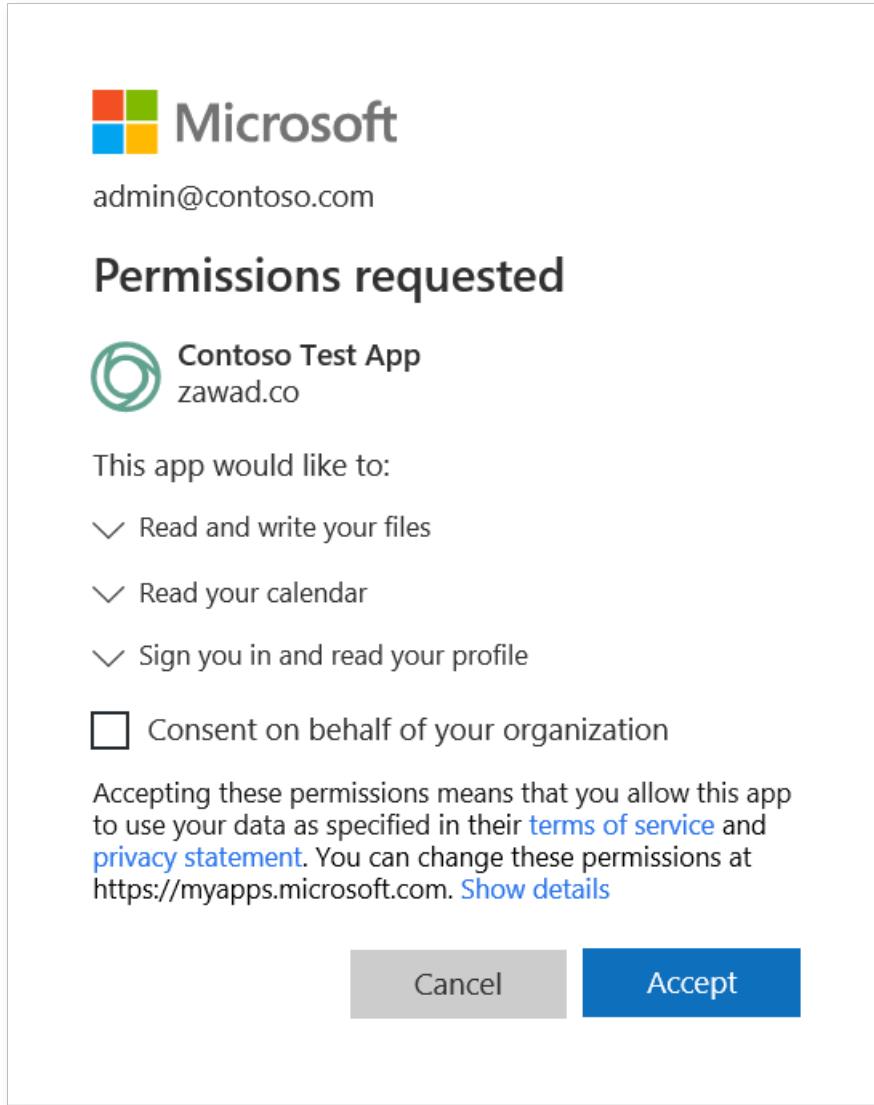
#	COMPONENT	PURPOSE
3	App logo	This image should help users have a visual cue of whether this app is the app they intended to access. This image is provided by application developers and the ownership of this image isn't validated.
4	App name	This value should inform users which application is requesting access to their data. Note this name is provided by the developers and the ownership of this app name isn't validated.
5	Publisher domain	This value should provide users with a domain they may be able to evaluate for trustworthiness. This domain is provided by the developers and the ownership of this publisher domain is validated.
6	Permissions	This list contains the permissions being requested by the client application. Users should always evaluate the types of permissions being requested to understand what data the client application will be authorized to access on their behalf if they accept. As an application developer it is best to request access to the permissions with the least privilege.
7	Permission description	This value is provided by the service exposing the permissions. To see the permission descriptions, you must toggle the chevron next to the permission.
8	App terms	These terms contain links to the terms of service and privacy statement of the application. The publisher is responsible for outlining their rules in their terms of service. Additionally, the publisher is responsible for disclosing the way they use and share user data in their privacy statement. If the publisher doesn't provide links to these values for multi-tenant applications, there will be a bolded warning on the consent prompt.
9	https://myapps.microsoft.com	This is the link where users can review and remove any non-Microsoft applications that currently have access to their data.

Common consent scenarios

Here are the consent experiences that a user may see in the common consent scenarios:

1. Individuals accessing an app that directs them to the user consent flow while requiring a permission set that is within their scope of authority.

- a. Admins will see an additional control on the traditional consent prompt that will allow them consent on behalf of the entire tenant. The control will be defaulted to off, so only when admins explicitly check the box will consent be granted on behalf of the entire tenant. As of today, this checkbox will only show for the Global Admin role, so Cloud Admin and App Admin will not see this checkbox.



- b. Users will see the traditional consent prompt.



user@contoso.com

Permissions requested



Contoso Test App

zawad.co

This app would like to:

- ✓ Read and write your files
- ✓ Read your calendar
- ✓ Sign you in and read your profile

Accepting these permissions means that you allow this app to use your data as specified in their [terms of service](#) and [privacy statement](#). You can change these permissions at <https://myapps.microsoft.com>. [Show details](#)

Cancel

Accept

2. Individuals accessing an app that requires at least one permission that is outside their scope of authority.
 - a. Admins will see the same prompt as 1.i shown above.
 - b. Users will be blocked from granting consent to the application, and they will be told to ask their admin for access to the app.



user@contoso.com

Need admin approval



Contoso Test App

zawad.co

Contoso Test App needs permission to access resources in your organization that only an admin can grant. Please ask an admin to grant permission to this app before you can use it.

[Have an admin account? Sign in with that account](#)

[Return to the application without granting consent](#)

3. Individuals that navigate or are directed to the admin consent flow.

- a. Admin users will see the admin consent prompt. The title and the permission descriptions changed on this prompt, the changes highlight the fact that accepting this prompt will grant the app access to the requested data on behalf of the entire tenant.



user@contoso.com

Permissions requested Accept for your organization



Contoso Test App

zawad.co

This app would like to:

- ✓ Read user and shared contacts
- ✓ Read user and shared calendars
- ✓ Sign in and read user profile

Accepting these permissions means that you allow this app to use your data as specified in their [terms of service](#) and [privacy statement](#). You can change these permissions at <https://myapps.microsoft.com>. [Show details](#)

Cancel

Accept

- b. Non-admin users will see the same screen as 2.ii shown above.

Next steps

- Get a step-by-step overview of [how the Azure AD consent framework implements consent](#).
- For more depth, learn [how a multi-tenant application can use the consent framework](#) to implement "user" and "admin" consent, supporting more advanced multi-tier application patterns.
- Learn [how to configure the app's publisher domain](#).

Developer guidance for Azure Active Directory Conditional Access

10/15/2019 • 9 minutes to read • [Edit Online](#)

The Conditional Access feature in Azure Active Directory (Azure AD) 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:

- Multi-factor authentication
- Allowing only Intune enrolled devices to access specific services
- Restricting user locations and IP ranges

For more information on the full capabilities of Conditional Access, see [Conditional Access in Azure Active Directory](#).

For developers building apps for Azure AD, this article shows how you can use Conditional Access and you'll also learn about the impact of accessing resources that you don't have control over that may have Conditional Access policies applied. The article also explores the implications of Conditional Access in the on-behalf-of flow, web apps, accessing Microsoft Graph, and calling APIs.

Knowledge of [single](#) and [multi-tenant](#) apps and [common authentication patterns](#) is assumed.

How does Conditional Access impact an app?

App types impacted

In most common cases, Conditional Access does not change an app's behavior or requires any changes from the developer. Only in certain cases when an app indirectly or silently requests a token for a service, an app requires 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 ADAL.js
- Web Apps calling a resource

Conditional Access policies can be applied to the app, but also can be applied to a web API your app accesses. To learn more about how to configure a Conditional Access policy, see [Quickstart: Require MFA for specific apps with Azure Active Directory Conditional Access](#).

Depending on the scenario, an enterprise customer can apply and remove Conditional Access policies at any time. In order for your app to continue functioning when a new policy is applied, you need to implement the "challenge" handling. The following examples illustrate 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 multi-factor 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 multi-factor authentication (MFA).

- You are building a native 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 native 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 native app, which needs to comply with the Conditional Access policy.

Microsoft Graph

Microsoft Graph has special considerations when building apps in Conditional Access environments. Generally, the mechanics of Conditional Access behave the same, but the policies your users see will be based on the underlying data your app is requesting from the graph.

Specifically, all Microsoft Graph scopes represent some dataset that can individually have policies applied. Since Conditional Access policies are assigned the specific datasets, Azure AD will enforce Conditional Access policies based on the data behind Graph - rather than Graph itself.

For example, if an app requests the following Microsoft Graph scopes,

```
scopes="Bookings.Read.All Mail.Read"
```

An app can expect their users to fulfill all policies set on Bookings and Exchange. Some scopes may map to multiple datasets if it grants access.

Complying with a Conditional Access policy

For several different app topologies, a Conditional Access policy is evaluated when the session is established. As a Conditional Access policy operates on the granularity of apps and services, the point at which it is invoked depends heavily on the scenario you're trying to accomplish.

When your app attempts to access a service with a Conditional Access policy, it may encounter a Conditional Access challenge. This challenge is encoded in the `claims` parameter that comes in a response from Azure AD. Here's an example of this challenge parameter:

```
claims={"access_token":{"polids":{"essential":true,"Values":["<GUID>"]}}}
```

Developers can take this challenge and append it onto a new request to Azure AD. Passing this state prompts the end user to perform any action necessary to comply with the Conditional Access policy. In the following scenarios, specifics of the error and how to extract the parameter are explained.

Scenarios

Prerequisites

Azure AD Conditional Access is a feature included in [Azure AD Premium](#). You can learn more about licensing requirements in the [unlicensed usage report](#). Developers can join the [Microsoft Developer Network](#), which includes a free subscription to the Enterprise Mobility Suite, which includes Azure AD Premium.

Considerations for specific scenarios

The following information only applies in these Conditional Access scenarios:

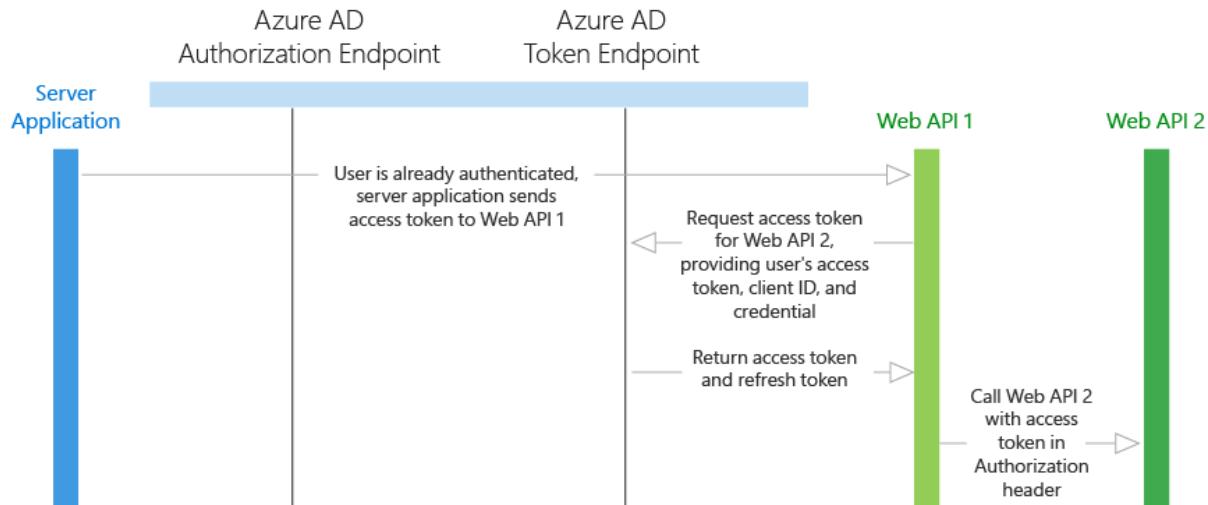
- Apps performing the on-behalf-of flow
- Apps accessing multiple services/resources
- Single-page apps using ADAL.js

The following sections discuss common scenarios that are more complex. The core operating principle is

Conditional Access policies are evaluated at the time the token is requested for the service that has a Conditional Access policy applied.

Scenario: App performing the on-behalf-of flow

In this scenario, we walk through the case in which a native app calls a web service/API. In turn, this service does the "on-behalf-of" flow to call a downstream service. In our case, we've applied our Conditional Access policy to the downstream service (Web API 2) and are using a native app rather than a server/daemon app.



The initial token request for Web API 1 does not prompt the end user for multi-factor authentication as Web API 1 may not always hit the downstream API. Once Web API 1 tries to request a token on-behalf-of the user for Web API 2, the request fails since the user has not signed in with multi-factor authentication.

Azure AD returns an HTTP response with some interesting data:

NOTE

In this instance it's a multi-factor authentication error description, but there's a wide range of `interaction_required` possible pertaining to Conditional Access.

```
HTTP 400; Bad Request
error=interaction_required
error_description=AADSTS50076: Due to a configuration change made by your administrator, or because you moved
to a new location, you must use multi-factor authentication to access '<Web API 2 App/Client ID>'.
claims={"access_token":{"polids":{"essential":true,"Values":["<GUID>"]}}}
```

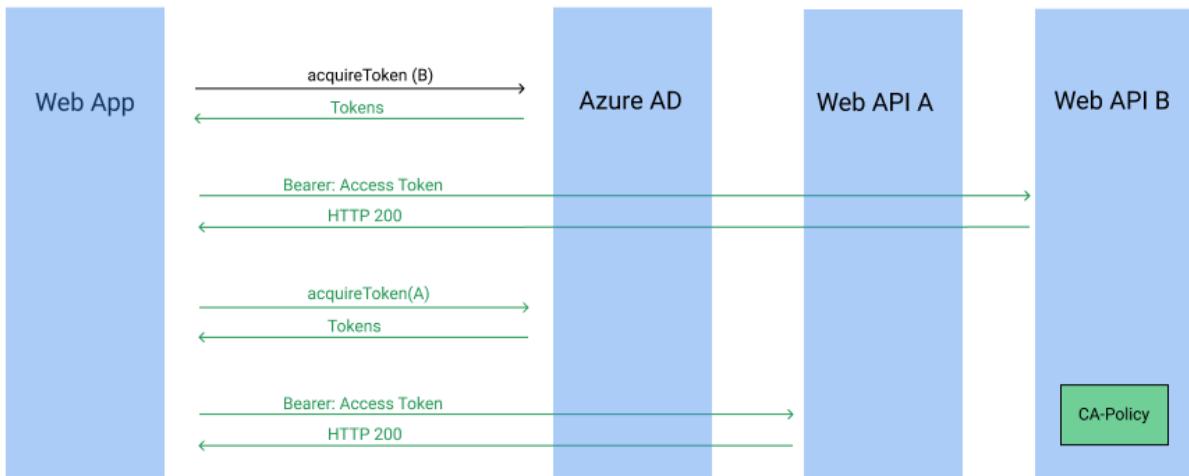
In Web API 1, we catch the error `error=interaction_required`, and send back the `claims` challenge to the desktop app. At that point, the desktop app can make a new `acquireToken()` call and append the `claims` challenge as an extra query string parameter. This new request requires the user to do multi-factor authentication and then send this new token back to Web API 1 and complete the on-behalf-of flow.

To try out this scenario, see our [.NET code sample](#). It demonstrates how to pass the claims challenge back from Web API 1 to the native app and construct a new request inside the client app.

Scenario: App accessing multiple services

In this scenario, we walk through the case in which a web app accesses two services one of which has a Conditional Access policy assigned. Depending on your app logic, there may exist a path in which your app does not require access to both web services. In this scenario, the order in which you request a token plays an important role in the end user experience.

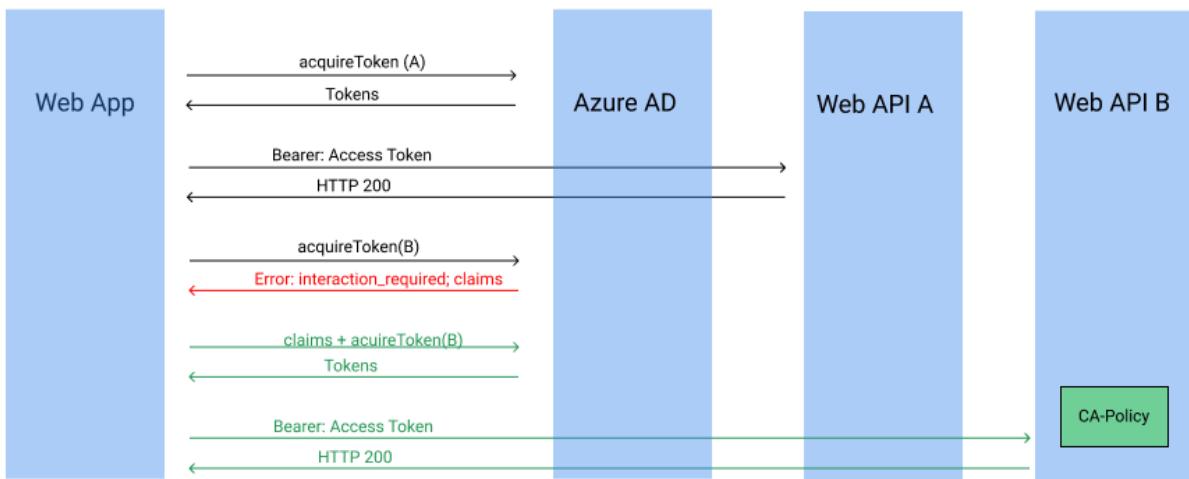
Let's assume we have web service A and B and web service B has our Conditional Access policy applied. While the initial interactive auth request requires consent for both services, the Conditional Access policy is not required in all cases. If the app requests a token for web service B, then the policy is invoked and subsequent requests for web service A also succeeds as follows.



Alternatively, if the app initially requests a token for web service A, the end user does not invoke the Conditional Access policy. This allows the app developer to control the end user experience and not force the Conditional Access policy to be invoked in all cases. The tricky case is if the app subsequently requests a token for web service B. At this point, the end user needs to comply with the Conditional Access policy. When the app tries to `acquireToken`, it may generate the following error (illustrated in the following diagram):

```

HTTP 400; Bad Request
error=interaction_required
error_description=AADSTS50076: Due to a configuration change made by your administrator, or because you moved
to a new location, you must use multi-factor authentication to access '<Web API App/Client ID>'.
claims={"access_token":{"polids":{"essential":true,"Values":["<GUID>"]}}}
  
```



If the app is using the ADAL library, a failure to acquire the token is always retried interactively. When this interactive request occurs, the end user has the opportunity to comply with the Conditional Access. This is true unless the request is a `AcquireTokenSilentAsync` or `PromptBehavior.Never` in which case the app needs to perform an interactive `AcquireToken` request to give the end user the opportunity to comply with the policy.

Scenario: Single-page app (SPA) using ADAL.js

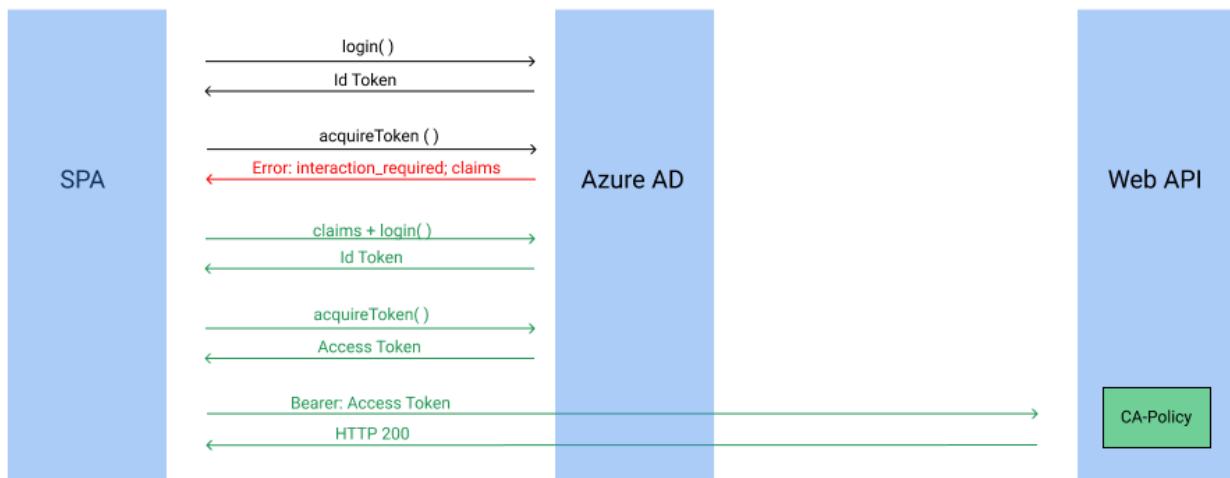
In this scenario, we walk through the case when we have a single-page app (SPA), using ADAL.js to call a Conditional Access protected web API. This is a simple architecture but has some nuances that need to be taken

into account when developing around Conditional Access.

In ADAL.js, there are a few functions that obtain tokens: `login()`, `acquireToken(...)`, `acquireTokenPopup(...)`, and `acquireTokenRedirect(...)`.

- `login()` obtains an ID token through an interactive sign-in request but does not obtain access tokens for any service (including a Conditional Access protected web API).
- `acquireToken(...)` can then be used to silently obtain an access token meaning it does not show UI in any circumstance.
- `acquireTokenPopup(...)` and `acquireTokenRedirect(...)` are both used to interactively request a token for a resource meaning they always show sign-in UI.

When an app needs an access token to call a Web API, it attempts an `acquireToken(...)`. If the token session is expired or we need to comply with a Conditional Access policy, then the `acquireToken` function fails and the app uses `acquireTokenPopup()` or `acquireTokenRedirect()`.



Let's walk through an example with our Conditional Access scenario. The end user just landed on the site and doesn't have a session. We perform a `login()` call, get an ID token without multi-factor authentication. Then the user hits a button that requires the app to request data from a web API. The app tries to do an `acquireToken()` call but fails since the user has not performed multi-factor authentication yet and needs to comply with the Conditional Access policy.

Azure AD sends back the following HTTP response:

```
HTTP 400; Bad Request
error=interaction_required
error_description=AADSTS50076: Due to a configuration change made by your administrator, or because you moved
to a new location, you must use multi-factor authentication to access '<Web API App/Client ID>'.
```

Our app needs to catch the `error=interaction_required`. The application can then use either `acquireTokenPopup()` or `acquireTokenRedirect()` on the same resource. The user is forced to do a multi-factor authentication. After the user completes the multi-factor authentication, the app is issued a fresh access token for the requested resource.

To try out this scenario, see our [JS SPA On-behalf-of code sample](#). This code sample uses the Conditional Access policy and web API you registered earlier with a JS SPA to demonstrate this scenario. It shows how to properly handle the claims challenge and get an access token that can be used for your Web API. Alternatively, checkout the general [Angular.js code sample](#) for guidance on an Angular SPA

See also

- To learn more about the capabilities, see [Conditional Access in Azure Active Directory](#).

- For more Azure AD code samples, see [GitHub repo of code samples](#).
- For more info on the ADAL SDK's and access the reference documentation, see [library guide](#).
- To learn more about multi-tenant scenarios, see [How to sign in users using the multi-tenant pattern](#).

National clouds

10/23/2019 • 2 minutes to read • [Edit Online](#)

National clouds are physically isolated instances of Azure. These regions of Azure are designed to make sure that data residency, sovereignty, and compliance requirements are honored within geographical boundaries.

Including the global cloud, Azure ActiveDirectory(Azure AD) is deployed in the following national clouds:

- Azure Government
- Azure Germany
- Azure China 21Vianet

National clouds are unique and a separate environment from Azure global. It's important to be aware of key differences while developing your application for these environments. Differences include registering applications, acquiring tokens, and configuring endpoints.

App registration endpoints

There's a separate Azure portal for each one of the national clouds. To integrate applications with the Microsoft identity platform in a national cloud, you're required to register your application separately in each Azure portal that's specific to the environment.

The following table lists the base URLs for the Azure AD endpoints used to register an application for each national cloud.

NATIONAL CLOUD	AZURE AD PORTAL ENDPOINT
Azure AD for US Government	https://portal.azure.us
Azure AD Germany	https://portal.microsoftazure.de
Azure AD China operated by 21Vianet	https://portal.azure.cn
Azure AD (global service)	https://portal.azure.com

Azure AD authentication endpoints

All the national clouds authenticate users separately in each environment and have separate authentication endpoints.

The following table lists the base URLs for the Azure AD endpoints used to acquire tokens for each national cloud.

NATIONAL CLOUD	AZURE AD AUTHENTICATION ENDPOINT
Azure AD for US Government	https://login.microsoftonline.us
Azure AD Germany	https://login.microsoftonline.de
Azure AD China operated by 21Vianet	https://login.chinacloudapi.cn

NATIONAL CLOUD	AZURE AD AUTHENTICATION ENDPOINT
Azure AD (global service)	https://login.microsoftonline.com

You can form requests to the Azure AD authorization or token endpoints by using the appropriate region-specific base URL. For example, for Azure Germany:

- Authorization common endpoint is <https://login.microsoftonline.de/common/oauth2/authorize>.
- Token common endpoint is <https://login.microsoftonline.de/common/oauth2/token>.

For single-tenant applications, replace "common" in the previous URLs with your tenant ID or name. An example is <https://login.microsoftonline.de/contoso.com>.

Microsoft Graph API

To learn how to call the Microsoft Graph APIs in a national cloud environment, go to [Microsoft Graph in national cloud deployments](#).

IMPORTANT

Certain services and features that are in specific regions of the global service might not be available in all of the national clouds. To find out what services are available, go to [Products available by region](#).

To learn how to build an application by using the Microsoft identity platform, follow the [Microsoft Authentication Library \(MSAL\) tutorial](#). Specifically, this app will sign in a user and get an access token to call the Microsoft Graph API.

Next steps

Learn more about:

- [Azure Government](#)
- [Azure China 21Vianet](#)
- [Azure Germany](#)
- [Azure AD authentication basics](#)

How to: Enable cross-app SSO on Android using ADAL

5/21/2019 • 6 minutes to read • [Edit Online](#)

Applies to:

- Azure AD v1.0 endpoint
- Azure Active Directory Authentication Library (ADAL)

Single sign-on (SSO) allows users to only enter their credentials once and have those credentials automatically work across applications and across platforms that other applications may use (such as Microsoft Accounts or a work account from Microsoft 365) no matter the publisher.

Microsoft's identity platform, along with the SDKs, makes it easy to enable SSO within your own suite of apps, or with the broker capability and Authenticator applications, across the entire device.

In this how-to, you'll learn how to configure the SDK within your application to provide SSO to your customers.

Prerequisites

This how-to assumes that you know how to:

- Provision your app using the legacy portal for Azure Active Directory (Azure AD). For more info, see [Register an app](#)
- Integrate your application with the [Azure AD Android SDK](#).

Single sign-on concepts

Identity brokers

Microsoft provides applications for every mobile platform that allow for the bridging of credentials across applications from different vendors and for enhanced features that require a single secure place from where to validate credentials. These are called **brokers**.

On iOS and Android, brokers are provided through downloadable applications that customers either install independently or pushed to the device by a company who manages some, or all, of the devices for their employees. Brokers support managing security just for some applications or the entire device based on IT admin configuration. In Windows, this functionality is provided by an account chooser built in to the operating system, known technically as the Web Authentication Broker.

Broker assisted login

Broker-assisted logins are login experiences that occur within the broker application and use the storage and security of the broker to share credentials across all applications on the device that apply the identity platform. The implication being your applications will rely on the broker to sign users in. On iOS and Android, these brokers are provided through downloadable applications that customers either install independently or can be pushed to the device by a company who manages the device for their user. An example of this type of application is the Microsoft Authenticator application on iOS. In Windows, this functionality is provided by an account chooser built in to the operating system, known technically as the Web Authentication Broker. The experience varies by platform and can sometimes be disruptive to users if not managed correctly. You're probably most familiar with this pattern if you

have the Facebook application installed and use Facebook Connect from another application. The identity platform uses the same pattern.

On Android, the account chooser is displayed on top of your application, which is less disruptive to the user.

How the broker gets invoked

If a compatible broker is installed on the device, like the Microsoft Authenticator application, the identity SDKs will automatically do the work of invoking the broker for you when a user indicates they wish to log in using any account from the identity platform.

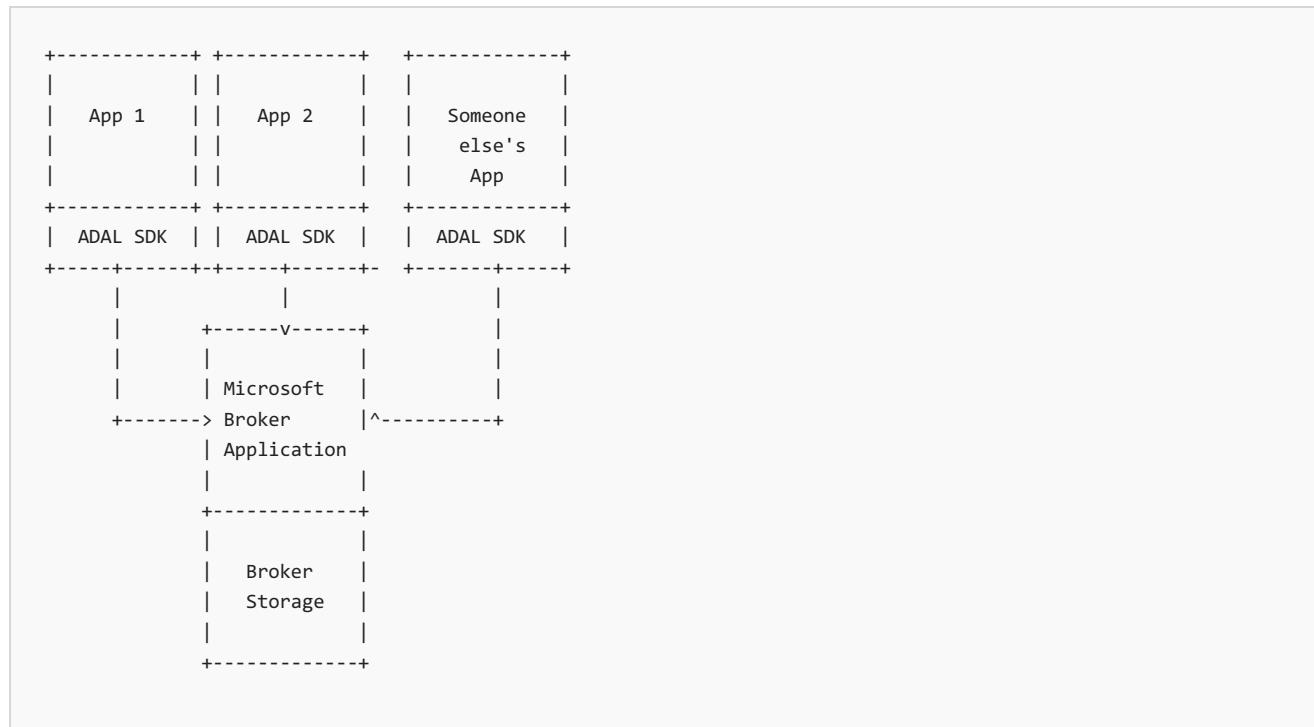
How Microsoft ensures the application is valid

The need to ensure the identity of an application call the broker is crucial to the security provided in broker assisted logins. iOS and Android do not enforce unique identifiers that are valid only for a given application, so malicious applications may "spoof" a legitimate application's identifier and receive the tokens meant for the legitimate application. To ensure Microsoft is always communicating with the right application at runtime, the developer is asked to provide a custom redirectURI when registering their application with Microsoft. **How developers should craft this redirect URI is discussed in detail below.** This custom redirectURI contains the certificate thumbprint of the application and is ensured to be unique to the application by the Google Play Store. When an application calls the broker, the broker asks the Android operating system to provide it with the certificate thumbprint that called the broker. The broker provides this certificate thumbprint to Microsoft in the call to the identity system. If the certificate thumbprint of the application does not match the certificate thumbprint provided to us by the developer during registration, access is denied to the tokens for the resource the application is requesting. This check ensures that only the application registered by the developer receives tokens.

Brokered-SSO logins have the following benefits:

- User experiences SSO across all their applications no matter the vendor.
- Your application can use more advanced business features such as Conditional Access and support Intune scenarios.
- Your application can support certificate-based authentication for business users.
- More secure sign-in experience as the identity of the application and the user are verified by the broker application with additional security algorithms and encryption.

Here is a representation of how the SDKs work with the broker applications to enable SSO:



Turning on SSO for broker assisted SSO

The ability for an application to use any broker that is installed on the device is turned off by default. In order to use your application with the broker, you must do some additional configuration and add some code to your application.

The steps to follow are:

1. Enable broker mode in your application code's calling to the MS SDK
2. Establish a new redirect URI and provide that to both the app and your app registration
3. Setting up the correct permissions in the Android manifest

Step 1: Enable broker mode in your application

The ability for your application to use the broker is turned on when you create the "settings" or initial setup of your Authentication instance. To do this in your app:

```
AuthenticationSettings.Instance.setUseBroker(true);
```

Step 2: Establish a new redirect URI with your URL Scheme

In order to ensure that the right application receives the returned credential tokens, there is a need to make sure the call back to your application in a way that the Android operating system can verify. The Android operating system uses the hash of the certificate in the Google Play store. This hash of the certificate cannot be spoofed by a rogue application. Along with the URI of the broker application, Microsoft ensures that the tokens are returned to the correct application. A unique redirect URI is required to be registered on the application.

Your redirect URI must be in the proper form of:

```
msauth://packagename/Base64UrlencodedSignature
```

ex: *msauth://com.example.userapp/lcB5PxlyvbLkbFVtBI%2FitkW%2Fejk%3D*

You can register this redirect URI in your app registration using the [Azure portal](#). For more information on Azure AD app registration, see [Integrating with Azure Active Directory](#).

Step 3: Set up the correct permissions in your application

The broker application in Android uses the Accounts Manager feature of the Android OS to manage credentials across applications. In order to use the broker in Android your app manifest must have permissions to use AccountManager accounts. These permissions are discussed in detail in the [Google documentation for Account Manager here](#)

In particular, these permissions are:

```
GET_ACCOUNTS  
USE_CREDENTIALS  
MANAGE_ACCOUNTS
```

You've configured SSO!

Now the identity SDK will automatically both share credentials across your applications and invoke the broker if it's present on their device.

Next steps

- Learn about [Single sign-on SAML protocol](#)

How to: Enable cross-app SSO on iOS using ADAL

5/21/2019 • 12 minutes to read • [Edit Online](#)

Applies to:

- Azure AD v1.0 endpoint
- Azure Active Directory Authentication Library (ADAL)

Single sign-on (SSO) allows users to only enter their credentials once and have those credentials automatically work across applications and across platforms that other applications may use (such as Microsoft Accounts or a work account from Microsoft 365) no matter the publisher.

Microsoft's identity platform, along with the SDKs, makes it easy to enable SSO within your own suite of apps, or with the broker capability and Authenticator applications, across the entire device.

In this how-to, you'll learn how to configure the SDK within your application to provide SSO to your customers.

This how-to applies to:

- Azure Active Directory (Azure Active Directory)
- Azure Active Directory B2C
- Azure Active Directory B2B
- Azure Active Directory Conditional Access

Prerequisites

This how-to assumes that you know how to:

- Provision your app using the legacy portal for Azure AD. For more info, see [Register an app](#)
- Integrate your application with the [Azure AD iOS SDK](#).

Single sign-on concepts

Identity brokers

Microsoft provides applications for every mobile platform that allow for the bridging of credentials across applications from different vendors and for enhanced features that require a single secure place from where to validate credentials. These are called **brokers**.

On iOS and Android, brokers are provided through downloadable applications that customers either install independently or pushed to the device by a company who manages some, or all, of the devices for their employees. Brokers support managing security just for some applications or the entire device based on IT admin configuration. In Windows, this functionality is provided by an account chooser built in to the operating system, known technically as the Web Authentication Broker.

Patterns for logging in on mobile devices

Access to credentials on devices follow two basic patterns:

- Non-broker assisted logins
- Broker assisted logins

Non-broker assisted logins

Non-broker assisted logins are login experiences that happen inline with the application and use the local storage on the device for that application. This storage may be shared across applications but the credentials are tightly bound to the app or suite of apps using that credential. You've most likely experienced this in many mobile applications when you enter a username and password within the application itself.

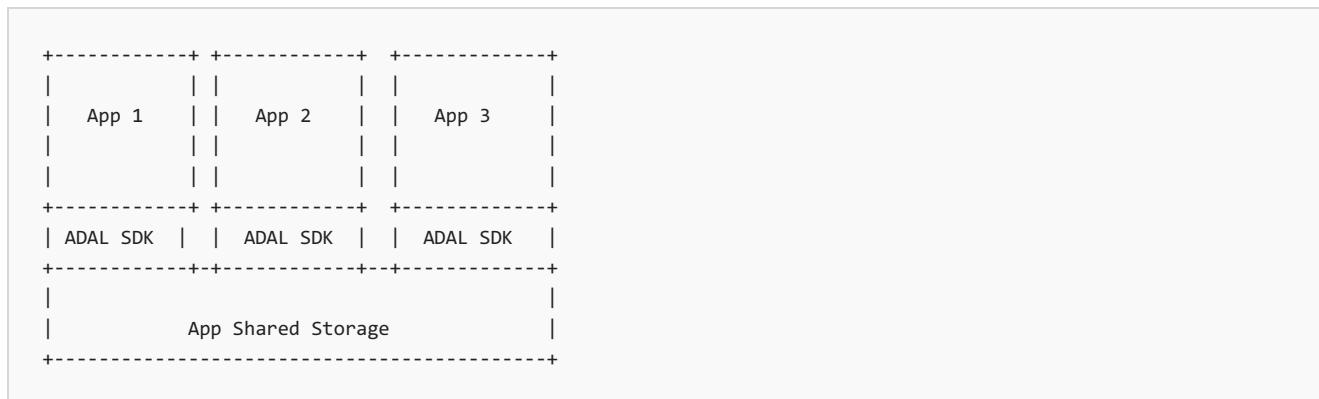
These logins have the following benefits:

- User experience exists entirely within the application.
- Credentials can be shared across applications that are signed by the same certificate, providing a single sign-on experience to your suite of applications.
- Control around the experience of logging in is provided to the application before and after sign-in.

These logins have the following drawbacks:

- Users cannot experience single-sign on across all apps that use a Microsoft identity, only across those Microsoft identities that your application has configured.
- Your application cannot be used with more advanced business features such as Conditional Access or use the Intune suite of products.
- Your application can't support certificate-based authentication for business users.

Here is a representation of how the SDKs work with the shared storage of your applications to enable SSO:



Broker assisted logins

Broker-assisted logins are login experiences that occur within the broker application and use the storage and security of the broker to share credentials across all applications on the device that apply the identity platform. This means that your applications rely on the broker to sign users in. On iOS and Android, these brokers are provided through downloadable applications that customers either install independently or pushed to the device by a company who manages the device for their user. An example of this type of application is the Microsoft Authenticator application on iOS. In Windows this functionality is provided by an account chooser built in to the operating system, known technically as the Web Authentication Broker.

The experience varies by platform and can sometimes be disruptive to users if not managed correctly. You're probably most familiar with this pattern if you have the Facebook application installed and use Facebook Connect from another application. The identity platform uses the same pattern.

For iOS this leads to a "transition" animation where your application is sent to the background while the Microsoft Authenticator application comes to the foreground for the user to select which account they would like to sign in with.

For Android and Windows the account chooser is displayed on top of your application, which is less disruptive to the user.

How the broker gets invoked

If a compatible broker is installed on the device, like the Microsoft Authenticator application, the SDKs will automatically do the work of invoking the broker for you when a user indicates they wish to log in using any

account from the identity platform. This account could be a personal Microsoft Account, a work or school account, or an account that you provide and host in Azure using our B2C and B2B products.

How we ensure the application is valid

The need to ensure the identity of an application call the broker is crucial to the security we provide in broker assisted logins. Neither iOS nor Android enforces unique identifiers that are valid only for a given application, so malicious applications may "spoof" a legitimate application's identifier and receive the tokens meant for the legitimate application. To ensure we are always communicating with the right application at runtime, we ask the developer to provide a custom redirectURI when registering their application with Microsoft. How developers should craft this redirect URI is discussed in detail below. This custom redirectURI contains the Bundle ID of the application and is ensured to be unique to the application by the Apple App Store. When an application calls the broker, the broker asks the iOS operating system to provide it with the Bundle ID that called the broker. The broker provides this Bundle ID to Microsoft in the call to our identity system. If the Bundle ID of the application does not match the Bundle ID provided to us by the developer during registration, we will deny access to the tokens for the resource the application is requesting. This check ensures that only the application registered by the developer receives tokens.

The developer has the choice whether the SDK calls the broker or uses the non-broker assisted flow.

However if the developer chooses not to use the broker-assisted flow they lose the benefit of using SSO credentials that the user may have already added on the device and prevents their application from being used with business features Microsoft provides its customers such as Conditional Access, Intune management capabilities, and certificate-based authentication.

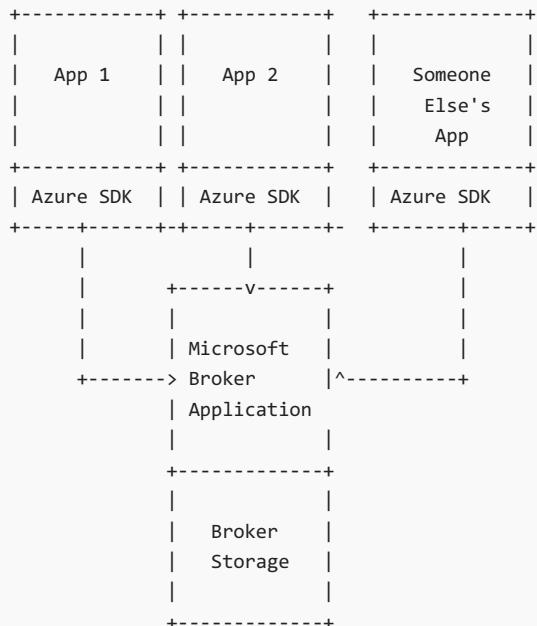
These logins have the following benefits:

- User experiences SSO across all their applications no matter the vendor.
- Your application can use more advanced business features such as Conditional Access or use the Intune suite of products.
- Your application can support certificate-based authentication for business users.
- Much more secure sign-in experience as the identity of the application and the user are verified by the broker application with additional security algorithms and encryption.

These logins have the following drawbacks:

- In iOS the user is transitioned out of your application's experience while credentials are chosen.
- Loss of the ability to manage the login experience for your customers within your application.

Here is a representation of how the SDKs work with the broker applications to enable SSO:



Enabling cross-app SSO using ADAL

Here we use the ADAL iOS SDK to:

- Turn on non-broker assisted SSO for your suite of apps
- Turn on support for broker-assisted SSO

Turning on SSO for non-broker assisted SSO

For non-broker assisted SSO across applications, the SDKs manage much of the complexity of SSO for you. This includes finding the right user in the cache and maintaining a list of logged in users for you to query.

To enable SSO across applications you own you need to do the following:

1. Ensure all your applications use the same Client ID or Application ID.
2. Ensure that all of your applications share the same signing certificate from Apple so that you can share keychains.
3. Request the same keychain entitlement for each of your applications.
4. Tell the SDKs about the shared keychain you want us to use.

Using the same Client ID / Application ID for all the applications in your suite of apps

In order for the identity platform to know that it's allowed to share tokens across your applications, each of your applications will need to share the same Client ID or Application ID. This is the unique identifier that was provided to you when you registered your first application in the portal.

Redirect URIs allow you to identify different apps to the Microsoft identity service if it uses the same Application ID. Each application can have multiple Redirect URIs registered in the onboarding portal. Each app in your suite will have a different redirect URI. An example of how this looks is below:

App1 Redirect URI: `x-msauth-mytestiosapp://com.myapp.mytestapp`

App2 Redirect URI: `x-msauth-mytestiosapp://com.myapp.mytestapp2`

App3 Redirect URI: `x-msauth-mytestiosapp://com.myapp.mytestapp3`

...

These are nested under the same client ID / application ID and looked up based on the redirect URI you return to us in your SDK configuration.



The format of these redirect URIs is explained below. You may use any Redirect URI unless you wish to support the broker, in which case they must look something like the above*

Create keychain sharing between applications

Enabling keychain sharing is beyond the scope of this document and covered by Apple in their document [Adding Capabilities](#). What is important is that you decide what you want your keychain to be called and add that capability across all your applications.

When you do have entitlements set up correctly you should see a file in your project directory entitled

`entitlements.plist` that contains something that looks like the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "https://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>keychain-access-groups</key>
    <array>
        <string>$(AppIdentifierPrefix)com.myapp.mytestapp</string>
        <string>$(AppIdentifierPrefix)com.myapp.mycache</string>
    </array>
</dict>
</plist>

```

Once you have the keychain entitlement enabled in each of your applications, and you are ready to use SSO, tell the identity SDK about your keychain by using the following setting in your `ADAuthenticationSettings` with the following setting:

```
defaultKeychainSharingGroup=@"com.myapp.mycache";
```

WARNING

When you share a keychain across your applications any application can delete users or worse delete all the tokens across your application. This is particularly disastrous if you have applications that rely on the tokens to do background work. Sharing a keychain means that you must be very careful in any and all remove operations through the identity SDKs.

That's it! The SDK will now share credentials across all your applications. The user list will also be shared across application instances.

Turning on SSO for broker assisted SSO

The ability for an application to use any broker that is installed on the device is **turned off by default**. In order to use your application with the broker you must do some additional configuration and add some code to your application.

The steps to follow are:

1. Enable broker mode in your application code's call to the MS SDK.
2. Establish a new redirect URI and provide that to both the app and your app registration.
3. Registering a URL Scheme.
4. Add a permission to your info.plist file.

Step 1: Enable broker mode in your application

The ability for your application to use the broker is turned on when you create the "context" or initial setup of your Authentication object. You do this by setting your credentials type in your code:

```
/*! See the ADCredentialsType enumeration definition for details */
@propertyADCredentialsType credentialsType;
```

The `AD_CREDENTIALS_AUTO` setting will allow the SDK to try to call out to the broker, `AD_CREDENTIALS_EMBEDDED` will prevent the SDK from calling to the broker.

Step 2: Registering a URL Scheme

The identity platform uses URLs to invoke the broker and then return control back to your application. To finish that round trip you need a URL scheme registered for your application that the identity platform will know about. This can be in addition to any other app schemes you may have previously registered with your application.

WARNING

We recommend making the URL scheme fairly unique to minimize the chances of another app using the same URL scheme. Apple does not enforce the uniqueness of URL schemes that are registered in the app store.

Below is an example of how this appears in your project configuration. You may also do this in XCode as well:

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleTypeRole</key>
        <string>Editor</string>
        <key>CFBundleURLName</key>
        <string>com.myapp.mytestapp</string>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>x-msauth-mytestiosapp</string>
        </array>
    </dict>
</array>
```

Step 3: Establish a new redirect URI with your URL Scheme

In order to ensure that we always return the credential tokens to the correct application, we need to make sure we call back to your application in a way that the iOS operating system can verify. The iOS operating system reports to the Microsoft broker applications the Bundle ID of the application calling it. This cannot be spoofed by a rogue application. Therefore, we leverage this along with the URL of our broker application to ensure that the tokens are returned to the correct application. We require you to establish this unique redirect URI both in your application and set as a Redirect URI in our developer portal.

Your redirect URI must be in the proper form of:

```
<app-scheme>://<your.bundle.id>
```

ex: `x-msauth-mytestiosapp://com.myapp.mytestapp`

This redirect URI needs to be specified in your app registration using the [Azure portal](#). For more information on Azure AD app registration, see [Integrating with Azure Active Directory](#).

Step 3a: Add a redirect URI in your app and dev portal to support certificate-based authentication

To support cert-based authentication a second "msauth" needs to be registered in your application and the [Azure portal](#) to handle certificate authentication if you wish to add that support in your application.

```
msauth://code/<broker-redirect-uri-in-url-encoded-form>
```

ex: `msauth://code/x-msauth-mytestiosapp%3A%2F%2Fcom.myapp.mytestapp`

Step 4: Add a configuration parameter to your app

ADAL uses `-canOpenURL` to check if the broker is installed on the device. In iOS 9 on, Apple locked down what schemes an application can query for. You will need to add "msauth" to the `LSApplicationQueriesSchemes` section of your `info.plist` file.

```
<key>LSApplicationQueriesSchemes</key>
<array>
    <string>msauth</string>
</array>
```

You've configured SSO!

Now the identity SDK will automatically both share credentials across your applications and invoke the broker if it's present on their device.

Next steps

- Learn about [Single sign-on SAML protocol](#)

Debug SAML-based single sign-on to applications in Azure Active Directory

7/1/2019 • 4 minutes to read • [Edit Online](#)

Learn how to find and fix [single sign-on](#) issues for applications in Azure Active Directory (Azure AD) that support [Security Assertion Markup Language \(SAML\) 2.0](#).

Before you begin

We recommend installing the [My Apps Secure Sign-in Extension](#). This browser extension makes it easy to gather the SAML request and SAML response information that you need to resolving issues with single sign-on. In case you cannot install the extension, this article shows you how to resolve issues both with and without the extension installed.

To download and install the My Apps Secure Sign-in Extension, use one of the following links.

- [Chrome](#)
- [Microsoft Edge](#)
- [Firefox](#)

Test SAML-based single sign-on

To test SAML-based single sign-on between Azure AD and a target application:

1. Sign in to the [Azure portal](#) as a global administrator or other administrator that is authorized to manage applications.
2. In the left blade, select **Azure Active Directory**, and then select **Enterprise applications**.
3. From the list of enterprise applications, select the application for which you want to test single sign-on, and then from the options on the left select **Single sign-on**.
4. To open the SAML-based single sign-on testing experience, go to **Test single sign-on** (step 5). If the **Test** button is greyed out, you need to fill out and save the required attributes first in the **Basic SAML Configuration** section.
5. In the **Test single sign-on** blade, use your corporate credentials to sign in to the target application. You can sign in as the current user or as a different user. If you sign in as a different user, a prompt will ask you to authenticate.

Applications > Salesforce - Ignite - Single sign-on > Single sign-on

Test single sign-on with Salesforce - Ignite

Please make sure you have configured Salesforce - Ignite before testing.

[Sign in as current user](#)

[Sign in as someone else](#)

Resolving errors

If you encounter an error in the sign-in page, please paste it below. If you still see the same issue, please wait for couple of minutes and retry.

What does the error look like? [?](#)

```
Request Id: 39ddf9da-aaf3-4fa8-9bd4-5271348c0000
Correlation Id: b8beb14d-c8b6-4d5a-ba9c-c2eb7aab7c19
Timestamp: 2019-02-15T23:34:03Z
Message: AADSTS650056: Misconfigured application. This could be due to one of the following: The client has not listed any permissions for 'AAD Graph' in the requested permissions in the client's application registration. Or, The admin has not consented in the tenant. Or, Check the application identifier in the request to ensure it matches the configured client application identifier. Please contact your admin to fix the
```

[Get resolution guidance](#)

1 Signin is unsuccessful. See resolution steps below.

- Download the SAML request

Root cause: Identifier (EntityID) mismatch or missing. <https://salesforce.net> is not the correct identifier (EntityID) for this application.

Resolution:

Update the Identifier (EntityID) for Salesforce - Ignite with the value below:

<https://luisca9111-dev-ed.my.salesforce.com> [Fix It](#)

Was this helpful?

[Yes](#) [No](#)

5 Test single sign-on with Salesforce - Ignite

Test to see if single sign-on is working. Users will be redirected to the application.

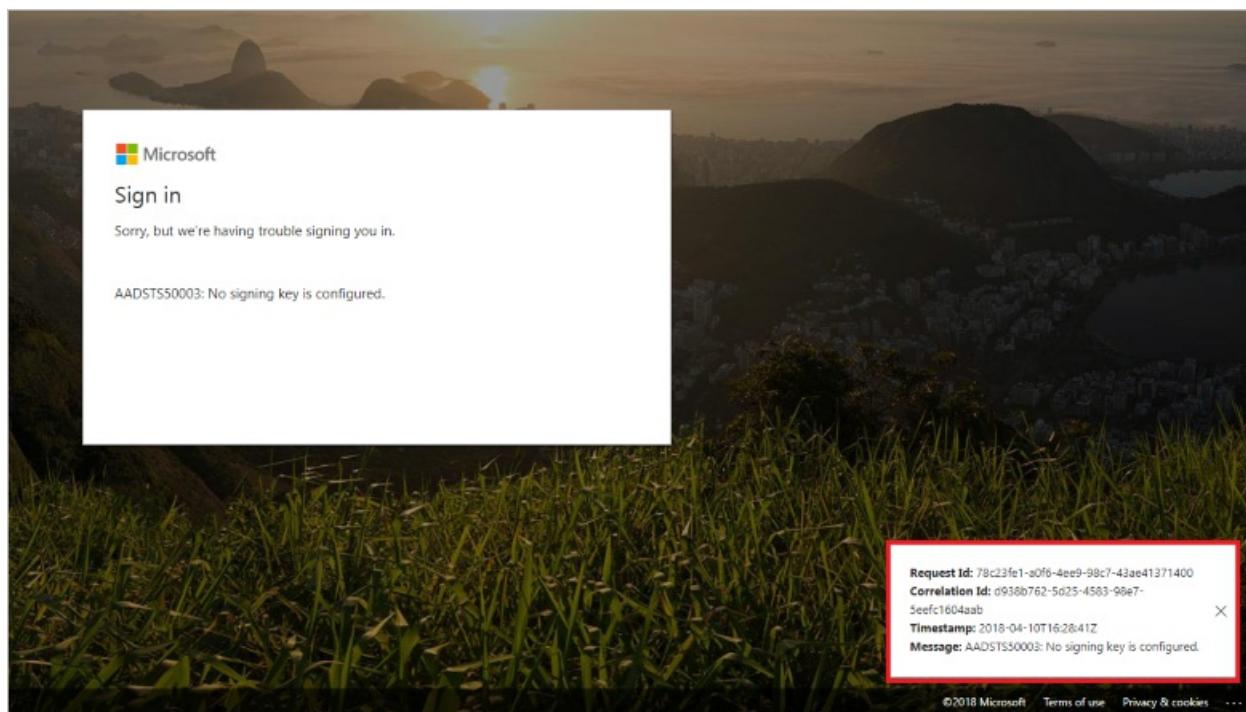
[Test](#)

If you are successfully signed in, the test has passed. In this case, Azure AD issued a SAML response token to the application. The application used the SAML token to successfully sign you in.

If you have an error on the company sign-in page or the application's page, use one of the next sections to resolve the error.

Resolve a sign-in error on your company sign-in page

When you try to sign in, you might see an error on your company sign-in page that's similar to the following example.



To debug this error, you need the error message and the SAML request. The My Apps Secure Sign-in Extension

automatically gathers this information and displays resolution guidance on Azure AD.

To resolve the sign-in error with the My Apps Secure Sign-in Extension installed

1. When an error occurs, the extension redirects you back to the Azure AD **Test single sign-on** blade.
2. On the **Test single sign-on** blade, select **Download the SAML request**.
3. You should see specific resolution guidance based on the error and the values in the SAML request.
4. You will see a **Fix it** button to automatically update the configuration in Azure AD to resolve the issue. If you don't see this button, then the sign-in issue is not due to a misconfiguration on Azure AD.

If no resolution is provided for the sign-in error, we suggest that you use the feedback textbox to inform us.

To resolve the error without installing the My Apps Secure Sign-in Extension

1. Copy the error message at the bottom right corner of the page. The error message includes:
 - A CorrelationID and Timestamp. These values are important when you create a support case with Microsoft because they help the engineers to identify your problem and provide an accurate resolution to your issue.
 - A statement identifying the root cause of the problem.
2. Go back to Azure AD and find the **Test single sign-on** blade.
3. In the text box above **Get resolution guidance**, paste the error message.
4. Click **Get resolution guidance** to display steps for resolving the issue. The guidance might require information from the SAML request or SAML response. If you're not using the My Apps Secure Sign-in Extension, you might need a tool such as [Fiddler](#) to retrieve the SAML request and response.
5. Verify that the destination in the SAML request corresponds to the SAML Single Sign-On Service URL obtained from Azure AD.
6. Verify the issuer in the SAML request is the same identifier you have configured for the application in Azure AD. Azure AD uses the issuer to find an application in your directory.
7. Verify AssertionConsumerServiceURL is where the application expects to receive the SAML token from Azure AD. You can configure this value in Azure AD, but it's not mandatory if it's part of the SAML request.

Resolve a sign-in error on the application page

You might sign in successfully and then see an error on the application's page. This occurs when Azure AD issued a token to the application, but the application does not accept the response.

To resolve the error, follow these steps:

1. If the application is in the Azure AD Gallery, verify that you've followed all the steps for integrating the application with Azure AD. To find the integration instructions for your application, see the [list of SaaS application integration tutorials](#).
2. Retrieve the SAML response.
 - If the My Apps Secure Sign-in extension is installed, from the **Test single sign-on** blade, click **download the SAML response**.
 - If the extension is not installed, use a tool such as [Fiddler](#) to retrieve the SAML response.
3. Notice these elements in the SAML response token:
 - User unique identifier of NameID value and format
 - Claims issued in the token
 - Certificate used to sign the token.

For more information on the SAML response, see [Single Sign-on SAML protocol](#).

4. Now that you have reviewed the SAML response, see [Error on an application's page after signing in](#) for guidance on how to resolve the problem.
5. If you're still not able to sign in successfully, you can ask the application vendor what is missing from the SAML response.

Next steps

Now that single sign-on is working to your application, you could [Automate user provisioning and de-provisioning to SaaS applications](#) or [get started with Conditional Access](#).

How to: Configure the role claim issued in the SAML token for enterprise applications

10/23/2019 • 6 minutes to read • [Edit Online](#)

By using Azure Active Directory (Azure AD), you can customize the claim type for the role claim in the response token that you receive after you authorize an app.

Prerequisites

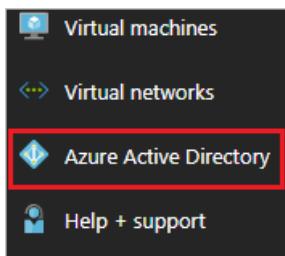
- An Azure AD subscription with directory setup.
- A subscription that has single sign-on (SSO) enabled. You must configure SSO with your application.

When to use this feature

If your application expects custom roles to be passed in a SAML response, you need to use this feature. You can create as many roles as you need to be passed back from Azure AD to your application.

Create roles for an application

1. In the [Azure portal](#), in the left pane, select the **Azure Active Directory** icon.



2. Select **Enterprise applications**. Then select **All applications**.

The screenshot shows the Azure Active Directory interface for the Contoso tenant. On the left, there's a sidebar with various icons and a search bar. The main area is titled 'Enterprise applications' and shows a list of management options: Overview, Quick start, MANAGE (Users and groups, Enterprise applications, App registrations, Licenses, Azure AD Connect, Domain names, Mobility (MDM and MAM), Password reset), ACTIVITY (Sign-ins, Audit logs), and SUPPORT + TROUBLESHOOTING (New support request, Troubleshoot). The 'Enterprise applications' link in the MANAGE section is highlighted with a red box. The 'All applications' link under the same heading is also highlighted with a red box.

3. To add a new application, select the **New application** button on the top of the dialog box.



4. In the search box, type the name of your application, and then select your application from the result panel. Select the **Add** button to add the application.

A screenshot of the 'Add from the gallery' dialog box. The search bar contains the text '...' and has a red box around it. Below the search bar, a message says '1 applications matched ... Choose one below or...'. A table follows, with columns 'NAME' and 'CATEGORY'. A single row is visible, with the 'NAME' column containing '...' and the 'CATEGORY' column containing '...'. The entire row has a red box around it.

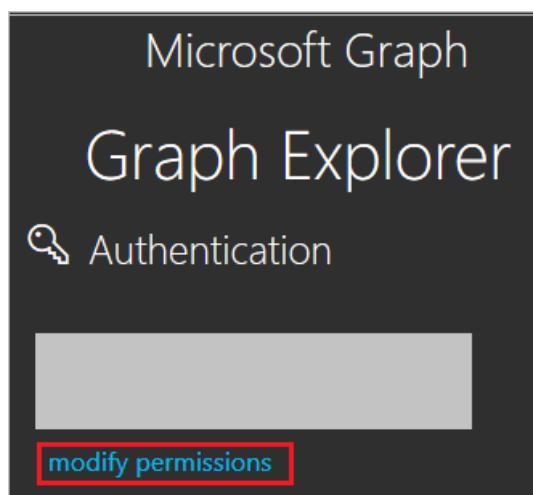
5. After the application is added, go to the **Properties** page and copy the object ID.

The screenshot shows the 'Properties' tab selected in the left sidebar under 'MANAGE'. The main area displays various configuration settings:

- Name: [REDACTED]
- Publisher: Active Directory Application Registry
- Homepage URL: [REDACTED]
- Logo: [REDACTED]
- User access URL: https://myapps.microsoft.com/signin/[REDACTED] (with copy icon)
- Application ID: [REDACTED] (with copy icon)
- Object ID: [REDACTED] (highlighted with a red box) (with copy icon)

6. Open [Microsoft Graph Explorer](#) in another window and take the following steps:

- Sign in to the Graph Explorer site by using the global admin or coadmin credentials for your tenant.
- You need sufficient permissions to create the roles. Select **modify permissions** to get the permissions.



- Select the following permissions from the list (if you don't have these already) and select **Modify Permissions**.

The screenshot shows the 'Modify Permissions' dialog with the following details:

- Selected permissions (highlighted with a red box):
 - Directory.AccessAsUser.All
 - Directory.Read.All
 - Directory.ReadWrite.All
- Other listed permissions:
 - Group.Read.All
- Note: You have selected permissions that only an administrator can grant. To get access, an administrator can grant [access to your entire organization](#).
- Buttons: Modify Permissions (highlighted with a red box) and Close.

NOTE

Cloud App Administrator and App Administrator role will not work in this scenario as we need the Global Admin permissions for Directory Read and Write.

d. Accept the consent. You're logged in to the system again.

e. Change the version to **beta**, and fetch the list of service principals from your tenant by using the following query:

```
https://graph.microsoft.com/beta/servicePrincipals
```

If you're using multiple directories, follow this pattern:

```
https://graph.microsoft.com/beta/contoso.com/servicePrincipals
```

The screenshot shows the Microsoft Graph Explorer interface. At the top, there are dropdown menus for 'GET' and 'beta', and a URL input field containing 'https://graph.microsoft.com/beta/servicePrincipals'. To the right is a 'Run Query' button. Below the input field, there are tabs for 'Request Body' and 'Request Headers', with 'Request Body' being active. The body contains a JSON object with several fields like '@odata.context', 'id', 'deletedDateTime', 'accountEnabled', etc. In the bottom left corner of the main area, there's a green status bar indicating 'Success - Status Code 200, 6146ms'. At the bottom, there are tabs for 'Response Preview' and 'Response Headers', with 'Response Preview' being active. The preview shows the JSON response, which includes an '@odata.context' field, an '@odata.nextLink' field pointing to a specific service principal, and a 'value' array containing one service principal object with fields like 'id', 'deletedDateTime', and 'accountEnabled'.

NOTE

We are already in the process of upgrading the APIs so customers might see some disruption in the service.

f. From the list of fetched service principals, get the one that you need to modify. You can also use Ctrl+F to search the application from all the listed service principals. Search for the object ID that you copied from the **Properties** page, and use the following query to get to the service principal:

```
https://graph.microsoft.com/beta/servicePrincipals/<objectID>
```

GET beta https://graph.microsoft.com/beta/servicePrincipals/8164e784-8a01-46c9-89fd-3076bd9656d0

Request Body Request Headers

```
{ "customKeyIdentifier": "SUFNX1NTMF80LzYvMjAxOCAxMjoxMT0zNyBBTQ==", "endDate": "2021-04-06T00:11:36Z", "keyId": "e101c4ec-3e89-4d3b-9ba3-4f524cb6eeda", "startDateTime": "2018-04-06T00:11:36Z", "secretText": null, "hint": null }
```

Success - Status Code 200, 312ms

Response Preview Response Headers

```
+ { "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity", "id": "8164e784-8a01-46c9-89fd-3076bd9656d0", "deletedDateTime": null, "accountEnabled": true, }
```

g. Extract the **appRoles** property from the service principal object.

```
"appRoles": [ { "allowedMemberTypes": [ "User" ], "description": "msiam_access", "displayName": "msiam_access", "id": "7dfd756e-8c27-4472-b2b7-38c17fc5de5e", "isEnabled": true, "origin": "Application", "value": null } ],
```

NOTE

If you're using the custom app (not the Azure Marketplace app), you see two default roles: user and msiam_access. For the Marketplace app, msiam_access is the only default role. You don't need to make any changes in the default roles.

h. Generate new roles for your application.

The following JSON is an example of the **appRoles** object. Create a similar object to add the roles that you want for your application.

```
{
  "appRoles": [
    {
      "allowedMemberTypes": [
        "User"
      ],
      "description": "msiam_access",
      "displayName": "msiam_access",
      "id": "b9632174-c057-4f7e-951b-be3adc52bfe6",
      "isEnabled": true,
      "origin": "Application",
      "value": null
    },
    {
      "allowedMemberTypes": [
        "User"
      ],
      "description": "Administrators Only",
      "displayName": "Admin",
      "id": "4f8f8640-f081-492d-97a0-caf24e9bc134",
      "isEnabled": true,
      "origin": "ServicePrincipal",
      "value": "Administrator"
    }
  ]
}
```

NOTE

You can only add new roles after msiam_access for the patch operation. Also, you can add as many roles as your organization needs. Azure AD will send the value of these roles as the claim value in the SAML response. To generate the GUID values for the ID of new roles use the web tools like [this](#)

- Go back to Graph Explorer and change the method from **GET** to **PATCH**. Patch the service principal object to have the desired roles by updating the **appRoles** property like the one shown in the preceding example. Select **Run Query** to execute the patch operation. A success message confirms the creation of the role.

The screenshot shows the Microsoft Graph Explorer interface. At the top, there are dropdown menus for 'PATCH' and 'beta', and a URL bar containing 'https://graph.microsoft.com/beta/servicePrincipals/8164e784-8a01-46c9-89fd-3076bd9656d0'. To the right of the URL is a blue 'Run Query' button. Below the URL bar, there are tabs for 'Request Body' and 'Request Headers'. The 'Request Body' tab is selected and contains the JSON code from the previous code block. The 'Request Headers' tab is visible but empty. At the bottom of the interface, a green status bar displays 'Success - Status Code 204, 776ms'.

- After the service principal is patched with more roles, you can assign users to the respective roles. You can assign the users by going to portal and browsing to the application. Select the **Users and groups** tab. This tab lists all the users and groups that are already assigned to the app. You can add new users on the new roles. You can also select an existing user and select **Edit** to change the role.

The screenshot shows the Azure portal's 'Users and groups' page. In the top navigation bar, there are icons for 'Add user', 'Edit', 'Remove', and 'Update Credentials'. Below the search bar, there are columns for 'DISPLAY NAME', 'OBJECT TYPE', and 'ROLE ASSIGNED'. One user, 'SSO TestUser', is listed with 'User' as both the object type and role assigned. The left sidebar has sections for 'Overview', 'Quick start', 'MANAGE', 'Properties', 'Users and groups' (which is selected and highlighted in blue), 'Single sign-on', and 'Provisioning'.

To assign the role to any user, select the new role and select the **Assign** button on the bottom of the page.

The screenshot shows two overlapping dialogs. The left dialog is titled 'Edit Assignment' and shows 'newfedssoe2etest' with '1 user selected' and 'Admin' selected. The right dialog is titled 'Select Role' and shows a list with 'Role assigned' dropdown containing 'Admin' and 'User', and a 'Select' button at the bottom. Both dialogs have a red border.

NOTE

You need to refresh your session in the Azure portal to see new roles.

8. Update the **Attributes** table to define a customized mapping of the role claim.
9. In the **User Claims** section on the **User Attributes** dialog, perform the following steps to add SAML token attribute as shown in the below table:

ATTRIBUTE NAME	ATTRIBUTE VALUE
Role name	user.assignedroles

NOTE

If the role claim value is null, then Azure AD will not send this value in the token and this is default as per design.

- a. click **Edit** icon to open **User Attributes & Claims** dialog.

2 User Attributes

givenname	user.givenname
surname	user.surname
emailaddress	user.mail
name	user.userprincipalname
Unique User Identifier	user.userprincipalname

b. In the **Manage user claims** dialog, add the SAML token attribute by clicking on **Add new claim**.

User Attributes & Claims

+ Add new claim

Manage user claims

* Name

Namespace

Source Attribute Transformation

* Source attribute

Save

c. In the **Name** box, type the attribute name as needed. This example uses **Role Name** as the claim name.

d. Leave the **Namespace** box blank.

e. From the **Source attribute** list, type the attribute value shown for that row.

f. Select **Save**.

10. To test your application in a single sign-on that's initiated by an identity provider, sign in to the [Access Panel](#) and select your application tile. In the SAML token, you should see all the assigned roles for the user with the claim name that you have given.

Update an existing role

To update an existing role, perform the following steps:

1. Open [Microsoft Graph Explorer](#).

- Sign in to the Graph Explorer site by using the global admin or coadmin credentials for your tenant.
- Change the version to **beta**, and fetch the list of service principals from your tenant by using the following query:

```
https://graph.microsoft.com/beta/servicePrincipals
```

If you're using multiple directories, follow this pattern:

```
https://graph.microsoft.com/beta/contoso.com/servicePrincipals
```

The screenshot shows the Microsoft Graph Explorer interface. The top navigation bar has 'GET' and 'beta' selected. The URL is set to `https://graph.microsoft.com/beta/servicePrincipals`. On the right, there is a red 'Run Query' button. Below the URL, there are tabs for 'Request Body' (selected) and 'Request Headers'. The 'Request Body' section contains an empty JSON object. The 'Response Preview' tab shows the results of the query, which is a list of service principals. One item is expanded to show its details: it has an ID of `00008fa9-7197-4f17-b74b-cf702e697ddc`, is not deleted, and is account-enabled.

- From the list of fetched service principals, get the one that you need to modify. You can also use Ctrl+F to search the application from all the listed service principals. Search for the object ID that you copied from the **Properties** page, and use the following query to get to the service principal:

```
https://graph.microsoft.com/beta/servicePrincipals/<objectID>
```

The screenshot shows the Microsoft Graph Explorer interface. The top navigation bar has 'GET' and 'beta' selected. The URL is set to `https://graph.microsoft.com/beta/servicePrincipals/8164e784-8a01-46c9-89fd-3076bd9656d0`. On the right, there is a red 'Run Query' button. Below the URL, there are tabs for 'Request Body' (selected) and 'Request Headers'. The 'Request Body' section contains an empty JSON object. The 'Response Preview' tab shows the results of the query, which is a single service principal object. The object has an ID of `8164e784-8a01-46c9-89fd-3076bd9656d0`, is not deleted, and is account-enabled.

- Extract the **appRoles** property from the service principal object.

```
"appRoles": [  
    {  
        "allowedMemberTypes": [  
            "User"  
        ],  
        "description": "msiam_access",  
        "displayName": "msiam_access",  
        "id": "7dfd756e-8c27-4472-b2b7-38c17fc5de5e",  
        "isEnabled": true,  
        "origin": "Application",  
        "value": null  
    },
```

6. To update the existing role, use the following steps.

The screenshot shows the Microsoft Graph Explorer interface. The top bar has 'PATCH' selected, 'beta' as the version, and the URL 'https://graph.microsoft.com/beta/servicePrincipals/'. A red box highlights the 'Run Query' button. Below the URL is a 'Request Body' tab. The body contains a JSON patch document:

```
{"appRoles": [ { "allowedMemberTypes": [ "User" ], "description": "student", "displayName": "student", "id": "b31a14ab-5319-4bb1-a668-5626904561be", "isEnabled": true, "origin": "ServicePrincipal" } ]}
```

- Change the method from **GET** to **PATCH**.
- Copy the existing roles and paste them under **Request Body**.
- Update the value of a role by updating the role description, role value, or role display name as needed.
- After you update all the required roles, select **Run Query**.

Delete an existing role

To delete an existing role, perform the following steps:

- Open [Microsoft Graph Explorer](#) in another window.
- Sign in to the Graph Explorer site by using the global admin or coadmin credentials for your tenant.
- Change the version to **beta**, and fetch the list of service principals from your tenant by using the following query:

```
https://graph.microsoft.com/beta/servicePrincipals
```

If you're using multiple directories, follow this pattern:

```
https://graph.microsoft.com/beta/contoso.com/servicePrincipals
```

Request Body

```
{
  "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity",
  "id": "8164e784-8a01-46c9-89fd-3076bd9656d0",
  "deletedDateTime": null,
  "accountEnabled": true,
  "appDisplayName": "",
  "appId": "8e1d26f3-9519-4d66-8577-65fa3261c7ff",
  "appOwnerOrganizationId": "0ac53016-3006-4227-9eeb-89d63f8055b6",
  "appRoleAssignmentRequired": true,
  "displayName": ""
}
```

Response Preview

```
{
  "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals",
  "@odata.nextLink": "https://graph.microsoft.com/beta/servicePrincipals?$skiptoken=X%274453707402000100000035536572766963655072696E636970616C5F31643466396561652D32&$top=1",
  "value": [
    {
      "id": "00008fa9-7197-4f17-b74b-cf702e697ddc",
      "deletedDateTime": null,
      "accountEnabled": true
    }
  ]
}
```

- From the list of fetched service principals, get the one that you need to modify. You can also use Ctrl+F to search the application from all the listed service principals. Search for the object ID that you copied from the **Properties** page, and use the following query to get to the service principal:

Request Body

```
{
  "customKeyIdentifier": "SUFNX1NTMF80LzYvMjAxOCAxMjoxMTozNyBBTQ==",
  "endDateTime": "2021-04-06T00:11:36Z",
  "keyId": "e101c4ec-3e89-4d3b-9ba3-4f524cb6eeda",
  "startDateTime": "2018-04-06T00:11:36Z",
  "secretText": null,
  "hint": null
}
```

Response Preview

```
{
  "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity",
  "id": "8164e784-8a01-46c9-89fd-3076bd9656d0",
  "deletedDateTime": null,
  "accountEnabled": true
}
```

- Extract the **appRoles** property from the service principal object.

```
{
  "allowedMemberTypes": [
    "User"
  ],
  "description": "msiam_access",
  "displayName": "msiam_access",
  "id": "b9632174-c057-4f7e-951b-be3adc52bfe6",
  "isEnabled": true,
  "origin": "Application",
  "value": null
},
{
  "allowedMemberTypes": [
    "User"
  ],
  "description": "Administrators Only",
  "displayName": "Admin",
  "id": "4f8f8640-f081-492d-97a0-caf24e9bc134",
  "isEnabled": true,
  "origin": "ServicePrincipal",
  "value": "Administrator"
}
}
```

6. To delete the existing role, use the following steps.

The screenshot shows the Microsoft Graph Explorer interface. The method dropdown is set to **PATCH**, the API version dropdown is set to **beta**, and the URL is **https://graph.microsoft.com/beta/servicePrincipals/8164e784-8a01-46c9-89fd-3076bd9656d0**. The **Run Query** button is highlighted with a red box. Below the URL, there are two tabs: **Request Body** and **Request Headers**. The **Request Body** tab is selected and contains the JSON code from the previous code block. In the JSON, the **isEnabled** field for the second role object is highlighted with a red box and has its value changed to **false**.

- Change the method from **GET** to **PATCH**.
- Copy the existing roles from the application and paste them under **Request Body**.
- Set the **Enabled** value to **false** for the role that you want to delete.
- Select **Run Query**.

NOTE

Make sure that you have the msiam_access role, and the ID is matching in the generated role.

7. After the role is disabled, delete that role block from the **appRoles** section. Keep the method as **PATCH**, and select **Run Query**.
8. After you run the query, the role is deleted.

NOTE

The role needs to be disabled before it can be removed.

Next steps

For additional steps, see the [app documentation](#).

How to: customize claims issued in the SAML token for enterprise applications

10/24/2019 • 10 minutes to read • [Edit Online](#)

Today, Azure Active Directory (Azure AD) supports single sign-on (SSO) with most enterprise applications, including both applications pre-integrated in the Azure AD app gallery as well as custom applications. When a user authenticates to an application through Azure AD using the SAML 2.0 protocol, Azure AD sends a token to the application (via an HTTP POST). And then, the application validates and uses the token to log the user in instead of prompting for a username and password. These SAML tokens contain pieces of information about the user known as *claims*.

A *claim* is information that an identity provider states about a user inside the token they issue for that user. In [SAML token](#), this data is typically contained in the SAML Attribute Statement. The user's unique ID is typically represented in the SAML Subject also called as Name Identifier.

By default, Azure AD issues a SAML token to your application that contains a `NameIdentifier` claim with a value of the user's username (also known as the user principal name) in Azure AD, which can uniquely identify the user. The SAML token also contains additional claims containing the user's email address, first name, and last name.

To view or edit the claims issued in the SAML token to the application, open the application in Azure portal. Then open the **User Attributes & Claims** section.

The screenshot shows the Azure portal interface for configuring SAML-based Single Sign-On. At the top, there are three buttons: 'Change single sign-on mode', 'Switch to the old experience', and 'Test this application'. Below these is a purple banner with a rocket icon and the text 'Welcome to the new experience for configuring SAML based SSO. Please click here to provide feedback.' A blue numbered '1' indicates the 'Basic SAML Configuration' section, which includes fields for Reply URL, Identifier, Sign on URL, and Relay State. A blue numbered '2' indicates the 'User Attributes & Claims' section, which lists mappings between user attributes and SAML claim URIs. Both sections have a pencil icon in the top right corner for editing.

Set up Single Sign-On with SAML - Preview

Read the [configuration guide](#) for help integrating Atlassian Cloud.

1 Basic SAML Configuration

Reply URL (Assertion Consumer Service URL)	https://id.atlassian.com/login
Identifier (Entity ID)	https://tt--id.atlassian.com/login
Sign on URL	<i>Optional</i>
Relay State	<i>Optional</i>

2 User Attributes & Claims

Givenname	user.givenname
Surname	user.surname
Emailaddress	user.mail
Name	user.userprincipalname
Unique User Identifier	user.userprincipalname

There are two possible reasons why you might need to edit the claims issued in the SAML token:

- The application requires the `NameIdentifier` or `NameID` claim to be something other than the username (or user principal name) stored in Azure AD.
- The application has been written to require a different set of claim URIs or claim values.

Editing nameID

To edit the NameID (name identifier value):

1. Open the **Name identifier value** page.
2. Select the attribute or transformation you want to apply to the attribute. Optionally, you can specify the format you want the NameID claim to have.

Manage claim

Save Discard changes

* Name	nameidentifier
Namespace	http://schemas.xmlsoap.org/ws/2005/05/identity/cla...
▼ Choose name identifier format	
* Source	<input checked="" type="radio"/> Attribute <input type="radio"/> Transformation
* Source attribute	user.employeeid
▼ Claim Conditions	

NameID format

If the SAML request contains the element NameIDPolicy with a specific format, then Azure AD will honor the format in the request.

If the SAML request doesn't contain an element for NameIDPolicy, then Azure AD will issue the NameID with the format you specify. If no format is specified Azure AD will use the default source format associated with the claim source selected.

From the **Choose name identifier format** dropdown, you can select one of the following options.

NAMEID FORMAT	DESCRIPTION
Default	Azure AD will use the default source format.
Persistent	Azure AD will use Persistent as the NameID format.
EmailAddress	Azure AD will use EmailAddress as the NameID format.
Unspecified	Azure AD will use Unspecified as the NameID format.

Transient NameID is also supported, but is not available in the dropdown and cannot be configured on Azure's side. To learn more about the NameIDPolicy attribute, see [Single Sign-On SAML protocol](#).

Attributes

Select the desired source for the **NameIdentifier** (or NameID) claim. You can select from the following options.

NAME	DESCRIPTION
Email	Email address of the user
userprincipalName	User principal name (UPN) of the user
onpremisessamaccount	SAM account name that has been synced from on-premises Azure AD
objectid	Objectid of the user in Azure AD
employeeid	Employee ID of the user
Directory extensions	Directory extensions synced from on-premises Active Directory using Azure AD Connect Sync
Extension Attributes 1-15	On-premises extension attributes used to extend the Azure AD schema

For more info, see [Table 3: Valid ID values per source](#).

You can also assign any constant (static) value to any claims which you define in Azure AD. Please follow the below steps to assign a constant value:

1. In the [Azure portal](#), on the **User Attributes & Claims** section, click on the **Edit** icon to edit the claims.
2. Click on the required claim which you want to modify.
3. Enter the constant value without quotes in the **Source attribute** as per your organization and click **Save**.

Manage claim

Save
 Discard changes

* Name	OrganizationID ✓
Namespace	http://schemas.xmlsoap.org/ws/2005...
* Source	<input checked="" type="radio"/> Attribute <input type="radio"/> Transformation
* Source attribute	"Contoso1234"
Claim conditions	

4. The constant value will be displayed as below.

2

User Attributes & Claims	
Givenname	user.givenname
Surname	user.surname
Emailaddress	user.mail
Name	user.userprincipalname
OrganizationID	"Contoso1234"
Unique User Identifier	user.userprincipalname

Special claims - transformations

You can also use the claims transformations functions.

FUNCTION	DESCRIPTION
ExtractMailPrefix()	Removes the domain suffix from either the email address or the user principal name. This extracts only the first part of the user name being passed through (for example, "joe_smith" instead of joe_smith@contoso.com).
Join()	Joins an attribute with a verified domain. If the selected user identifier value has a domain, it will extract the username to append the selected verified domain. For example, if you select the email (joe_smith@contoso.com) as the user identifier value and select contoso.onmicrosoft.com as the verified domain, this will result in joe_smith@contoso.onmicrosoft.com.
ToLower()	Converts the characters of the selected attribute into lowercase characters.
ToUpper()	Converts the characters of the selected attribute into uppercase characters.

Adding application-specific claims

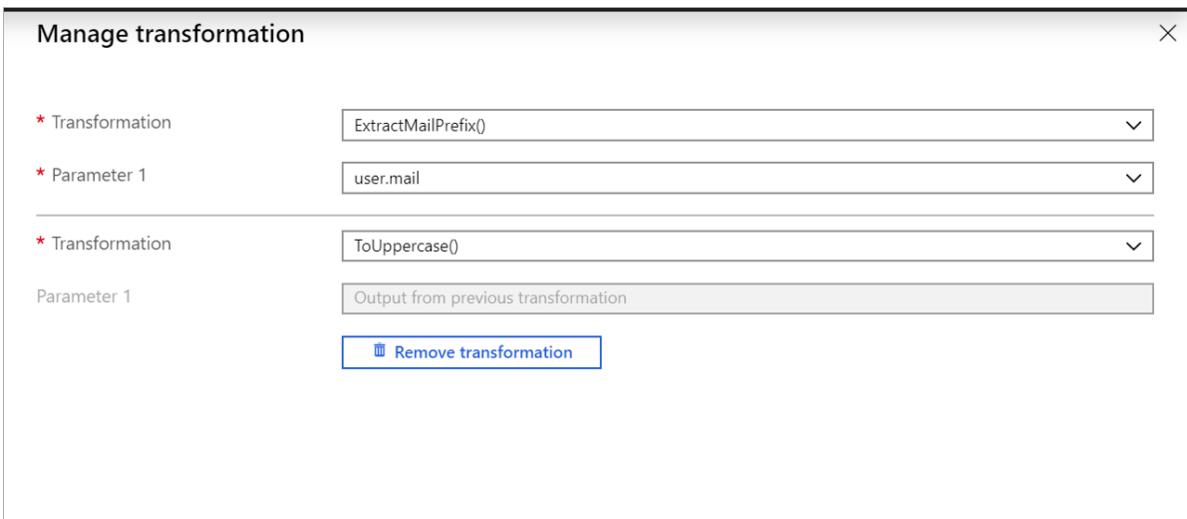
To add application-specific claims:

1. In **User Attributes & Claims**, select **Add new claim** to open the **Manage user claims** page.
2. Enter the **name** of the claims. The value doesn't strictly need to follow a URI pattern, per the SAML spec. If you need a URI pattern, you can put that in the **Namespace** field.
3. Select the **Source** where the claim is going to retrieve its value. You can select a user attribute from the source attribute dropdown or apply a transformation to the user attribute before emitting it as a claim.

Claim transformations

To apply a transformation to a user attribute:

1. In **Manage claim**, select *Transformation* as the claim source to open the **Manage transformation** page.
2. Select the function from the transformation dropdown. Depending on the function selected, you will have to provide parameters and a constant value to evaluate in the transformation. Refer to the table below for more information about the available functions.
3. To apply multiple transformation, click on **Add transformation**. You can apply a maximum of two transformation to a claim. For example, you could first extract the email prefix of the `user.mail`. Then, make the string upper case.



You can use the following functions to transform claims.

FUNCTION	DESCRIPTION
ExtractMailPrefix()	Removes the domain suffix from either the email address or the user principal name. This extracts only the first part of the user name being passed through (for example, "joe_smith" instead of joe_smith@contoso.com).
Join()	Creates a new value by joining two attributes. Optionally, you can use a separator between the two attributes. For NameID claim transformation, the join is restricted to a verified domain. If the selected user identifier value has a domain, it will extract the username to append the selected verified domain. For example, if you select the email (joe_smith@contoso.com) as the user identifier value and select contoso.onmicrosoft.com as the verified domain, this will result in joe_smith@contoso.onmicrosoft.com.
ToLower()	Converts the characters of the selected attribute into lowercase characters.
ToUpper()	Converts the characters of the selected attribute into uppercase characters.
Contains()	Outputs an attribute or constant if the input matches the specified value. Otherwise, you can specify another output if there's no match. For example, if you want to emit a claim where the value is the user's email address if it contains the domain "@contoso.com", otherwise you want to output the user principal name. To do this, you would configure the following values: <i>Parameter 1 (input): user.email Value: "@contoso.com"</i> <i>Parameter 2 (output): user.email</i> <i>Parameter 3 (output if there's no match): user.userprincipalname</i>

FUNCTION	DESCRIPTION
EndWith()	<p>Outputs an attribute or constant if the input ends with the specified value. Otherwise, you can specify another output if there's no match.</p> <p>For example, if you want to emit a claim where the value is the user's employee ID if the employee ID ends with "000", otherwise you want to output an extension attribute. To do this, you would configure the following values:</p> <p><i>Parameter 1(input):</i> user.employeeid <i>Value:</i> "000"</p> <p><i>Parameter 2 (output):</i> user.employeeid <i>Parameter 3 (output if there's no match):</i> user.extensionattribute1</p>
StartWith()	<p>Outputs an attribute or constant if the input starts with the specified value. Otherwise, you can specify another output if there's no match.</p> <p>For example, if you want to emit a claim where the value is the user's employee ID if the country/region starts with "US", otherwise you want to output an extension attribute. To do this, you would configure the following values:</p> <p><i>Parameter 1(input):</i> user.country <i>Value:</i> "US"</p> <p><i>Parameter 2 (output):</i> user.employeeid <i>Parameter 3 (output if there's no match):</i> user.extensionattribute1</p>
Extract() - After matching	<p>Returns the substring after it matches the specified value. For example, if the input's value is "Finance_BSimon", the matching value is "Finance_ ", then the claim's output is "BSimon".</p>
Extract() - Before matching	<p>Returns the substring until it matches the specified value. For example, if the input's value is "BSimon_US", the matching value is "_US", then the claim's output is "BSimon".</p>
Extract() - Between matching	<p>Returns the substring until it matches the specified value. For example, if the input's value is "Finance_BSimon_US", the first matching value is "Finance_ ", the second matching value is "_US", then the claim's output is "BSimon".</p>
ExtractAlpha() - Prefix	<p>Returns the prefix alphabetical part of the string. For example, if the input's value is "BSimon_123", then it returns "BSimon".</p>
ExtractAlpha() - Suffix	<p>Returns the suffix alphabetical part of the string. For example, if the input's value is "123_Simon", then it returns "Simon".</p>
ExtractNumeric() - Prefix	<p>Returns the prefix numerical part of the string. For example, if the input's value is "123_BSimon", then it returns "123".</p>
ExtractNumeric() - Suffix	<p>Returns the suffix numerical part of the string. For example, if the input's value is "BSimon_123", then it returns "123".</p>

FUNCTION	DESCRIPTION
IfEmpty()	Outputs an attribute or constant if the input is null or empty. For example, if you want to output an attribute stored in an extensionattribute if the employee ID for a given user is empty. To do this, you would configure the following values: Parameter 1(input): user.employeeid Parameter 2 (output): user.extensionattribute1 Parameter 3 (output if there's no match): user.employeeid
IfNotEmpty()	Outputs an attribute or constant if the input is not null or empty. For example, if you want to output an attribute stored in an extensionattribute if the employee ID for a given user is not empty. To do this, you would configure the following values: Parameter 1(input): user.employeeid Parameter 2 (output): user.extensionattribute1

If you need additional transformations, submit your idea in the [feedback forum in Azure AD](#) under the *SaaS application* category.

Emitting claims based on conditions

You can specify the source of a claim based on user type and the group to which the user belongs.

The user type can be:

- **Any:** All users are allowed to access the application.
- **Members:** Native member of the tenant
- **All guests:** User is brought over from an external organization with or without Azure AD.
- **AAD guests:** Guest user belongs to another organization using Azure AD.
- **External guests:** Guest user belongs to an external organization that doesn't have Azure AD.

One scenario where this is helpful is when the source of a claim is different for a guest and an employee accessing an application. You may want to specify that if the user is an employee the NamelD is sourced from user.email, but if the user is a guest then the NamelD is sourced from user.extensionattribute1.

To add a claim condition:

1. In **Manage claim**, expand the Claim conditions.
2. Select the user type.
3. Select the group(s) to which the user should belong. You can select up to 10 unique groups across all claims for a given application.
4. Select the **Source** where the claim is going to retrieve its value. You can select a user attribute from the source attribute dropdown or apply a transformation to the user attribute before emitting it as a claim.

The order in which you add the conditions are important. Azure AD evaluates the conditions from top to bottom to decide which value to emit in the claim.

For example, Brita Simon is a guest user in the Contoso tenant. She belongs to another organization that also uses Azure AD. Given the below configuration for the Fabrikam application, when Brita tries to sign in to Fabrikam, Azure AD will evaluate the conditions as follow.

First, Azure AD verifies if Brita's user type is `All guests`. Since, this is true then Azure AD assigns the source for the claim to `user.extensionattribute1`. Second, Azure AD verifies if Brita's user type is `AAD guests`, since this is also true then Azure AD assigns the source for the claim to `user.mail`. Finally, the claim is emitted with value `user.email` for Brita.

Manage claim

 Save  Discard changes

* Name

Namespace

* Source Attribute Transformation

* Source attribute

^ Claim Conditions

Returns the claim only if all the conditions below are met.



Multiple conditions can be applied to a claim. When adding conditions, order of operation is important. [Read the documentation](#) for more information.

USER TYPE	SCOPED GROUPS	SOURCE	VALUE	...
All Guests	0 groups	Attribute	user.extensionattribute1	...
AAD Guests	0 groups	Attribute	user.mail	...
Select from drop down	Select Groups	<input type="radio"/> Attribute <input type="radio"/> Transformation		

Next steps

- [Application management in Azure AD](#)
- [Configure single sign-on on applications that are not in the Azure AD application gallery](#)
- [Troubleshoot SAML-based single sign-on](#)

How to: Customize claims emitted in tokens for a specific app in a tenant (Preview)

10/23/2019 • 13 minutes to read • [Edit Online](#)

NOTE

This feature replaces and supersedes the [claims customization](#) offered through the portal today. On the same application, if you customize claims using the portal in addition to the Graph/PowerShell method detailed in this document, tokens issued for that application will ignore the configuration in the portal. Configurations made through the methods detailed in this document will not be reflected in the portal.

This feature is used by tenant admins to customize the claims emitted in tokens for a specific application in their tenant. You can use claims-mapping policies to:

- Select which claims are included in tokens.
- Create claim types that do not already exist.
- Choose or change the source of data emitted in specific claims.

NOTE

This capability currently is in public preview. Be prepared to revert or remove any changes. The feature is available in any Azure Active Directory (Azure AD) subscription during public preview. However, when the feature becomes generally available, some aspects of the feature might require an Azure AD premium subscription. This feature supports configuring claim mapping policies for WS-Fed, SAML, OAuth, and OpenID Connect protocols.

Claims mapping policy type

In Azure AD, a **Policy** object represents a set of rules enforced on individual applications or on all applications in an organization. Each type of policy has a unique structure, with a set of properties that are then applied to objects to which they are assigned.

A claims mapping policy is a type of **Policy** object that modifies the claims emitted in tokens issued for specific applications.

Claim sets

There are certain sets of claims that define how and when they're used in tokens.

CLAIM SET	DESCRIPTION
Core claim set	Are present in every token regardless of the policy. These claims are also considered restricted, and can't be modified.
Basic claim set	Includes the claims that are emitted by default for tokens (in addition to the core claim set). You can omit or modify basic claims by using the claims mapping policies.

CLAIM SET	DESCRIPTION
Restricted claim set	Can't be modified using policy. The data source cannot be changed, and no transformation is applied when generating these claims.

Table 1: JSON Web Token (JWT) restricted claim set

CLAIM TYPE (NAME)
_claim_names
_claim_sources
access_token
account_type
acr
actor
actortoken
aio
altsecid
amr
app_chain
app_displayname
app_res
appctx
appctxsender
appid
appidacr
assertion
at_hash
aud
auth_data
auth_time

CLAIM TYPE (NAME)
authorization_code
azp
azpacr
c_hash
ca_enf
cc
cert_token_use
client_id
cloud_graph_host_name
cloud_instance_name
cnf
code
controls
credential_keys
csr
csr_type
deviceid
dns_names
domain_dns_name
domain_netbios_name
e_exp
email
endpoint
enfpolids
exp

CLAIM TYPE (NAME)

expires_on

grant_type

graph

group_sids

groups

hasgroups

hash_alg

home_oid

`http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationinstant``http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationmethod``http://schemas.microsoft.com/ws/2008/06/identity/claims/expiration``http://schemas.microsoft.com/ws/2008/06/identity/claims/expired``http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress``http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name``http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier`

iat

identityprovider

idp

in_corp

instance

ipaddr

isbrowserhostedapp

iss

jwk

key_id

CLAIM TYPE (NAME)

key_type

mam_compliance_url

mam_enrollment_url

mam_terms_of_use_url

mdm_compliance_url

mdm_enrollment_url

mdm_terms_of_use_url

nameid

nbf

netbios_name

nonce

oid

on_prem_id

onprem_sam_account_name

onprem_sid

openid2_id

password

platf

polids

pop_jwk

preferred_username

previous_refresh_token

primary_sid

puid

pwd_exp

CLAIM TYPE (NAME)
pwd_url
redirect_uri
refresh_token
refreshtoken
request_nonce
resource
role
roles
scope
scp
sid
signature
signin_state
src1
src2
sub
tbid
tenant_display_name
tenant_region_scope
thumbnail_photo
tid
tokenAutologonEnabled
trustedfordelegation
unique_name
upn

CLAIM TYPE (NAME)

user_setting_sync_url

username

uti

ver

verified_primary_email

verified_secondary_email

wids

win_ver

Table 2: SAML restricted claim set**CLAIM TYPE (URI)**`http://schemas.microsoft.com/ws/2008/06/identity/claims/expiration``http://schemas.microsoft.com/ws/2008/06/identity/claims/expired``http://schemas.microsoft.com/identity/claims/accesstoken``http://schemas.microsoft.com/identity/claims/openid2_id``http://schemas.microsoft.com/identity/claims/identityprovider``http://schemas.microsoft.com/identity/claims/objectidentifier``http://schemas.microsoft.com/identity/claims/puid``http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier [MR1]``http://schemas.microsoft.com/identity/claims/tenantid``http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationinstant``http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationmethod``http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider``http://schemas.microsoft.com/ws/2008/06/identity/claims/groups``http://schemas.microsoft.com/claims/groups.link``http://schemas.microsoft.com/ws/2008/06/identity/claims/role`

CLAIM TYPE (URI)

`http://schemas.microsoft.com/ws/2008/06/identity/claims/wids`

`http://schemas.microsoft.com/2014/09/devicecontext/claims/iscompliant`

`http://schemas.microsoft.com/2014/02/devicecontext/claims/isknown`

`http://schemas.microsoft.com/2012/01/devicecontext/claims/ismanaged`

`http://schemas.microsoft.com/2014/03/pss0`

`http://schemas.microsoft.com/claims/authnmethodsreferences`

`http://schemas.xmlsoap.org/ws/2009/09/identity/claims/actor`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/samlissuername`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/confirmationkey`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsaccountname`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/primarygroupsid`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/authorizationdecision`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/authentication`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/sid`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/denyonlyprimarygroupsid`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/denyonlyprimarysid`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/denyonlysid`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/denyonlywindowsdevicegroup`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsdeviceclaim`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsdevicegroup`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsfqnversion`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowssubauthority`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsuserclaim`

CLAIM TYPE (URI)
<code>http://schemas.xmlsoap.org/ws/2005/05/identity/claims/x500distinguishedname</code>
<code>http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn</code>
<code>http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid</code>
<code>http://schemas.xmlsoap.org/ws/2005/05/identity/claims/spn</code>
<code>http://schemas.microsoft.com/ws/2008/06/identity/claims/ispersistent</code>
<code>http://schemas.xmlsoap.org/ws/2005/05/identity/claims/privatepersonalidentifier</code>
<code>http://schemas.microsoft.com/identity/claims/scope</code>

Claims mapping policy properties

To control what claims are emitted and where the data comes from, use the properties of a claims mapping policy. If a policy is not set, the system issues tokens that include the core claim set, the basic claim set, and any [optional claims](#) that the application has chosen to receive.

Include basic claim set

String: IncludeBasicClaimSet

Data type: Boolean (True or False)

Summary: This property determines whether the basic claim set is included in tokens affected by this policy.

- If set to True, all claims in the basic claim set are emitted in tokens affected by the policy.
- If set to False, claims in the basic claim set are not in the tokens, unless they are individually added in the claims schema property of the same policy.

NOTE

Claims in the core claim set are present in every token, regardless of what this property is set to.

Claims schema

String: ClaimsSchema

Data type: JSON blob with one or more claim schema entries

Summary: This property defines which claims are present in the tokens affected by the policy, in addition to the basic claim set and the core claim set. For each claim schema entry defined in this property, certain information is required. Specify where the data is coming from (**Value** or **Source/ID pair**), and which claim the data is emitted as (**Claim Type**).

Claim schema entry elements

Value: The Value element defines a static value as the data to be emitted in the claim.

Source/ID pair: The Source and ID elements define where the data in the claim is sourced from.

Set the Source element to one of the following values:

- "user": The data in the claim is a property on the User object.

- "application": The data in the claim is a property on the application (client) service principal.
- "resource": The data in the claim is a property on the resource service principal.
- "audience": The data in the claim is a property on the service principal that is the audience of the token (either the client or resource service principal).
- "company": The data in the claim is a property on the resource tenant's Company object.
- "transformation": The data in the claim is from claims transformation (see the "Claims transformation" section later in this article).

If the source is transformation, the **TransformationID** element must be included in this claim definition as well.

The ID element identifies which property on the source provides the value for the claim. The following table lists the values of ID valid for each value of Source.

Table 3: Valid ID values per source

SOURCE	ID	DESCRIPTION
User	surname	Family Name
User	givenname	Given Name
User	displayname	Display Name
User	objectid	ObjectID
User	mail	Email Address
User	userprincipalname	User Principal Name
User	department	Department
User	onpremisesamaccountname	On-premises SAM Account Name
User	netbiosname	NetBios Name
User	dnsdomainname	DNS Domain Name
User	onpremisesecurityidentifier	On-premises Security Identifier
User	companyname	Organization Name
User	streetaddress	Street Address
User	postalcode	Postal Code
User	preferredlanguage	Preferred Language
User	onpremisesuserprincipalname	On-premises UPN
User	mailnickname	Mail Nickname
User	extensionattribute1	Extension Attribute 1

SOURCE	ID	DESCRIPTION
User	extensionattribute2	Extension Attribute 2
User	extensionattribute3	Extension Attribute 3
User	extensionattribute4	Extension Attribute 4
User	extensionattribute5	Extension Attribute 5
User	extensionattribute6	Extension Attribute 6
User	extensionattribute7	Extension Attribute 7
User	extensionattribute8	Extension Attribute 8
User	extensionattribute9	Extension Attribute 9
User	extensionattribute10	Extension Attribute 10
User	extensionattribute11	Extension Attribute 11
User	extensionattribute12	Extension Attribute 12
User	extensionattribute13	Extension Attribute 13
User	extensionattribute14	Extension Attribute 14
User	extensionattribute15	Extension Attribute 15
User	othermail	Other Mail
User	country	Country
User	city	City
User	state	State
User	jobtitle	Job Title
User	employeeid	Employee ID
User	facsimiletelephonenumber	Facsimile Telephone Number
application, resource, audience	displayname	Display Name
application, resource, audience	objected	ObjectID
application, resource, audience	tags	Service Principal Tag
Company	tenantcountry	Tenant's country

TransformationID: The TransformationID element must be provided only if the Source element is set to

"transformation".

- This element must match the ID element of the transformation entry in the **ClaimsTransformation** property that defines how the data for this claim is generated.

Claim Type: The **JwtClaimType** and **SamlClaimType** elements define which claim this claim schema entry refers to.

- The JwtClaimType must contain the name of the claim to be emitted in JWTs.
- The SamlClaimType must contain the URI of the claim to be emitted in SAML tokens.

NOTE

Names and URIs of claims in the restricted claim set cannot be used for the claim type elements. For more information, see the "Exceptions and restrictions" section later in this article.

Claims transformation

String: ClaimsTransformation

Data type: JSON blob, with one or more transformation entries

Summary: Use this property to apply common transformations to source data, to generate the output data for claims specified in the Claims Schema.

ID: Use the ID element to reference this transformation entry in the TransformationID Claims Schema entry. This value must be unique for each transformation entry within this policy.

TransformationMethod: The TransformationMethod element identifies which operation is performed to generate the data for the claim.

Based on the method chosen, a set of inputs and outputs is expected. Define the inputs and outputs by using the **InputClaims**, **InputParameters** and **OutputClaims** elements.

Table 4: Transformation methods and expected inputs and outputs

TRANSFORMATIONMETHOD	EXPECTED INPUT	EXPECTED OUTPUT	DESCRIPTION
Join	string1, string2, separator	outputClaim	Joins input strings by using a separator in between. For example: string1:"foo@bar.com" , string2:"sandbox" , separator:"." results in outputClaim:"foo@bar.com.s andbox"
ExtractMailPrefix	mail	outputClaim	Extracts the local part of an email address. For example: mail:"foo@bar.com" results in outputClaim:"foo". If no @ sign is present, then the original input string is returned as is.

InputClaims: Use an InputClaims element to pass the data from a claim schema entry to a transformation. It has two attributes: **ClaimTypeReferenceId** and **TransformationClaimType**.

- **ClaimTypeReferenceId** is joined with ID element of the claim schema entry to find the appropriate input claim.

- **TransformationClaimType** is used to give a unique name to this input. This name must match one of the expected inputs for the transformation method.

InputParameters: Use an InputParameters element to pass a constant value to a transformation. It has two attributes: **Value** and **ID**.

- **Value** is the actual constant value to be passed.
- **ID** is used to give a unique name to the input. The name must match one of the expected inputs for the transformation method.

OutputClaims: Use an OutputClaims element to hold the data generated by a transformation, and tie it to a claim schema entry. It has two attributes: **ClaimTypeReferenceId** and **TransformationClaimType**.

- **ClaimTypeReferenceId** is joined with the ID of the claim schema entry to find the appropriate output claim.
- **TransformationClaimType** is used to give a unique name to the output. The name must match one of the expected outputs for the transformation method.

Exceptions and restrictions

SAML NameID and UPN: The attributes from which you source the NameID and UPN values, and the claims transformations that are permitted, are limited. See table 5 and table 6 to see the permitted values.

Table 5: Attributes allowed as a data source for SAML NameID

SOURCE	ID	DESCRIPTION
User	mail	Email Address
User	userprincipalname	User Principal Name
User	onpremisesamaccountname	On Premises Sam Account Name
User	employeeid	Employee ID
User	extensionattribute1	Extension Attribute 1
User	extensionattribute2	Extension Attribute 2
User	extensionattribute3	Extension Attribute 3
User	extensionattribute4	Extension Attribute 4
User	extensionattribute5	Extension Attribute 5
User	extensionattribute6	Extension Attribute 6
User	extensionattribute7	Extension Attribute 7
User	extensionattribute8	Extension Attribute 8
User	extensionattribute9	Extension Attribute 9
User	extensionattribute10	Extension Attribute 10
User	extensionattribute11	Extension Attribute 11

SOURCE	ID	DESCRIPTION
User	extensionattribute12	Extension Attribute 12
User	extensionattribute13	Extension Attribute 13
User	extensionattribute14	Extension Attribute 14
User	extensionattribute15	Extension Attribute 15

Table 6: Transformation methods allowed for SAML NameID

TRANSFORMATIONMETHOD	RESTRICTIONS
ExtractMailPrefix	None
Join	The suffix being joined must be a verified domain of the resource tenant.

Custom signing key

A custom signing key must be assigned to the service principal object for a claims mapping policy to take effect. This ensures acknowledgment that tokens have been modified by the creator of the claims mapping policy and protects applications from claims mapping policies created by malicious actors. Apps that have claims mapping enabled must check a special URI for their token signing keys by appending `appid={client_id}` to their [OpenID Connect metadata requests](#).

Cross-tenant scenarios

Claims mapping policies do not apply to guest users. If a guest user tries to access an application with a claims mapping policy assigned to its service principal, the default token is issued (the policy has no effect).

Claims mapping policy assignment

Claims mapping policies can only be assigned to service principal objects.

Example claims mapping policies

In Azure AD, many scenarios are possible when you can customize claims emitted in tokens for specific service principals. In this section, we walk through a few common scenarios that can help you grasp how to use the claims mapping policy type.

Prerequisites

In the following examples, you create, update, link, and delete policies for service principals. If you are new to Azure AD, we recommend that you [learn about how to get an Azure AD tenant](#) before you proceed with these examples.

To get started, do the following steps:

1. Download the latest [Azure AD PowerShell Module public preview release](#).
2. Run the Connect command to sign in to your Azure AD admin account. Run this command each time you start a new session.

```
Connect-AzureAD -Confirm
```

3. To see all policies that have been created in your organization, run the following command. We recommend that you run this command after most operations in the following scenarios, to check that your

policies are being created as expected.

```
Get-AzureADPolicy
```

Example: Create and assign a policy to omit the basic claims from tokens issued to a service principal

In this example, you create a policy that removes the basic claim set from tokens issued to linked service principals.

1. Create a claims mapping policy. This policy, linked to specific service principals, removes the basic claim set from tokens.

- a. To create the policy, run this command:

```
New-AzureADPolicy -Definition @('{"ClaimsMappingPolicy":  
{"Version":1,"IncludeBasicClaimSet":"false"}") -DisplayName "OmitBasicClaims" -Type  
"ClaimsMappingPolicy"
```

- b. To see your new policy, and to get the policy ObjectId, run the following command:

```
Get-AzureADPolicy
```

2. Assign the policy to your service principal. You also need to get the ObjectId of your service principal.

- a. To see all your organization's service principals, you can [query Microsoft Graph](#). Or, in [Graph Explorer](#), sign in to your Azure AD account.

- b. When you have the ObjectId of your service principal, run the following command:

```
Add-AzureADServicePrincipalPolicy -Id <ObjectId of the ServicePrincipal> -RefObjectId <ObjectId  
of the Policy>
```

Example: Create and assign a policy to include the EmployeeID and TenantCountry as claims in tokens issued to a service principal

In this example, you create a policy that adds the EmployeeID and TenantCountry to tokens issued to linked service principals. The EmployeeID is emitted as the name claim type in both SAML tokens and JWTs. The TenantCountry is emitted as the country claim type in both SAML tokens and JWTs. In this example, we continue to include the basic claims set in the tokens.

1. Create a claims mapping policy. This policy, linked to specific service principals, adds the EmployeeID and TenantCountry claims to tokens.

- a. To create the policy, run the following command:

```
New-AzureADPolicy -Definition @('{"ClaimsMappingPolicy":  
{"Version":1,"IncludeBasicClaimSet":"true", "ClaimsSchema":  
[{"Source":"user","ID":"employeeid","SamlClaimType":"http://schemas.xmlsoap.org/ws/2005/05/ident  
ity/claims/name","JwtClaimType":"name"},  
 {"Source":"company","ID":"tenantcountry","SamlClaimType":"http://schemas.xmlsoap.org/ws/2005/05/  
identity/claims/country","JwtClaimType":"country"}]}') -DisplayName "ExtraClaimsExample" -Type  
"ClaimsMappingPolicy"
```

- b. To see your new policy, and to get the policy ObjectId, run the following command:

```
Get-AzureADPolicy
```

2. Assign the policy to your service principal. You also need to get the ObjectId of your service principal.

a. To see all your organization's service principals, you can [query Microsoft Graph](#). Or, in [Graph Explorer](#), sign in to your Azure AD account.

b. When you have the ObjectId of your service principal, run the following command:

```
Add-AzureADServicePrincipalPolicy -Id <ObjectId of the ServicePrincipal> -RefObjectId <ObjectId of the Policy>
```

Example: Create and assign a policy that uses a claims transformation in tokens issued to a service principal

In this example, you create a policy that emits a custom claim "JoinedData" to JWTs issued to linked service principals. This claim contains a value created by joining the data stored in the extensionattribute1 attribute on the user object with ".sandbox". In this example, we exclude the basic claims set in the tokens.

1. Create a claims mapping policy. This policy, linked to specific service principals, adds the EmployeeID and TenantCountry claims to tokens.

a. To create the policy, run the following command:

```
New-AzureADPolicy -Definition @('{"ClaimsMappingPolicy": {"Version":1,"IncludeBasicClaimSet":"true", "ClaimsSchema": [{"Source":"user","ID":"extensionattribute1"}, {"Source":"transformation","ID":"DataJoin","TransformationId":"JoinTheData","JwtClaimType":"JoinedData"}],"ClaimsTransformations": [{"ID":"JoinTheData","TransformationMethod":"Join","InputClaims": [{"ClaimTypeReferenceId":"extensionattribute1","TransformationClaimType":"string1"}], "InputParameters": [{"ID":"string2","Value":"sandbox"}, {"ID":"separator","Value":"."}], "OutputClaims": [{"ClaimTypeReferenceId":"DataJoin","TransformationClaimType":"outputClaim"}]}]}') -DisplayName "TransformClaimsExample" -Type "ClaimsMappingPolicy"
```

b. To see your new policy, and to get the policy ObjectId, run the following command:

```
Get-AzureADPolicy
```

2. Assign the policy to your service principal. You also need to get the ObjectId of your service principal.

a. To see all your organization's service principals, you can [query Microsoft Graph](#). Or, in [Graph Explorer](#), sign in to your Azure AD account.

b. When you have the ObjectId of your service principal, run the following command:

```
Add-AzureADServicePrincipalPolicy -Id <ObjectId of the ServicePrincipal> -RefObjectId <ObjectId of the Policy>
```

See also

To learn how to customize claims issued in the SAML token through the Azure portal, see [How to: Customize claims issued in the SAML token for enterprise applications](#)

How to: Provide optional claims to your Azure AD app

10/23/2019 • 12 minutes to read • [Edit Online](#)

Application developers can use optional claims in their Azure AD apps to specify which claims they want in tokens sent to their application.

You can use optional claims to:

- Select additional claims to include in tokens for your application.
- Change the behavior of certain claims that Azure AD returns in tokens.
- Add and access custom claims for your application.

For the lists of standard claims, see the [access token](#) and [id_token](#) claims documentation.

While optional claims are supported in both v1.0 and v2.0 format tokens, as well as SAML tokens, they provide most of their value when moving from v1.0 to v2.0. One of the goals of the [v2.0 Microsoft identity platform endpoint](#) is smaller token sizes to ensure optimal performance by clients. As a result, several claims formerly included in the access and ID tokens are no longer present in v2.0 tokens and must be asked for specifically on a per-application basis.

Table 1: Applicability

ACCOUNT TYPE	V1.0 TOKENS	V2.0 TOKENS
Personal Microsoft account	N/A	Supported
Azure AD account	Supported	Supported

v1.0 and v2.0 optional claims set

The set of optional claims available by default for applications to use are listed below. To add custom optional claims for your application, see [Directory Extensions](#), below. When adding claims to the **access token**, this will apply to access tokens requested *for* the application (a web API), not those *by* the application. This ensures that no matter the client accessing your API, the right data is present in the access token they use to authenticate against your API.

NOTE

The majority of these claims can be included in JWTs for v1.0 and v2.0 tokens, but not SAML tokens, except where noted in the Token Type column. Consumer accounts support a subset of these claims, marked in the "User Type" column. Many of the claims listed do not apply to consumer users (they have no tenant, so `tenant_ctry` has no value).

Table 2: v1.0 and v2.0 optional claim set

NAME	DESCRIPTION	TOKEN TYPE	USER TYPE	NOTES
<code>auth_time</code>	Time when the user last authenticated. See OpenID Connect spec.	JWT		

NAME	DESCRIPTION	TOKEN TYPE	USER TYPE	NOTES
<code>tenant_region_scope</code>	Region of the resource tenant	JWT		
<code>home_oid</code>	For guest users, the object ID of the user in the user's home tenant.	JWT		
<code>sid</code>	Session ID, used for per-session user sign out.	JWT	Personal and Azure AD accounts.	
<code>platf</code>	Device platform	JWT		Restricted to managed devices that can verify device type.
<code>verified_primary_email</code>	Sourced from the user's PrimaryAuthoritativeEmail	JWT		
<code>verified_secondary_email</code>	Sourced from the user's SecondaryAuthoritativeEmail	JWT		
<code>enfpolids</code>	Enforced policy IDs. A list of the policy IDs that were evaluated for the current user.	JWT		
<code>vnet</code>	VNET specifier information.	JWT		
<code>fwd</code>	IP address.	JWT		Adds the original IPv4 address of the requesting client (when inside a VNET)
<code>ctry</code>	User's country	JWT		Azure AD returns the <code>ctry</code> optional claim if it's present and the value of the claim is a standard two-letter country code, such as FR, JP, SZ, and so on.
<code>tenant_ctry</code>	Resource tenant's country	JWT		

NAME	DESCRIPTION	TOKEN TYPE	USER TYPE	NOTES
xms_pd1	Preferred data location	JWT		<p>For Multi-Geo tenants, this is the 3-letter code showing the geographic region the user is in. For more info, see the Azure AD Connect documentation about preferred data location. For example: APC for Asia Pacific.</p>
xms_pl1	User preferred language	JWT		The user's preferred language, if set. Sourced from their home tenant, in guest access scenarios. Formatted LL-CC ("en-us").
xms_tpl1	Tenant preferred language	JWT		The resource tenant's preferred language, if set. Formatted LL ("en").
ztdid	Zero-touch Deployment ID	JWT		The device identity used for Windows AutoPilot
email	The addressable email for this user, if the user has one.	JWT, SAML	MSA, Azure AD	This value is included by default if the user is a guest in the tenant. For managed users (those inside the tenant), it must be requested through this optional claim or, on v2.0 only, with the OpenID scope. For managed users, the email address must be set in the Office admin portal .

NAME	DESCRIPTION	TOKEN TYPE	USER TYPE	NOTES
<code>groups</code>	Optional formatting for group claims	JWT, SAML		Used in conjunction with the <code>GroupMembershipClaims</code> setting in the application manifest , which must be set as well. For details see [Group claims] (#Configuring-group-optional claims) below. For more information on group claims see How to configure group claims
<code>acct</code>	Users account status in tenant.	JWT, SAML		If the user is a member of the tenant, the value is <code>0</code> . If they are a guest, the value is <code>1</code> .
<code>upn</code>	UserPrincipalName claim.	JWT, SAML		Although this claim is automatically included, you can specify it as an optional claim to attach additional properties to modify its behavior in the guest user case.

v2.0 optional claims

These claims are always included in v1.0 Azure AD tokens, but not included in v2.0 tokens unless requested.

These claims are only applicable for JWTs (ID tokens and Access Tokens).

Table 3: v2.0-only optional claims

JWT CLAIM	NAME	DESCRIPTION	NOTES
<code>ipaddr</code>	IP Address	The IP address the client logged in from.	
<code>onprem_sid</code>	On-Premises Security Identifier		
<code>pwd_exp</code>	Password Expiration Time	The datetime at which the password expires.	
<code>pwd_url</code>	Change Password URL	A URL that the user can visit to change their password.	

JWT CLAIM	NAME	DESCRIPTION	NOTES
<code>in_corp</code>	Inside Corporate Network	Signals if the client is logging in from the corporate network. If they're not, the claim isn't included.	Based off of the trusted IPs settings in MFA.
<code>nickname</code>	Nickname	An additional name for the user, separate from first or last name.	
<code>family_name</code>	Last Name	Provides the last name, surname, or family name of the user as defined in the user object. <code>"family_name": "Miller"</code>	Supported in MSA and Azure AD
<code>given_name</code>	First name	Provides the first or "given" name of the user, as set on the user object. <code>"given_name": "Frank"</code>	Supported in MSA and Azure AD
<code>upn</code>	User Principal Name	An identifier for the user that can be used with the <code>username_hint</code> parameter. Not a durable identifier for the user and should not be used to key data.	See additional properties below for configuration of the claim.

Additional properties of optional claims

Some optional claims can be configured to change the way the claim is returned. These additional properties are mostly used to help migration of on-premises applications with different data expectations (for example, `include_externally_authenticated_upn_without_hash` helps with clients that cannot handle hash marks (#) in the UPN)

Table 4: Values for configuring optional claims

PROPERTY NAME	ADDITIONAL PROPERTY NAME	DESCRIPTION
<code>upn</code>		Can be used for both SAML and JWT responses, and for v1.0 and v2.0 tokens.
	<code>include_externally_authenticated_upn</code>	Includes the guest UPN as stored in the resource tenant. For example, <code>foo_hometenant.com#EXT#@resourcetenant.co</code>
	<code>include_externally_authenticated_upn_with_hash</code>	Same as above, except that the hash marks (#) are replaced with underscores (_), for example <code>foo_hometenant.com_EXT_@resourcetenant.co</code>

Additional properties example

```

"optionalClaims": [
    {
        "idToken": [
            {
                "name": "upn",
                "essential": false,
                "additionalProperties": [ "include_externally_authenticated_upn" ]
            }
        ]
    }
]

```

This OptionalClaims object causes the ID token returned to the client to include another upn with the additional home tenant and resource tenant information. This will only change the `upn` claim in the token if the user is a guest in the tenant (that uses a different IDP for authentication).

Configuring optional claims

You can configure optional claims for your application by modifying the application manifest (See example below). For more info, see the [Understanding the Azure AD application manifest article](#).

IMPORTANT

Access tokens are **always** generated using the manifest of the resource, not the client. So in the request

`...scope=https://graph.microsoft.com/user.read...` the resource is Graph. Thus, the access token is created using the Graph manifest, not the client's manifest. Changing the manifest for your application will never cause tokens for Graph to look different. In order to validate that your `accessToken` changes are in effect, request a token for your application, not another app.

Sample schema:

```

"optionalClaims": [
    {
        "idToken": [
            {
                "name": "auth_time",
                "essential": false
            }
        ],
        "accessToken": [
            {
                "name": "ipaddr",
                "essential": false
            }
        ],
        "saml2Token": [
            {
                "name": "upn",
                "essential": false
            },
            {
                "name": "extension_ab603c56068041afb2f6832e2a17e237_skypeId",
                "source": "user",
                "essential": false
            }
        ]
    }
]

```

OptionalClaims type

Declares the optional claims requested by an application. An application can configure optional claims to be returned in each of three types of tokens (ID token, access token, SAML 2 token) it can receive from the security

token service. The application can configure a different set of optional claims to be returned in each token type. The `OptionalClaims` property of the `Application` entity is an `OptionalClaims` object.

Table 5: OptionalClaims type properties

NAME	TYPE	DESCRIPTION
<code>idToken</code>	Collection (OptionalClaim)	The optional claims returned in the JWT ID token.
<code>accessToken</code>	Collection (OptionalClaim)	The optional claims returned in the JWT access token.
<code>saml2Token</code>	Collection (OptionalClaim)	The optional claims returned in the SAML token.

OptionalClaim type

Contains an optional claim associated with an application or a service principal. The `idToken`, `accessToken`, and `saml2Token` properties of the [OptionalClaims](#) type is a collection of `OptionalClaim`. If supported by a specific claim, you can also modify the behavior of the `OptionalClaim` using the `AdditionalProperties` field.

Table 6: OptionalClaim type properties

NAME	TYPE	DESCRIPTION
<code>name</code>	Edm.String	The name of the optional claim.
<code>source</code>	Edm.String	The source (directory object) of the claim. There are predefined claims and user-defined claims from extension properties. If the source value is null, the claim is a predefined optional claim. If the source value is user, the value in the <code>name</code> property is the extension property from the user object.
<code>essential</code>	Edm.Boolean	If the value is true, the claim specified by the client is necessary to ensure a smooth authorization experience for the specific task requested by the end user. The default value is false.
<code>additionalProperties</code>	Collection (Edm.String)	Additional properties of the claim. If a property exists in this collection, it modifies the behavior of the optional claim specified in the <code>name</code> property.

Configuring directory extension optional claims

In addition to the standard optional claims set, you can also configure tokens to include directory schema extensions. For more info, see [Directory schema extensions](#). This feature is useful for attaching additional user information that your app can use – for example, an additional identifier or important configuration option that the user has set.

NOTE

- Directory schema extensions are an Azure AD-only feature, so if your application manifest requests a custom extension and an MSA user logs into your app, these extensions will not be returned.
- Azure AD optional claims only work with the Azure AD extension and doesn't work with the Microsoft Graph directory extension. Both APIs require the `Directory.ReadWriteAll` permission, which can only be consented by admins.

Directory extension formatting

For extension attributes, use the full name of the extension (in the format: `extension_<appid>_<attributename>`) in the application manifest. The `<appid>` must match the ID of the application requesting the claim.

Within the JWT, these claims will be emitted with the following name format: `extn.<attributename>`.

Within the SAML tokens, these claims will be emitted with the following URI format:

`http://schemas.microsoft.com/identity/claims/extn.<attributename>`

Configuring group optional claims

NOTE

The ability to emit group names for users and groups synced from on-premises is Public Preview.

This section covers the configuration options under optional claims for changing the group attributes used in group claims from the default group objectID to attributes synced from on-premises Windows Active Directory.

IMPORTANT

See [Configure group claims for applications with Azure AD](#) for more details including important caveats for the public preview of group claims from on-premises attributes.

1. In the portal -> Azure Active Directory -> Application Registrations-> Select Application-> Manifest
2. Enable group membership claims by changing the groupMembershipClaim

The valid values are:

- "All"
- "SecurityGroup"
- "DistributionList"
- "DirectoryRole"

For example:

```
"groupMembershipClaims": "SecurityGroup"
```

By default Group ObjectIDs will be emitted in the group claim value. To modify the claim value to contain on premises group attributes, or to change the claim type to role, use OptionalClaims configuration as follows:

3. Set group name configuration optional claims.

If you want to groups in the token to contain the on premises AD group attributes in the optional claims section specify which token type optional claim should be applied to, the name of optional claim requested and any additional properties desired. Multiple token types can be listed:

- idToken for the OIDC ID token

- accessToken for the OAuth/OIDC access token
- Saml2Token for SAML tokens.

NOTE

The Saml2Token type applies to both SAML1.1 and SAML2.0 format tokens

For each relevant token type, modify the groups claim to use the OptionalClaims section in the manifest. The OptionalClaims schema is as follows:

```
{
  "name": "groups",
  "source": null,
  "essential": false,
  "additionalProperties": []
}
```

OPTIONAL CLAIMS SCHEMA	VALUE
name:	Must be "groups"
source:	Not used. Omit or specify null
essential:	Not used. Omit or specify false
additionalProperties:	List of additional properties. Valid options are "sam_account_name", "dns_domain_and_sam_account_name", "netbios_domain_and_sam_account_name", "emit_as_roles"

In additionalProperties only one of "sam_account_name", "dns_domain_and_sam_account_name", "netbios_domain_and_sam_account_name" are required. If more than one is present, the first is used and any others ignored.

Some applications require group information about the user in the role claim. To change the claim type to from a group claim to a role claim, add "emit_as_roles" to additional properties. The group values will be emitted in the role claim.

NOTE

If "emit_as_roles" is used any Application Roles configured that the user is assigned will not appear in the role claim

Examples: Emit groups as group names in OAuth access tokens in dnsDomainName\sAMAccountName format

```
"optionalClaims": {
  "accessToken": [
    {
      "name": "groups",
      "additionalProperties": ["dns_domain_and_sam_account_name"]
    }
  ]
}
```

To emit group names to be returned in netbiosDomain\sAMAccountName format as the roles claim in SAML and OIDC ID Tokens:

```
"optionalClaims": {
    "saml2Token": [
        {
            "name": "groups",
            "additionalProperties": ["netbios_name_and_sam_account_name", "emit_as_roles"]
        }
    ],
    "idToken": [
        {
            "name": "groups",
            "additionalProperties": ["netbios_name_and_sam_account_name", "emit_as_roles"]
        }
    ]
}
```

Optional claims example

In this section, you can walk through a scenario to see how you can use the optional claims feature for your application. There are multiple options available for updating the properties on an application's identity configuration to enable and configure optional claims:

- You can modify the application manifest. The example below will use this method to do the configuration. Read the [Understanding the Azure AD application manifest document](#) first for an introduction to the manifest.
- It's also possible to write an application that uses the [Graph API](#) to update your application. The [Entity and complex type reference](#) in the Graph API reference guide can help you with configuring the optional claims.

Example: In the example below, you will modify an application's manifest to add claims to access, ID, and SAML tokens intended for the application.

1. Sign in to the [Azure portal](#).
2. After you've authenticated, choose your Azure AD tenant by selecting it from the top right corner of the page.
3. Select **App Registrations** from the left hand side.
4. Find the application you want to configure optional claims for in the list and click on it.
5. From the application page, click **Manifest** to open the inline manifest editor.
6. You can directly edit the manifest using this editor. The manifest follows the schema for the [Application entity](#), and auto-formats the manifest once saved. New elements will be added to the `OptionalClaims` property.

```
"optionalClaims":  
{  
    "idToken": [  
        {  
            "name": "upn",  
            "essential": false,  
            "additionalProperties": [ "include_externally_authenticated_upn"]  
        }  
    ],  
    "accessToken": [  
        {  
            "name": "auth_time",  
            "essential": false  
        }  
    ],  
    "saml2Token": [  
        {  
            "name": "extension_ab603c56068041afb2f6832e2a17e237_skypeId",  
            "source": "user",  
            "essential": true  
        }  
    ]  
}
```

In this case, different optional claims were added to each type of token that the application can receive.

The ID tokens will now contain the UPN for federated users in the full form (

<upn>_<homedomain>#EXT#@<resourcedomain>). The access tokens that other clients request for this application will now include the auth_time claim. The SAML tokens will now contain the skypeId directory schema extension (in this example, the app ID for this app is ab603c56068041afb2f6832e2a17e237). The SAML tokens will expose the Skype ID as extension_skypeId .

- When you're finished updating the manifest, click **Save** to save the manifest

Next steps

Learn more about the standard claims provided by Azure AD.

- [ID tokens](#)
- [Access tokens](#)

Configurable token lifetimes in Azure Active Directory (Preview)

11/4/2019 • 22 minutes to read • [Edit Online](#)

You can specify the lifetime of a token issued by Azure Active Directory (Azure AD). You can set token lifetimes for all apps in your organization, for a multi-tenant (multi-organization) application, or for a specific service principal in your organization.

IMPORTANT

After hearing from customers during the preview, we've implemented [authentication session management capabilities](#) in Azure AD Conditional Access. You can use this new feature to configure refresh token lifetimes by setting sign in frequency. After May 1, 2020 you will not be able to use Configurable Token Lifetime policy to configure session and refresh tokens. You can still configure access token lifetimes after the deprecation.

In Azure AD, a policy object represents a set of rules that are enforced on individual applications or on all applications in an organization. Each policy type has a unique structure, with a set of properties that are applied to objects to which they are assigned.

You can designate a policy as the default policy for your organization. The policy is applied to any application in the organization, as long as it is not overridden by a policy with a higher priority. You also can assign a policy to specific applications. The order of priority varies by policy type.

NOTE

Configurable token lifetime policy is not supported for SharePoint Online. Even though you have the ability to create this policy via PowerShell, SharePoint Online will not acknowledge this policy. Refer to the [SharePoint Online blog](#) to learn more about configuring idle session timeouts.

- The default lifetime for the SharePoint Online access token is 1 hour.
- The default max inactive time of the SharePoint Online refresh token is 90 days.

Token types

You can set token lifetime policies for refresh tokens, access tokens, SAML tokens, session tokens, and ID tokens.

Access tokens

Clients use access tokens to access a protected resource. An access token can be used only for a specific combination of user, client, and resource. Access tokens cannot be revoked and are valid until their expiry. A malicious actor that has obtained an access token can use it for extent of its lifetime. Adjusting the lifetime of an access token is a trade-off between improving system performance and increasing the amount of time that the client retains access after the user's account is disabled. Improved system performance is achieved by reducing the number of times a client needs to acquire a fresh access token. The default is 1 hour - after 1 hour, the client must use the refresh token to (usually silently) acquire a new refresh token and access token.

SAML tokens

SAML tokens are used by many web based SAAS applications, and are obtained using Azure Active Directory's SAML2 protocol endpoint. They are also consumed by applications using WS-Federation. The default lifetime of the token is 1 hour. After From and applications perspective the validity period of the token is specified by the NotOnOrAfter value of the <conditions ...> element in the token. After the token validity period the client must initiate a new authentication request, which will often be satisfied without interactive sign in as a result of the Single Sign On

(SSO) Session token.

The value of `NotOnOrAfter` can be changed using the `AccessTokenLifetime` parameter in a `TokenLifetimePolicy`. It will be set to the lifetime configured in the policy if any, plus a clock skew factor of five minutes.

Note that the subject confirmation `NotOnOrAfter` specified in the element is not affected by the Token Lifetime configuration.

Refresh tokens

When a client acquires an access token to access a protected resource, the client also receives a refresh token. The refresh token is used to obtain new access/refresh token pairs when the current access token expires. A refresh token is bound to a combination of user and client. A refresh token can be [revoked at any time](#), and the token's validity is checked every time the token is used. Refresh tokens are not revoked when used to fetch new access tokens - it's best practice, however, to securely delete the old token when getting a new one.

It's important to make a distinction between confidential clients and public clients, as this impacts how long refresh tokens can be used. For more information about different types of clients, see [RFC 6749](#).

Token lifetimes with confidential client refresh tokens

Confidential clients are applications that can securely store a client password (secret). They can prove that requests are coming from the secured client application and not from a malicious actor. For example, a web app is a confidential client because it can store a client secret on the web server. It is not exposed. Because these flows are more secure, the default lifetimes of refresh tokens issued to these flows is `until-revoked`, cannot be changed by using policy, and will not be revoked on voluntary password resets.

Token lifetimes with public client refresh tokens

Public clients cannot securely store a client password (secret). For example, an iOS/Android app cannot obfuscate a secret from the resource owner, so it is considered a public client. You can set policies on resources to prevent refresh tokens from public clients older than a specified period from obtaining a new access/refresh token pair. (To do this, use the Refresh Token Max Inactive Time property (`MaxInactiveTime`)). You also can use policies to set a period beyond which the refresh tokens are no longer accepted. (To do this, use the Refresh Token Max Age property.) You can adjust the lifetime of a refresh token to control when and how often the user is required to reenter credentials, instead of being silently reauthenticated, when using a public client application.

NOTE

The Max Age property is the length of time a single token can be used.

ID tokens

ID tokens are passed to websites and native clients. ID tokens contain profile information about a user. An ID token is bound to a specific combination of user and client. ID tokens are considered valid until their expiry. Usually, a web application matches a user's session lifetime in the application to the lifetime of the ID token issued for the user. You can adjust the lifetime of an ID token to control how often the web application expires the application session, and how often it requires the user to be reauthenticated with Azure AD (either silently or interactively).

Single sign-on session tokens

When a user authenticates with Azure AD, a single sign-on session (SSO) is established with the user's browser and Azure AD. The SSO token, in the form of a cookie, represents this session. The SSO session token is not bound to a specific resource/client application. SSO session tokens can be revoked, and their validity is checked every time they are used.

Azure AD uses two kinds of SSO session tokens: persistent and nonpersistent. Persistent session tokens are stored as persistent cookies by the browser. Nonpersistent session tokens are stored as session cookies. (Session cookies are destroyed when the browser is closed.) Usually, a nonpersistent session token is stored. But, when the user selects the **Keep me signed in** check box during authentication, a persistent session token is stored.

Nonpersistent session tokens have a lifetime of 24 hours. Persistent tokens have a lifetime of 180 days. Anytime an

SSO session token is used within its validity period, the validity period is extended another 24 hours or 180 days, depending on the token type. If an SSO session token is not used within its validity period, it is considered expired and is no longer accepted.

You can use a policy to set the time after the first session token was issued beyond which the session token is no longer accepted. (To do this, use the Session Token Max Age property.) You can adjust the lifetime of a session token to control when and how often a user is required to reenter credentials, instead of being silently authenticated, when using a web application.

Token lifetime policy properties

A token lifetime policy is a type of policy object that contains token lifetime rules. Use the properties of the policy to control specified token lifetimes. If no policy is set, the system enforces the default lifetime value.

Configurable token lifetime properties

PROPERTY	POLICY PROPERTY STRING	AFFECTS	DEFAULT	MINIMUM	MAXIMUM
Access Token Lifetime	AccessTokenLifetime ²	Access tokens, ID tokens, SAML2 tokens	1 hour	10 minutes	1 day
Refresh Token Max Inactive Time	MaxInactiveTime	Refresh tokens	90 days	10 minutes	90 days
Single-Factor Refresh Token Max Age	MaxAgeSingleFactor	Refresh tokens (for any users)	Until-revoked	10 minutes	Until-revoked ¹
Multi-Factor Refresh Token Max Age	MaxAgeMultiFactor	Refresh tokens (for any users)	Until-revoked	10 minutes	Until-revoked ¹
Single-Factor Session Token Max Age	MaxAgeSessionSingleFactor	Session tokens (persistent and nonpersistent)	Until-revoked	10 minutes	Until-revoked ¹
Multi-Factor Session Token Max Age	MaxAgeSessionMultiFactor	Session tokens (persistent and nonpersistent)	Until-revoked	10 minutes	Until-revoked ¹

- ¹365 days is the maximum explicit length that can be set for these attributes.

- ²To ensure the Microsoft Teams Web client works, it is recommended to keep AccessTokenLifetime to greater than 15 minutes for Microsoft Teams.

Exceptions

PROPERTY	AFFECTS	DEFAULT
Refresh Token Max Age (issued for federated users who have insufficient revocation information ¹)	Refresh tokens (issued for federated users who have insufficient revocation information ¹)	12 hours
Refresh Token Max Inactive Time (issued for confidential clients)	Refresh tokens (issued for confidential clients)	90 days
Refresh Token Max Age (issued for confidential clients)	Refresh tokens (issued for confidential clients)	Until-revoked

- ¹Federated users who have insufficient revocation information include any users who do not have the

"LastPasswordChangeTimestamp" attribute synced. These users are given this short Max Age because AAD is unable to verify when to revoke tokens that are tied to an old credential (such as a password that has been changed) and must check back in more frequently to ensure that the user and associated tokens are still in good standing. To improve this experience, tenant admins must ensure that they are syncing the "LastPasswordChangeTimestamp" attribute (this can be set on the user object using Powershell or through AADSync).

Policy evaluation and prioritization

You can create and then assign a token lifetime policy to a specific application, to your organization, and to service principals. Multiple policies might apply to a specific application. The token lifetime policy that takes effect follows these rules:

- If a policy is explicitly assigned to the service principal, it is enforced.
- If no policy is explicitly assigned to the service principal, a policy explicitly assigned to the parent organization of the service principal is enforced.
- If no policy is explicitly assigned to the service principal or to the organization, the policy assigned to the application is enforced.
- If no policy has been assigned to the service principal, the organization, or the application object, the default values are enforced. (See the table in [Configurable token lifetime properties](#).)

For more information about the relationship between application objects and service principal objects, see [Application and service principal objects in Azure Active Directory](#).

A token's validity is evaluated at the time the token is used. The policy with the highest priority on the application that is being accessed takes effect.

All timespans used here are formatted according to the C# `TimeSpan` object - D.HH:MM:SS. So 80 days and 30 minutes would be `80.00:30:00`. The leading D can be dropped if zero, so 90 minutes would be `00:90:00`.

NOTE

Here's an example scenario.

A user wants to access two web applications: Web Application A and Web Application B.

Factors:

- Both web applications are in the same parent organization.
- Token Lifetime Policy 1 with a Session Token Max Age of eight hours is set as the parent organization's default.
- Web Application A is a regular-use web application and isn't linked to any policies.
- Web Application B is used for highly sensitive processes. Its service principal is linked to Token Lifetime Policy 2, which has a Session Token Max Age of 30 minutes.

At 12:00 PM, the user starts a new browser session and tries to access Web Application A. The user is redirected to Azure AD and is asked to sign in. This creates a cookie that has a session token in the browser. The user is redirected back to Web Application A with an ID token that allows the user to access the application.

At 12:15 PM, the user tries to access Web Application B. The browser redirects to Azure AD, which detects the session cookie. Web Application B's service principal is linked to Token Lifetime Policy 2, but it's also part of the parent organization, with default Token Lifetime Policy 1. Token Lifetime Policy 2 takes effect because policies linked to service principals have a higher priority than organization default policies. The session token was originally issued within the last 30 minutes, so it is considered valid. The user is redirected back to Web Application B with an ID token that grants them access.

At 1:00 PM, the user tries to access Web Application A. The user is redirected to Azure AD. Web Application A is not linked to any policies, but because it is in an organization with default Token Lifetime Policy 1, that policy takes effect. The session cookie that was originally issued within the last eight hours is detected. The user is silently redirected back to Web Application A with a new ID token. The user is not required to authenticate.

Immediately afterward, the user tries to access Web Application B. The user is redirected to Azure AD. As before, Token Lifetime Policy 2 takes effect. Because the token was issued more than 30 minutes ago, the user is prompted to reenter their sign-in credentials. A brand-new session token and ID token are issued. The user can then access Web Application B.

Configurable policy property details

Access Token Lifetime

String: AccessTokenLifetime

Affects: Access tokens, ID tokens, SAML tokens

Summary: This policy controls how long access and ID tokens for this resource are considered valid. Reducing the Access Token Lifetime property mitigates the risk of an access token or ID token being used by a malicious actor for an extended period of time. (These tokens cannot be revoked.) The trade-off is that performance is adversely affected, because the tokens have to be replaced more often.

Refresh Token Max Inactive Time

String: MaxInactiveTime

Affects: Refresh tokens

Summary: This policy controls how old a refresh token can be before a client can no longer use it to retrieve a new access/refresh token pair when attempting to access this resource. Because a new refresh token usually is returned when a refresh token is used, this policy prevents access if the client tries to access any resource by using the current refresh token during the specified period of time.

This policy forces users who have not been active on their client to reauthenticate to retrieve a new refresh token.

The Refresh Token Max Inactive Time property must be set to a lower value than the Single-Factor Token Max Age and the Multi-Factor Refresh Token Max Age properties.

Single-Factor Refresh Token Max Age

String: MaxAgeSingleFactor

Affects: Refresh tokens

Summary: This policy controls how long a user can use a refresh token to get a new access/refresh token pair after they last authenticated successfully by using only a single factor. After a user authenticates and receives a new refresh token, the user can use the refresh token flow for the specified period of time. (This is true as long as the current refresh token is not revoked, and it is not left unused for longer than the inactive time.) At that point, the user is forced to reauthenticate to receive a new refresh token.

Reducing the max age forces users to authenticate more often. Because single-factor authentication is considered less secure than multi-factor authentication, we recommend that you set this property to a value that is equal to or lesser than the Multi-Factor Refresh Token Max Age property.

Multi-Factor Refresh Token Max Age

String: MaxAgeMultiFactor

Affects: Refresh tokens

Summary: This policy controls how long a user can use a refresh token to get a new access/refresh token pair after they last authenticated successfully by using multiple factors. After a user authenticates and receives a new refresh token, the user can use the refresh token flow for the specified period of time. (This is true as long as the current refresh token is not revoked, and it is not unused for longer than the inactive time.) At that point, users are forced to reauthenticate to receive a new refresh token.

Reducing the max age forces users to authenticate more often. Because single-factor authentication is considered less secure than multi-factor authentication, we recommend that you set this property to a value that is equal to or greater than the Single-Factor Refresh Token Max Age property.

Single-Factor Session Token Max Age

String: MaxAgeSessionSingleFactor

Affects: Session tokens (persistent and nonpersistent)

Summary: This policy controls how long a user can use a session token to get a new ID and session token after they last authenticated successfully by using only a single factor. After a user authenticates and receives a new session token, the user can use the session token flow for the specified period of time. (This is true as long as the current session token is not revoked and has not expired.) After the specified period of time, the user is forced to reauthenticate to receive a new session token.

Reducing the max age forces users to authenticate more often. Because single-factor authentication is considered less secure than multi-factor authentication, we recommend that you set this property to a value that is equal to or less than the Multi-Factor Session Token Max Age property.

Multi-Factor Session Token Max Age

String: MaxAgeSessionMultiFactor

Affects: Session tokens (persistent and nonpersistent)

Summary: This policy controls how long a user can use a session token to get a new ID and session token after the last time they authenticated successfully by using multiple factors. After a user authenticates and receives a new session token, the user can use the session token flow for the specified period of time. (This is true as long as the current session token is not revoked and has not expired.) After the specified period of time, the user is forced to reauthenticate to receive a new session token.

Reducing the max age forces users to authenticate more often. Because single-factor authentication is considered less secure than multi-factor authentication, we recommend that you set this property to a value that is equal to or greater than the Single-Factor Session Token Max Age property.

Example token lifetime policies

Many scenarios are possible in Azure AD when you can create and manage token lifetimes for apps, service principals, and your overall organization. In this section, we walk through a few common policy scenarios that can help you impose new rules for:

- Token Lifetime
- Token Max Inactive Time
- Token Max Age

In the examples, you can learn how to:

- Manage an organization's default policy
- Create a policy for web sign-in
- Create a policy for a native app that calls a web API
- Manage an advanced policy

Prerequisites

In the following examples, you create, update, link, and delete policies for apps, service principals, and your overall organization. If you are new to Azure AD, we recommend that you learn about [how to get an Azure AD tenant](#) before you proceed with these examples.

To get started, do the following steps:

1. Download the latest [Azure AD PowerShell Module Public Preview release](#).
2. Run the `Connect` command to sign in to your Azure AD admin account. Run this command each time you start a new session.

```
Connect-AzureAD -Confirm
```

3. To see all policies that have been created in your organization, run the following command. Run this command after most operations in the following scenarios. Running the command also helps you get the *** of your policies.

```
Get-AzureADPolicy
```

Example: Manage an organization's default policy

In this example, you create a policy that lets your users' sign in less frequently across your entire organization. To do this, create a token lifetime policy for Single-Factor Refresh Tokens, which is applied across your organization. The policy is applied to every application in your organization, and to each service principal that doesn't already have a policy set.

1. Create a token lifetime policy.
 - a. Set the Single-Factor Refresh Token to "until-revoked." The token doesn't expire until access is revoked.
Create the following policy definition:

```
@('{
    "TokenLifetimePolicy": {
        "Version":1,
        "MaxAgeSingleFactor":"until-revoked"
    }
}')
```

- b. To create the policy, run the following command:

```
$policy = New-AzureADPolicy -Definition @('{"TokenLifetimePolicy":{"Version":1,"MaxAgeSingleFactor":"until-revoked"}}') -DisplayName "OrganizationDefaultPolicyScenario" -IsOrganizationDefault $true -Type "TokenLifetimePolicy"
```

- c. To see your new policy, and to get the policy's **ObjectId**, run the following command:

```
Get-AzureADPolicy -Id $policy.Id
```

2. Update the policy.

You might decide that the first policy you set in this example is not as strict as your service requires. To set your Single-Factor Refresh Token to expire in two days, run the following command:

```
Set-AzureADPolicy -Id $policy.Id -DisplayName $policy.DisplayName -Definition @('{"TokenLifetimePolicy":{"Version":1,"MaxAgeSingleFactor":"2.00:00:00"}}')
```

Example: Create a policy for web sign-in

In this example, you create a policy that requires users to authenticate more frequently in your web app. This policy sets the lifetime of the access/ID tokens and the max age of a multi-factor session token to the service principal of your web app.

1. Create a token lifetime policy.

This policy, for web sign-in, sets the access/ID token lifetime and the max single-factor session token age to two hours.

- a. To create the policy, run this command:

```
$policy = New-AzureADPolicy -Definition @('{"TokenLifetimePolicy":{"Version":1,"AccessTokenLifetime":"02:00:00","MaxAgeSessionSingleFactor":"02:00:00"}}') -DisplayName "WebPolicyScenario" -IsOrganizationDefault $false -Type "TokenLifetimePolicy"
```

- b. To see your new policy, and to get the policy **ObjectId**, run the following command:

```
Get-AzureADPolicy -Id $policy.Id
```

2. Assign the policy to your service principal. You also need to get the **ObjectId** of your service principal.

- a. Use the [Get-AzureADServicePrincipal](#) cmdlet to see all your organization's service principals or a single service principal.

```
# Get ID of the service principal  
$sp = Get-AzureADServicePrincipal -Filter "DisplayName eq '<service principal display name>'"
```

- b. When you have the service principal, run the following command:

```
# Assign policy to a service principal  
Add-AzureADServicePrincipalPolicy -Id $sp.ObjectId -RefObjectId $policy.Id
```

Example: Create a policy for a native app that calls a web API

In this example, you create a policy that requires users to authenticate less frequently. The policy also lengthens the amount of time a user can be inactive before the user must reauthenticate. The policy is applied to the web API. When the native app requests the web API as a resource, this policy is applied.

1. Create a token lifetime policy.

- To create a strict policy for a web API, run the following command:

```
$policy = New-AzureADPolicy -Definition @('{"TokenLifetimePolicy": {"Version":1,"MaxInactiveTime":"30.00:00:00","MaxAgeMultiFactor":"until-revoked","MaxAgeSingleFactor":"180.00:00:00"}}') -DisplayName "WebApiDefaultPolicyScenario" -IsOrganizationDefault $false -Type "TokenLifetimePolicy"
```

- To see your new policy, run the following command:

```
Get-AzureADPolicy -Id $policy.Id
```

2. Assign the policy to your web API. You also need to get the **ObjectId** of your application. Use the [Get-AzureADApplication](#) cmdlet to find your app's **ObjectId**, or use the [Azure portal](#).

Get the **ObjectId** of your app and assign the policy:

```
# Get the application  
$app = Get-AzureADApplication -Filter "DisplayName eq 'Fourth Coffee Web API'"  
  
# Assign the policy to your web API.  
Add-AzureADApplicationPolicy -Id $app.ObjectId -RefObjectId $policy.Id
```

Example: Manage an advanced policy

In this example, you create a few policies to learn how the priority system works. You also learn how to manage multiple policies that are applied to several objects.

1. Create a token lifetime policy.

- To create an organization default policy that sets the Single-Factor Refresh Token lifetime to 30 days, run the following command:

```
$policy = New-AzureADPolicy -Definition @('{"TokenLifetimePolicy": {"Version":1,"MaxAgeSingleFactor":"30.00:00:00"}}') -DisplayName "ComplexPolicyScenario" -IsOrganizationDefault $true -Type "TokenLifetimePolicy"
```

- To see your new policy, run the following command:

```
Get-AzureADPolicy -Id $policy.Id
```

2. Assign the policy to a service principal.

Now, you have a policy that applies to the entire organization. You might want to preserve this 30-day policy for a specific service principal, but change the organization default policy to the upper limit of "until-revoked."

- To see all your organization's service principals, you use the [Get-AzureADServicePrincipal](#) cmdlet.
- When you have the service principal, run the following command:

```
# Get ID of the service principal  
$sp = Get-AzureADServicePrincipal -Filter "DisplayName eq '<service principal display name>'"  
  
# Assign policy to a service principal  
Add-AzureADServicePrincipalPolicy -Id $sp.ObjectId -RefObjectId $policy.Id
```

3. Set the `IsOrganizationDefault` flag to false:

```
Set-AzureADPolicy -Id $policy.Id -DisplayName "ComplexPolicyScenario" -IsOrganizationDefault $false
```

4. Create a new organization default policy:

```
New-AzureADPolicy -Definition @('{"TokenLifetimePolicy":{"Version":1,"MaxAgeSingleFactor":"until-revoked"}}') -DisplayName "ComplexPolicyScenarioTwo" -IsOrganizationDefault $true -Type "TokenLifetimePolicy"
```

You now have the original policy linked to your service principal, and the new policy is set as your organization default policy. It's important to remember that policies applied to service principals have priority over organization default policies.

Cmdlet reference

Manage policies

You can use the following cmdlets to manage policies.

New-AzureADPolicy

Creates a new policy.

```
New-AzureADPolicy -Definition <Array of Rules> -DisplayName <Name of Policy> -IsOrganizationDefault <boolean> -Type <Policy Type>
```

PARAMETERS	DESCRIPTION	EXAMPLE
<code>-Definition</code>	Array of stringified JSON that contains all the policy's rules.	<code>-Definition @('{"TokenLifetimePolicy":{"Version":1,"MaxInactiveTime":"20:00:00"}}')</code>
<code>-DisplayName</code>	String of the policy name.	<code>-DisplayName "MyTokenPolicy"</code>
<code>-IsOrganizationDefault</code>	If true, sets the policy as the organization's default policy. If false, does nothing.	<code>-IsOrganizationDefault \$true</code>
<code>-Type</code>	Type of policy. For token lifetimes, always use "TokenLifetimePolicy."	<code>-Type "TokenLifetimePolicy"</code>
<code>-AlternativeIdentifier</code> [Optional]	Sets an alternative ID for the policy.	<code>-AlternativeIdentifier "myAltId"</code>

Get-AzureADPolicy

Gets all Azure AD policies or a specified policy.

```
Get-AzureADPolicy
```

PARAMETERS	DESCRIPTION	EXAMPLE
<code>-Id</code> [Optional]	ObjectId (ID) of the policy you want.	<code>-Id <ObjectId of Policy></code>

Get-AzureADPolicyAppliedObject

Gets all apps and service principals that are linked to a policy.

```
Get-AzureADPolicyAppliedObject -Id <ObjectId of Policy>
```

PARAMETERS	DESCRIPTION	EXAMPLE
-Id	ObjectId (ID) of the policy you want.	<code>-Id <ObjectId of Policy></code>

Set-AzureADPolicy

Updates an existing policy.

```
Set-AzureADPolicy -Id <ObjectId of Policy> -DisplayName <string>
```

PARAMETERS	DESCRIPTION	EXAMPLE
-Id	ObjectId (ID) of the policy you want.	<code>-Id <ObjectId of Policy></code>
-DisplayName	String of the policy name.	<code>-DisplayName "MyTokenPolicy"</code>
-Definition [Optional]	Array of stringified JSON that contains all the policy's rules.	<code>-Definition @('{"TokenLifetimePolicy": {"Version":1,"MaxInactiveTime":"20:00:00"}}')</code>
-IsOrganizationDefault [Optional]	If true, sets the policy as the organization's default policy. If false, does nothing.	<code>-IsOrganizationDefault \$true</code>
-Type [Optional]	Type of policy. For token lifetimes, always use "TokenLifetimePolicy."	<code>-Type "TokenLifetimePolicy"</code>
-AlternativeIdentifier [Optional]	Sets an alternative ID for the policy.	<code>-AlternativeIdentifier "myAltId"</code>

Remove-AzureADPolicy

Deletes the specified policy.

```
Remove-AzureADPolicy -Id <ObjectId of Policy>
```

PARAMETERS	DESCRIPTION	EXAMPLE
-Id	ObjectId (ID) of the policy you want.	<code>-Id <ObjectId of Policy></code>

Application policies

You can use the following cmdlets for application policies.

Add-AzureADApplicationPolicy

Links the specified policy to an application.

```
Add-AzureADApplicationPolicy -Id <ObjectId of Application> -RefObjectId <ObjectId of Policy>
```

PARAMETERS	DESCRIPTION	EXAMPLE
-Id	ObjectId (ID) of the application.	<code>-Id <ObjectId of Application></code>
-RefObjectId	ObjectId of the policy.	<code>-RefObjectId <ObjectId of Policy></code>

Get-AzureADApplicationPolicy

Gets the policy that is assigned to an application.

```
Get-AzureADApplicationPolicy -Id <ObjectId of Application>
```

PARAMETERS	DESCRIPTION	EXAMPLE
-Id	ObjectId (ID) of the application.	<code>-Id <ObjectId of Application></code>

Remove-AzureADApplicationPolicy

Removes a policy from an application.

```
Remove-AzureADApplicationPolicy -Id <ObjectId of Application> -PolicyId <ObjectId of Policy>
```

PARAMETERS	DESCRIPTION	EXAMPLE
-Id	ObjectId (ID) of the application.	<code>-Id <ObjectId of Application></code>
-PolicyId	ObjectId of the policy.	<code>-PolicyId <ObjectId of Policy></code>

Service principal policies

You can use the following cmdlets for service principal policies.

Add-AzureADServicePrincipalPolicy

Links the specified policy to a service principal.

```
Add-AzureADServicePrincipalPolicy -Id <ObjectId of ServicePrincipal> -RefObjectId <ObjectId of Policy>
```

PARAMETERS	DESCRIPTION	EXAMPLE
-Id	ObjectId (ID) of the application.	<code>-Id <ObjectId of Application></code>
-RefObjectId	ObjectId of the policy.	<code>-RefObjectId <ObjectId of Policy></code>

Get-AzureADServicePrincipalPolicy

Gets any policy linked to the specified service principal.

```
Get-AzureADServicePrincipalPolicy -Id <ObjectId of ServicePrincipal>
```

PARAMETERS	DESCRIPTION	EXAMPLE
<code>-Id</code>	ObjectId (ID) of the application.	<code>-Id <ObjectId of Application></code>

Remove-AzureADServicePrincipalPolicy

Removes the policy from the specified service principal.

```
Remove-AzureADServicePrincipalPolicy -Id <ObjectId of ServicePrincipal> -PolicyId <ObjectId of Policy>
```

PARAMETERS	DESCRIPTION	EXAMPLE
<code>-Id</code>	ObjectId (ID) of the application.	<code>-Id <ObjectId of Application></code>
<code>-PolicyId</code>	ObjectId of the policy.	<code>-PolicyId <ObjectId of Policy></code>

Error handling best practices for Azure Active Directory Authentication Library (ADAL) clients

10/15/2019 • 17 minutes to read • [Edit Online](#)

This article provides guidance on the type of errors that developers may encounter, when using ADAL to authenticate users. When using ADAL, there are several cases where a developer may need to step in and handle errors. Proper error handling ensures a great end-user experience, and limits the number of times the end user needs to sign in.

In this article, we explore the specific cases for each platform supported by ADAL, and how your application can handle each case properly. The error guidance is split into two broader categories, based on the token acquisition patterns provided by ADAL APIs:

- **AcquireTokenSilent:** Client attempts to get a token silently (no UI), and may fail if ADAL is unsuccessful.
- **AcquireToken:** Client can attempt silent acquisition, but can also perform interactive requests that require sign-in.

TIP

It's a good idea to log all errors and exceptions when using ADAL and Azure AD. Logs are not only helpful for understanding the overall health of your application, but are also important when debugging broader problems. While your application may recover from certain errors, they may hint at broader design problems that require code changes in order to resolve.

When implementing the error conditions covered in this document, you should log the error code and description for the reasons discussed earlier. See the [Error and logging reference](#) for examples of logging code.

AcquireTokenSilent

AcquireTokenSilent attempts to get a token with the guarantee that the end user does not see a User Interface (UI). There are several cases where silent acquisition may fail, and needs to be handled through interactive requests or by a default handler. We dive into the specifics of when and how to employ each case in the sections that follow.

There is a set of errors generated by the operating system, which may require error handling specific to the application. For more information, see "Operating System" errors section in [Error and logging reference](#).

Application scenarios

- [Native client](#) applications (iOS, Android, .NET Desktop, or Xamarin)
- [Web client](#) applications calling a [resource](#) (.NET)

Error cases and actionable steps

Fundamentally, there are two cases of AcquireTokenSilent errors:

CASE	DESCRIPTION
------	-------------

CASE	DESCRIPTION
Case 1: Error is resolvable with an interactive sign-in	<p>For errors caused by a lack of valid tokens, an interactive request is necessary. Specifically, cache lookup and an invalid/expired refresh token require an AcquireToken call to resolve.</p> <p>In these cases, the end user needs to be prompted to sign in. The application can choose to do an interactive request immediately, after end-user interaction (such as hitting a sign-in button), or later. The choice depends on the desired behavior of the application.</p> <p>See the code in the following section for this specific case and the errors that diagnose it.</p>
Case 2: Error is not resolvable with an interactive sign-in	<p>For network and transient/temporary errors, or other failures, performing an interactive AcquireToken request does not resolve the issue. Unnecessary interactive sign-in prompts can also frustrate end users. ADAL automatically attempts a single retry for most errors on AcquireTokenSilent failures.</p> <p>The client application can also attempt a retry at some later point, but when and how is dependent on the application behavior and desired end-user experience. For example, the application can do an AcquireTokenSilent retry after a few minutes, or in response to some end-user action. An immediate retry will result in the application being throttled, and should not be attempted.</p> <p>A subsequent retry failing with the same error does not mean the client should do an interactive request using AcquireToken, as it does not resolve the error.</p> <p>See the code in the following section for this specific case and the errors that diagnose it.</p>

.NET

The following guidance provides examples for error handling in conjunction with ADAL methods:

- acquireTokenSilentAsync(...)
- acquireTokenSilentSync(...)
- [deprecated] acquireTokenSilent(...)
- [deprecated] acquireTokenByRefreshToken(...)

Your code would be implemented as follows:

```

try{
    AcquireTokenSilentAsync(...);
}

catch (AdalSilentTokenAcquisitionException e) {
    // Exception: AdalSilentTokenAcquisitionException
    // Caused when there are no tokens in the cache or a required refresh failed.

    // Action: Case 1, resolvable with an interactive request.
}

catch(AdalServiceException e) {
    // Exception: AdalServiceException
    // Represents an error produced by the STS.
    // e.ErrorCode contains the error code and description, which can be used for debugging.
    // NOTE: Do not code a dependency on the contents of the error description, as it can change over time.

    // Action: Case 2, not resolvable with an interactive request.
    // Attempt retry after a timed interval or user action.
}

catch (AdalException e) {
    // Exception: AdalException
    // Represents a library exception generated by ADAL .NET.
    // e.ErrorCode contains the error code.

    // Action: Case 2, not resolvable with an interactive request.
    // Attempt retry after a timed interval or user action.
    // Example Error: network_not_available, default case.
}

```

Android

The following guidance provides examples for error handling in conjunction with ADAL methods:

- acquireTokenSilentSync(...)
- acquireTokenSilentAsync(...)
- [deprecated] acquireTokenSilent(...)

Your code would be implemented as follows:

```

// *Inside callback*
public void onError(Exception e) {

    if (e instanceof AuthenticationException) {
        // Exception: AdalException
        // Represents a library exception generated by ADAL Android.
        // Error Code: e.getCode().

        // Errors: ADALError.ERROR_SILENT_REQUEST,
        // ADALError.AUTH_REFRESH_FAILED_PROMPT_NOT_ALLOWED,
        // ADALError.INVALID_TOKEN_CACHE_ITEM
        // Description: Request failed due to no tokens in
        // cache or failed a required refresh.

        // Action: Case 1, resolvable with an interactive request.

        // Action: Case 2, not resolvable with an interactive request.
        // Attempt retry after a timed interval or user action.
        // Example Errors: default case,
        // DEVICE_CONNECTION_IS_NOT_AVAILABLE,
        // BROKER_AUTHENTICATOR_ERROR_GETAUTHTOKEN,
    }
}

```

iOS

The following guidance provides examples for error handling in conjunction with ADAL methods:

- `acquireTokenSilentWithResource(...)`

Your code would be implemented as follows:

```
[context acquireTokenSilentWithResource:[ARGS], completionBlock:^(ADAuthenticationResult *result) {
    if (result.status == AD_FAILED) {
        if ([error.domain isEqualToString:ADAuthenticationErrorDomain]){
            // Exception: AD_FAILED
            // Represents a library error generated by ADAL Objective-C.
            // Error Code: result.error.code

            // Errors: AD_ERROR_SERVER_REFRESH_TOKEN_REJECTED, AD_ERROR_CACHE_NO_REFRESH_TOKEN
            // Description: No tokens in cache or failed a required token refresh failed.
            // Action: Case 1, resolvable with an interactive request.

            // Error: AD_ERROR_CACHE_MULTIPLE_USERS
            // Description: There was ambiguity in the silent request resulting in multiple cache items.
            // Action: Special Case, application should perform another silent request and specify the user
            using ADUserIdentity.
            // Can be caused in cases of a multi-user application.

            // Action: Case 2, not resolvable with an interactive request.
            // Attempt retry after some time or user action.
            // Example Errors: default case,
            // AD_ERROR_CACHE_BAD_FORMAT
        }
    }
}]
```

AcquireToken

`AcquireToken` is the default ADAL method used to get tokens. In cases where user identity is required, `AcquireToken` attempts to get a token silently first, then displays UI if necessary (unless `PromptBehavior.Never` is passed). In cases where application identity is required, `AcquireToken` attempts to get a token, but doesn't show UI as there is no end user.

When handling `AcquireToken` errors, error handling is dependent on the platform and scenario the application is trying to achieve.

The operating system can also generate a set of errors, which require error handling dependent on the specific application. For more information, see "Operating System errors" in [Error and logging reference](#).

Application scenarios

- Native client applications (iOS, Android, .NET Desktop, or Xamarin)
- Web applications that call a resource API (.NET)
- Single-page applications (JavaScript)
- Service-to-Service applications (.NET, Java)
 - All scenarios, including on-behalf-of
 - On-Behalf-of specific scenarios

Error cases and actionable steps: Native client applications

If you're building a native client application, there are a few error handling cases to consider which relate to network issues, transient failures, and other platform-specific errors. In most cases, an application shouldn't perform immediate retries, but rather wait for end-user interaction that prompts a sign-in.

There are a few special cases in which a single retry may resolve the issue. For example, when a user needs to

enable data on a device, or completed the Azure AD broker download after the initial failure.

In cases of failure, an application can present UI to allow the end user to perform some interaction that prompts a retry. For instance, if the device failed for an offline error, a "Try to Sign in again" button prompting an AcquireToken retry rather than immediately retrying the failure.

Error handling in native applications can be defined by two cases:

Case 1: Non-Retryable Error (most cases)	1. Do not attempt immediate retry. Present the end-user UI based on the specific error that invokes a retry (for example, "Try to Sign in again" or "Download Azure AD broker application").
Case 2: Retryable Error	1. Perform a single retry as the end user may have entered a state that results in a success. 2. If retry fails, present the end-user UI based on the specific error that invokes a retry ("Try to Sign in again", "Download Azure AD broker app", etc.).

IMPORTANT

If a user account is passed to ADAL in a silent call and fails, the subsequent interactive request allows the end user to sign in using a different account. After a successful AcquireToken using a user account, the application must verify the signed-in user matches the application's local user object. A mismatch does not generate an exception (except in Objective C), but should be considered in cases where a user is known locally before the authentication requests (like a failed silent call).

.NET

The following guidance provides examples for error handling in conjunction with all non-silent AcquireToken(...) ADAL methods, *except*:

- AcquireTokenAsync(..., IClientAssertionCertification, ...)
- AcquireTokenAsync(..., ClientCredential, ...)
- AcquireTokenAsync(..., ClientAssertion, ...)
- AcquireTokenAsync(..., UserAssertion,...)

Your code would be implemented as follows:

```

try {
    AcquireTokenAsync(...);
}

catch(AdalServiceException e) {
    // Exception: AdalServiceException
    // Represents an error produced by the STS.
    // e.ErrorCode contains the error code and description, which can be used for debugging.
    // NOTE: Do not code a dependency on the contents of the error description, as it can change over time.

    // Design time consideration: Certain errors may be caused at development and exposed through this
exception.

    // Looking inside the description will give more guidance on resolving the specific issue.

    // Action: Case 1: Non-Retryable
    // Do not perform an immediate retry. Only retry after user action.
    // Example Errors: default case

}

catch (AdalException e) {
    // Exception: AdalException
    // Represents a library exception generated by ADAL .NET.
    // e.ErrorCode contains the error code

    // Action: Case 1, Non-Retryable
    // Do not perform an immediate retry. Only retry after user action.
    // Example Errors: network_not_available, default case
}

```

NOTE

ADAL .NET has an extra consideration as it supports PromptBehavior.Never, which has behavior like AcquireTokenSilent.

The following guidance provides examples for error handling in conjunction with ADAL methods:

- acquireToken(..., PromptBehavior.Never)

Your code would be implemented as follows:

```

        try {acquireToken(..., PromptBehavior.Never);
    }

catch(AdalServiceException e) {
    // Exception: AdalServiceException represents
    // Represents an error produced by the STS.
    // e.ErrorCode contains the error code and description, which can be used for debugging.
    // NOTE: Do not code a dependency on the contents of the error description, as it can change over time.

    // Action: Case 1: Non-Retryable
    // Do not perform an immediate retry. Only retry after user action.
    // Example Errors: default case

} catch (AdalException e) {
    // Error Code: e.ErrorCode == "user_interaction_required"
    // Description: user_interaction_required indicates the silent request failed
    // in a way that's resolvable with an interactive request.
    // Action: Resolvable with an interactive request.

    // Action: Case 1, Non-Retryable
    // Do not perform an immediate retry. Only retry after user action.
    // Example Errors: network_not_available, default case
}

}

```

Android

The following guidance provides examples for error handling in conjunction with all non-silent AcquireToken(...) ADAL methods.

Your code would be implemented as follows:

```

AcquireTokenAsync(...);

// *Inside callback*
public void onError(Exception e) {
    if (e instanceof AuthenticationException) {
        // Exception: AdalException
        // Represents a library exception generated by ADAL Android.
        // Error Code: e.getCode();

        // Error: ADALError.BROKER_APP_INSTALLATION_STARTED
        // Description: Broker app not installed, user will be prompted to download the app.

        // Action: Case 2, Retriable Error
        // Perform a single retry. If that fails, only try again after user action.

        // Action: Case 1, Non-Retriable
        // Do not perform an immediate retry. Only retry after user action.
        // Example Errors: default case, DEVICE_CONNECTION_IS_NOT_AVAILABLE
    }
}

```

iOS

The following guidance provides examples for error handling in conjunction with all non-silent AcquireToken(...) ADAL methods.

Your code would be implemented as follows:

```
[context acquireTokenWithResource:[ARGS], completionBlock:^(ADAuthenticationResult *result) {
    if (result.status == AD_FAILED) {
        if ([error.domain isEqualToString:ADAuthenticationErrorDomain]){
            // Exception: AD_FAILED
            // Represents a library error generated by ADAL ObjC.
            // Error Code: result.error.code

            // Error: AD_ERROR_SERVER_WRONG_USER
            // Description: App passed a user into ADAL and the end user signed in with a different account.
            // Action: Case 1, Non-retriable (as is) and up to the application on how to handle this case.
            // It can attempt a new request without specifying the user, or use UI to clarify the user account
            to sign in.

            // Action: Case 1, Non-Retriable
            // Do not perform an immediate retry. Only retry after user action.
            // Example Errors: default case
        }
    }
}]]
```

Error cases and actionable steps: Web applications that call a resource API (.NET)

If you're building a .NET web app that calls gets a token using an authorization code for a resource, the only code required is a default handler for the generic case.

The following guidance provides examples for error handling in conjunction with ADAL methods:

- `AcquireTokenByAuthorizationCodeAsync(...)`

Your code would be implemented as follows:

```
try {
    AcquireTokenByAuthorizationCodeAsync(...);
}

catch (AdalException e) {
    // Exception: AdalException
    // Represents a library exception generated by ADAL .NET.
    // Error Code: e.ErrorCode

    // Action: Do not perform an immediate retry. Only try again after user action or wait until much later.
    // Example Errors: default case
}
```

Error cases and actionable steps: Single-page applications (adal.js)

If you're building a single-page application using adal.js with `AcquireToken`, the error handling code is similar to that of a typical silent call. Specifically in adal.js, `AcquireToken` never shows a UI.

A failed `AcquireToken` has the following cases:

Case 1: Resolvable with an interactive request	1. If <code>login()</code> fails, do not perform immediate retry. Only retry after user action prompts a retry.
Case 2: Not Resolvable with an interactive request. Error is retriable.	<ol style="list-style-type: none"> 1. Perform a single retry as the end user major have entered a state that results in a success. 2. If retry fails, present the end user with an action based on the specific error that can invoke a retry ("Try to Sign in again").

Case 3:

Not Resolvable with an interactive request. Error is not retryable.

1. Do not attempt immediate retry. Present the end user with an action based on the specific error that can invoke a retry ("Try to Sign in again").

Your code would be implemented as follows:

```
AuthContext.acquireToken(..., function(error, errorDesc, token) {  
    if (error || errorDesc) {  
        // Represents any token acquisition failure that occurred.  
        // Error Code: error.indexOf("<ERROR_STRING>")  
  
        // Errors: if (error.indexOf("interaction_required"))  
        //           if (error.indexOf("login required"))  
        // Description: ADAL wasn't able to silently acquire a token because of expire or fresh session.  
        // Action: Case 1, Resolvable with an interactive login() request.  
  
        // Error: if (error.indexOf("Token Renewal Failed"))  
        // Description: Timeout when refreshing the token.  
        // Action: Case 2, Not resolvable interactively, error is retriable.  
        // Perform a single retry. Only try again after user action.  
  
        // Action: Case 3, Not resolvable interactively, error is not retriable.  
        // Do not perform an immediate retry. Only retry after user action.  
        // Example Errors: default case  
    }  
}
```

Error cases and actionable steps: service-to-service applications (.NET only)

If you're building a service-to-service application that uses AcquireToken, there are a few key errors your code must handle. The only recourse to failure is to return the error back to the calling app (for on-behalf-of cases) or apply a retry strategy.

All scenarios

For *all* service-to-service application scenarios, including on-behalf-of:

- Do not attempt an immediate retry. ADAL attempts a single retry for certain failed requests.
- Only continue retrying after a user or app action is prompts a retry. For example, a daemon application that does work on some set interval should wait until the next interval to retry.

The following guidance provides examples for error handling in conjunction with ADAL methods:

- AcquireTokenAsync(..., IClientAssertionCertification, ...)
- AcquireTokenAsync(..., ClientCredential, ...)
- AcquireTokenAsync(..., ClientAssertion, ...)
- AcquireTokenAsync(..., UserAssertion, ...)

Your code would be implemented as follows:

```

try {
    AcquireTokenAsync(...);
}

catch (AdalException e) {
    // Exception: AdalException
    // Represents a library exception generated by ADAL .NET.
    // Error Code: e.ErrorCode

    // Action: Do not perform an immediate retry. Only try again after user action (if applicable) or wait
    until much later.
    // Example Errors: default case
}

```

On-behalf-of scenarios

For *on-behalf-of* service-to-service application scenarios.

The following guidance provides examples for error handling in conjunction with ADAL methods:

- `AcquireTokenAsync(..., UserAssertion, ...)`

Your code would be implemented as follows:

```

try {
    AcquireTokenAsync(...);
}

catch (AdalServiceException e) {
    // Exception: AdalServiceException
    // Represents an error produced by the STS.
    // e.ErrorCode contains the error code and description, which can be used for debugging.
    // NOTE: Do not code a dependency on the contents of the error description, as it can change over time.

    // Error: On-Behalf-Of Error Handler
    if (e.ErrorCode == "interaction_required") {
        // Description: The client needs to perform some action due to a config from admin.
        // Action: Capture `claims` parameter inside ex.InnerException.InnerException.
        // Generate HTTP error 403 with claims, throw back HTTP error to client.
        // Wait for client to retry.
    }
}

catch (AdalException e) {
    // Exception: AdalException
    // Represents a library exception generated by ADAL .NET.
    // Error Code: e.ErrorCode

    // Action: Do not perform an immediate retry. Only try again after user action (if applicable) or wait
    until much later.
    // Example Error: default case
}

```

We've built a [complete sample](#) that demonstrates this scenario.

Error and logging reference

Logging Personal Identifiable Information & Organizational Identifiable Information

By default, ADAL logging does not capture or log any personal identifiable information or organizational identifiable information. The library allows app developers to turn this on through a setter in the Logger class. By logging personal identifiable information or organizational identifiable information, the app takes responsibility for safely handling highly sensitive data and complying with any regulatory requirements.

.NET

ADAL library errors

To explore specific ADAL errors, the source code in the [azure-active-directory-library-for-dotnet repository](#) is the best error reference.

Guidance for error logging code

ADAL .NET logging changes depending on the platform being worked on. Refer to the [Logging wiki](#) for code on how to enable logging.

Android

ADAL library errors

To explore specific ADAL errors, the source code in the [azure-active-directory-library-for-android repository](#) is the best error reference.

Operating System errors

Android OS errors are exposed through AuthenticationException in ADAL, are identifiable as "SERVER_INVALID_REQUEST", and can be further granular through the error descriptions.

For a full list of common errors and what steps to take when your app or end users encounter them, refer to the [ADAL Android Wiki](#).

Guidance for error logging code

```
// 1. Configure Logger
Logger.getInstance().setExternalLogger(new ILogger() {
    @Override
    public void Log(String tag, String message, String additionalMessage, LogLevel level, ADALError errorCode)
    {
        // ...
        // You can write this to logfile depending on level or errorcode.
        writeToFile(getApplicationContext(), tag + ":" + message + "-" + additionalMessage);
    }
}

// 2. Set the log level
Logger.getInstance().setLogLevel(Logger.LogLevel.Verbose);

// By default, the `Logger` does not capture any PII or OII

// PII or OII will be logged
Logger.getInstance().setEnablePII(true);

// To STOP logging PII or OII, use the following setter
Logger.getInstance().setEnablePII(false);

// 3. Send logs to logcat.
adb logcat > "C:\logmsg\logfile.txt";
```

iOS

ADAL library errors

To explore specific ADAL errors, the source code in the [azure-active-directory-library-for-objc repository](#) is the best error reference.

Operating System errors

iOS errors may arise during sign-in when users use web views, and the nature of authentication. This can be caused by conditions such as SSL errors, timeouts, or network errors:

- For Entitlement Sharing, logins are not persistent and the cache appears empty. You can resolve by adding the following line of code to the keychain:

```
[[ADAuthenticationSettings sharedInstance] setSharedCacheKeychainGroup:nil];
```

- For the NsUrlDomain set of errors, the action changes depending on the app logic. See the [NSURLErrorDomain reference documentation](#) for specific instances that can be handled.
- See [ADAL Obj-C Common Issues](#) for the list of common errors maintained by the ADAL Objective-C team.

Guidance for error logging code

```
// 1. Enable NSLogging
[ADLogger setNSLogging:YES];

// 2. Set the log level (if you want verbose)
[ADLogger setLevel:ADAL_LOG_LEVEL_VERBOSE];

// 3. Set up a callback block to simply print out
[ADLogger setLogCallBack:^(ADAL_LOG_LEVEL logLevel, NSString *message, NSString *additionalInformation,
NSInteger errorCode, NSDictionary *userInfo) {
    NSString* log = [NSString stringWithFormat:@"%@", message, additionalInformation];
    NSLog(@"%@", log);
}];
```

Guidance for error logging code - JavaScript

```
0: Error1: Warning2: Info3: Verbose
window.Logging = {
  level: 3,
  log: function (message) {
    console.log(message);
  }
};
```

Related content

- [Azure AD Developer's Guide](#)
- [Azure AD Authentication Libraries](#)
- [Azure AD Authentication Scenarios](#)
- [Integrating Applications with Azure Active Directory](#)

Use the comments section that follows, to provide feedback and help us refine and shape our content.



How to: Sign in any Azure Active Directory user using the multi-tenant application pattern

10/23/2019 • 13 minutes to read • [Edit Online](#)

If you offer a Software as a Service (SaaS) application to many organizations, you can configure your application to accept sign-ins from any Azure Active Directory (Azure AD) tenant. This configuration is called *making your application multi-tenant*. Users in any Azure AD tenant will be able to sign in to your application after consenting to use their account with your application.

If you have an existing application that has its own account system, or supports other kinds of sign-ins from other cloud providers, adding Azure AD sign-in from any tenant is simple. Just register your app, add sign-in code via OAuth2, OpenID Connect, or SAML, and put a "Sign in with Microsoft" button in your application.

NOTE

This article assumes you're already familiar with building a single tenant application for Azure AD. If you're not, start with one of the quickstarts on the [developer guide homepage](#).

There are four simple steps to convert your application into an Azure AD multi-tenant app:

1. [Update your application registration to be multi-tenant](#)
2. [Update your code to send requests to the /common endpoint](#)
3. [Update your code to handle multiple issuer values](#)
4. [Understand user and admin consent and make appropriate code changes](#)

Let's look at each step in detail. You can also jump straight to [this list of multi-tenant samples](#).

Update registration to be multi-tenant

By default, web app/API registrations in Azure AD are single tenant. You can make your registration multi-tenant by finding the **Supported account types** switch on the **Authentication** pane of your application registration in the [Azure portal](#) and setting it to **Accounts in any organizational directory**.

Before an application can be made multi-tenant, Azure AD requires the App ID URI of the application to be globally unique. The App ID URI is one of the ways an application is identified in protocol messages. For a single tenant application, it is sufficient for the App ID URI to be unique within that tenant. For a multi-tenant application, it must be globally unique so Azure AD can find the application across all tenants. Global uniqueness is enforced by requiring the App ID URI to have a host name that matches a verified domain of the Azure AD tenant.

By default, apps created via the Azure portal have a globally unique App ID URI set on app creation, but you can change this value. For example, if the name of your tenant was contoso.onmicrosoft.com then a valid App ID URI would be `https://contoso.onmicrosoft.com/myapp`. If your tenant had a verified domain of `contoso.com`, then a valid App ID URI would also be `https://contoso.com/myapp`. If the App ID URI doesn't follow this pattern, setting an application as multi-tenant fails.

NOTE

Native client registrations as well as [Microsoft identity platform applications](#) are multi-tenant by default. You don't need to take any action to make these application registrations multi-tenant.

Update your code to send requests to /common

In a single tenant application, sign-in requests are sent to the tenant's sign-in endpoint. For example, for contoso.onmicrosoft.com the endpoint would be: <https://login.microsoftonline.com/contoso.onmicrosoft.com>. Requests sent to a tenant's endpoint can sign in users (or guests) in that tenant to applications in that tenant.

With a multi-tenant application, the application doesn't know up front what tenant the user is from, so you can't send requests to a tenant's endpoint. Instead, requests are sent to an endpoint that multiplexes across all Azure AD tenants: <https://login.microsoftonline.com/common>

When Microsoft identity platform receives a request on the /common endpoint, it signs the user in and, as a consequence, discovers which tenant the user is from. The /common endpoint works with all of the authentication protocols supported by the Azure AD: OpenID Connect, OAuth 2.0, SAML 2.0, and WS-Federation.

The sign-in response to the application then contains a token representing the user. The issuer value in the token tells an application what tenant the user is from. When a response returns from the /common endpoint, the issuer value in the token corresponds to the user's tenant.

IMPORTANT

The /common endpoint is not a tenant and is not an issuer, it's just a multiplexer. When using /common, the logic in your application to validate tokens needs to be updated to take this into account.

Update your code to handle multiple issuer values

Web applications and web APIs receive and validate tokens from Microsoft identity platform.

NOTE

While native client applications request and receive tokens from Microsoft identity platform, they do so to send them to APIs, where they are validated. Native applications do not validate tokens and must treat them as opaque.

Let's look at how an application validates tokens it receives from Microsoft identity platform. A single tenant application normally takes an endpoint value like:

<https://login.microsoftonline.com/contoso.onmicrosoft.com>

and uses it to construct a metadata URL (in this case, OpenID Connect) like:

<https://login.microsoftonline.com/contoso.onmicrosoft.com/.well-known/openid-configuration>

to download two critical pieces of information that are used to validate tokens: the tenant's signing keys and issuer value. Each Azure AD tenant has a unique issuer value of the form:

```
https://sts.windows.net/31537af4-6d77-4bb9-a681-d2394888ea26/
```

where the GUID value is the rename-safe version of the tenant ID of the tenant. If you select the preceding metadata link for `contoso.onmicrosoft.com`, you can see this issuer value in the document.

When a single tenant application validates a token, it checks the signature of the token against the signing keys from the metadata document. This test allows it to make sure the issuer value in the token matches the one that was found in the metadata document.

Because the /common endpoint doesn't correspond to a tenant and isn't an issuer, when you examine the issuer value in the metadata for /common it has a templated URL instead of an actual value:

```
https://sts.windows.net/{tenantid}/
```

Therefore, a multi-tenant application can't validate tokens just by matching the issuer value in the metadata with the `issuer` value in the token. A multi-tenant application needs logic to decide which issuer values are valid and which are not based on the tenant ID portion of the issuer value.

For example, if a multi-tenant application only allows sign-in from specific tenants who have signed up for their service, then it must check either the issuer value or the `tid` claim value in the token to make sure that tenant is in their list of subscribers. If a multi-tenant application only deals with individuals and doesn't make any access decisions based on tenants, then it can ignore the issuer value altogether.

In the [multi-tenant samples](#), issuer validation is disabled to enable any Azure AD tenant to sign in.

Understand user and admin consent

For a user to sign in to an application in Azure AD, the application must be represented in the user's tenant. This allows the organization to do things like apply unique policies when users from their tenant sign in to the application. For a single tenant application, this registration is simple; it's the one that happens when you register the application in the [Azure portal](#).

For a multi-tenant application, the initial registration for the application lives in the Azure AD tenant used by the developer. When a user from a different tenant signs in to the application for the first time, Azure AD asks them to consent to the permissions requested by the application. If they consent, then a representation of the application called a *service principal* is created in the user's tenant, and sign-in can continue. A delegation is also created in the directory that records the user's consent to the application. For details on the application's Application and ServicePrincipal objects, and how they relate to each other, see [Application objects and service principal objects](#).

This consent experience is affected by the permissions requested by the application. Microsoft identity platform supports two kinds of permissions, app-only and delegated.

- A delegated permission grants an application the ability to act as a signed in user for a subset of the things the user can do. For example, you can grant an application the delegated permission to read the signed in user's calendar.
- An app-only permission is granted directly to the identity of the application. For example, you can grant an application the app-only permission to read the list of users in a tenant, regardless of who is signed in to the application.

Some permissions can be consented to by a regular user, while others require a tenant administrator's consent.

Admin consent

App-only permissions always require a tenant administrator's consent. If your application requests an app-only permission and a user tries to sign in to the application, an error message is displayed saying the user isn't able to consent.

Certain delegated permissions also require a tenant administrator's consent. For example, the ability to write back to Azure AD as the signed in user requires a tenant administrator's consent. Like app-only permissions, if an ordinary user tries to sign in to an application that requests a delegated permission that requires administrator consent, your application receives an error. Whether a permission requires admin consent is determined by the developer that published the resource, and can be found in the documentation for the resource. The permissions documentation for the [Azure AD Graph API](#) and [Microsoft Graph API](#) indicate which permissions require admin consent.

If your application uses permissions that require admin consent, you need to have a gesture such as a button or link where the admin can initiate the action. The request your application sends for this action is the usual OAuth2/OpenID Connect authorization request that also includes the `prompt=admin_consent` query string parameter. Once the admin has consented and the service principal is created in the customer's tenant, subsequent sign-in requests do not need the `prompt=admin_consent` parameter. Since the administrator has decided the requested permissions are acceptable, no other users in the tenant are prompted for consent from that point forward.

A tenant administrator can disable the ability for regular users to consent to applications. If this capability is disabled, admin consent is always required for the application to be used in the tenant. If you want to test your application with end-user consent disabled, you can find the configuration switch in the [Azure portal](#) in the **User settings** section under **Enterprise applications**.

The `prompt=admin_consent` parameter can also be used by applications that request permissions that do not require admin consent. An example of when this would be used is if the application requires an experience where the tenant admin "signs up" one time, and no other users are prompted for consent from that point on.

If an application requires admin consent and an admin signs in without the `prompt=admin_consent` parameter being sent, when the admin successfully consents to the application it will apply **only for their user account**. Regular users will still not be able to sign in or consent to the application. This feature is useful if you want to give the tenant administrator the ability to explore your application before allowing other users access.

NOTE

Some applications want an experience where regular users are able to consent initially, and later the application can involve the administrator and request permissions that require admin consent. There is no way to do this with a v1.0 application registration in Azure AD today; however, using the Microsoft identity platform (v2.0) endpoint allows applications to request permissions at runtime instead of at registration time, which enables this scenario. For more information, see [Microsoft identity platform endpoint](#).

Consent and multi-tier applications

Your application may have multiple tiers, each represented by its own registration in Azure AD. For example, a native application that calls a web API, or a web application that calls a web API. In both of these cases, the client (native app or web app) requests permissions to call the resource (web API). For the client to be successfully consented into a customer's tenant, all resources to which it requests permissions must already exist in the customer's tenant. If this condition isn't met, Azure AD returns an error that the resource must be added first.

Multiple tiers in a single tenant

This can be a problem if your logical application consists of two or more application registrations, for example a separate client and resource. How do you get the resource into the customer tenant first? Azure AD covers this case by enabling client and resource to be consented in a single step. The user sees the sum total of the permissions requested by both the client and resource on the consent page. To enable this behavior, the

resource's application registration must include the client's App ID as a `knownClientApplications` in its [application manifest](#). For example:

```
knownClientApplications": ["94da0930-763f-45c7-8d26-04d5938baab2"]
```

This is demonstrated in a multi-tier native client calling web API sample in the [Related content](#) section at the end of this article. The following diagram provides an overview of consent for a multi-tier app registered in a single tenant.

Multiple tiers in multiple tenants

A similar case happens if the different tiers of an application are registered in different tenants. For example, consider the case of building a native client application that calls the Office 365 Exchange Online API. To develop the native application, and later for the native application to run in a customer's tenant, the Exchange Online service principal must be present. In this case, the developer and customer must purchase Exchange Online for the service principal to be created in their tenants.

If it's an API built by an organization other than Microsoft, the developer of the API needs to provide a way for their customers to consent the application into their customers' tenants. The recommended design is for the third-party developer to build the API such that it can also function as a web client to implement sign-up. To do this:

1. Follow the earlier sections to ensure the API implements the multi-tenant application registration/code requirements.
2. In addition to exposing the API's scopes/roles, make sure the registration includes the "Sign in and read user profile" permission (provided by default).
3. Implement a sign-in/sign-up page in the web client and follow the [admin consent](#) guidance.
4. Once the user consents to the application, the service principal and consent delegation links are created in their tenant, and the native application can get tokens for the API.

The following diagram provides an overview of consent for a multi-tier app registered in different tenants.

Revoking consent

Users and administrators can revoke consent to your application at any time:

- Users revoke access to individual applications by removing them from their [Access Panel Applications](#) list.
- Administrators revoke access to applications by removing them using the [Enterprise applications](#) section of the [Azure portal](#).

If an administrator consents to an application for all users in a tenant, users cannot revoke access individually. Only the administrator can revoke access, and only for the whole application.

Multi-tenant applications and caching access tokens

Multi-tenant applications can also get access tokens to call APIs that are protected by Azure AD. A common error when using the Active Directory Authentication Library (ADAL) with a multi-tenant application is to initially request a token for a user using /common, receive a response, then request a subsequent token for that same user also using /common. Because the response from Azure AD comes from a tenant, not /common, ADAL caches the token as being from the tenant. The subsequent call to /common to get an access token for the user misses the cache entry, and the user is prompted to sign in again. To avoid missing the cache, make sure subsequent calls for an already signed in user are made to the tenant's endpoint.

Next steps

In this article, you learned how to build an application that can sign in a user from any Azure AD tenant. After enabling Single Sign-On (SSO) between your app and Azure AD, you can also update your application to access APIs exposed by Microsoft resources like Office 365. This lets you offer a personalized experience in your application, such as showing contextual information to the users, like their profile picture or their next calendar appointment. To learn more about making API calls to Azure AD and Office 365 services like Exchange, SharePoint, OneDrive, OneNote, and more, visit [Microsoft Graph API](#).

Related content

- [Multi-tenant application samples](#)
- [Branding guidelines for applications](#)
- [Application objects and service principal objects](#)
- [Integrating applications with Azure Active Directory](#)
- [Overview of the Consent Framework](#)
- [Microsoft Graph API permission scopes](#)
- [Azure AD Graph API permission scopes](#)

How to: Use Azure PowerShell to create a service principal with a certificate

10/23/2019 • 6 minutes to read • [Edit Online](#)

When you have an app or script that needs to access resources, you can set up an identity for the app and authenticate the app with its own credentials. This identity is known as a service principal. This approach enables you to:

- Assign permissions to the app identity that are different than your own permissions. Typically, these permissions are restricted to exactly what the app needs to do.
- Use a certificate for authentication when executing an unattended script.

IMPORTANT

Instead of creating a service principal, consider using managed identities for Azure resources for your application identity. If your code runs on a service that supports managed identities and accesses resources that support Azure Active Directory (Azure AD) authentication, managed identities are a better option for you. To learn more about managed identities for Azure resources, including which services currently support it, see [What is managed identities for Azure resources?](#).

This article shows you how to create a service principal that authenticates with a certificate. To set up a service principal with password, see [Create an Azure service principal with Azure PowerShell](#).

You must have the [latest version](#) of PowerShell for this article.

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

Required permissions

To complete this article, you must have sufficient permissions in both your Azure AD and Azure subscription. Specifically, you must be able to create an app in the Azure AD, and assign the service principal to a role.

The easiest way to check whether your account has adequate permissions is through the portal. See [Check required permission](#).

Assign the application to a role

To access resources in your subscription, you must assign the application to a role. Decide which role offers the right permissions for the application. To learn about the available roles, see [RBAC: Built in Roles](#).

You can set the scope at the level of the subscription, resource group, or resource. Permissions are inherited to lower levels of scope. For example, adding an application to the *Reader* role for a resource group means it can read the resource group and any resources it contains. To allow the application to execute actions like reboot, start and stop instances, select the *Contributor* role.

Create service principal with self-signed certificate

The following example covers a simple scenario. It uses [New-AzServicePrincipal](#) to create a service principal with a self-signed certificate, and uses [New-AzureRmRoleAssignment](#) to assign the [Reader](#) role to the service principal. The role assignment is scoped to your currently selected Azure subscription. To select a different subscription, use [Set-AzContext](#).

NOTE

The New-SelfSignedCertificate cmdlet and the PKI module are currently not supported in PowerShell Core.

```
$cert = New-SelfSignedCertificate -CertStoreLocation "cert:\CurrentUser\My" `  
    -Subject "CN=exampleappScriptCert" `  
    -KeySpec KeyExchange  
$keyValue = [System.Convert]::ToBase64String($cert.GetRawCertData())  
  
$sp = New-AzServicePrincipal -DisplayName exampleapp `  
    -CertValue $keyValue `  
    -EndDate $cert.NotAfter `  
    -StartDate $cert.NotBefore  
Sleep 20  
New-AzRoleAssignment -RoleDefinitionName Reader -ServicePrincipalName $sp.ApplicationId
```

The example sleeps for 20 seconds to allow some time for the new service principal to propagate throughout Azure AD. If your script doesn't wait long enough, you'll see an error stating: "Principal {ID} does not exist in the directory {DIR-ID}." To resolve this error, wait a moment then run the [New-AzRoleAssignment](#) command again.

You can scope the role assignment to a specific resource group by using the **ResourceGroupName** parameter. You can scope to a specific resource by also using the **ResourceType** and **ResourceName** parameters.

If you **do not have Windows 10 or Windows Server 2016**, you need to download the [Self-signed certificate generator](#) from Microsoft Script Center. Extract its contents and import the cmdlet you need.

```
# Only run if you could not use New-SelfSignedCertificate  
Import-Module -Name c:\ExtractedModule\New-SelfSignedCertificateEx.ps1
```

In the script, substitute the following two lines to generate the certificate.

```
New-SelfSignedCertificateEx -StoreLocation CurrentUser `  
    -Subject "CN=exampleapp" `  
    -KeySpec "Exchange" `  
    -FriendlyName "exampleapp"  
$cert = Get-ChildItem -path Cert:\CurrentUser\my | where {$PSitem.Subject -eq 'CN=exampleapp' }
```

Provide certificate through automated PowerShell script

Whenever you sign in as a service principal, you need to provide the tenant ID of the directory for your AD app. A tenant is an instance of Azure AD.

```

$TenantId = (Get-AzSubscription -SubscriptionName "Contoso Default").TenantId
$ApplicationId = (Get-AzADApplication -DisplayNameStartWith exampleapp).ApplicationId

$Thumbprint = (Get-ChildItem cert:\CurrentUser\My\ | Where-Object {$_.Subject -eq "CN=exampleappScriptCert"}).Thumbprint
Connect-AzAccount -ServicePrincipal ` 
-CertificateThumbprint $Thumbprint ` 
-ApplicationId $ApplicationId ` 
-TenantId $TenantId

```

Create service principal with certificate from Certificate Authority

The following example uses a certificate issued from a Certificate Authority to create service principal. The assignment is scoped to the specified Azure subscription. It adds the service principal to the [Reader](#) role. If an error occurs during the role assignment, it retries the assignment.

```

Param (
    [Parameter(Mandatory=$true)]
    [String] $ApplicationDisplayName,
    [Parameter(Mandatory=$true)]
    [String] $SubscriptionId,
    [Parameter(Mandatory=$true)]
    [String] $CertPath,
    [Parameter(Mandatory=$true)]
    [String] $CertPlainPassword
)

Connect-AzAccount
Import-Module Az.Resources
Set-AzContext -Subscription $SubscriptionId

$CertPassword = ConvertTo-SecureString $CertPlainPassword -AsPlainText -Force

$PFXCert = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Certificate2 -ArgumentList
@($CertPath, $CertPassword)
$keyValue = [System.Convert]::ToBase64String($PFXCert.GetRawCertData())

$ServicePrincipal = New-AzADServicePrincipal -DisplayName $ApplicationDisplayName
New-AzADSpCredential -ObjectId $ServicePrincipal.Id -CertValue $keyValue -StartDate $PFXCert.NotBefore - 
EndDate $PFXCert.NotAfter
Get-AzADServicePrincipal -ObjectId $ServicePrincipal.Id

$NewRole = $null
$Retries = 0;
While ($NewRole -eq $null -and $Retries -le 6)
{
    # Sleep here for a few seconds to allow the service principal application to become active (should only
    # take a couple of seconds normally)
    Sleep 15
    New-AzRoleAssignment -RoleDefinitionName Reader -ServicePrincipalName $ServicePrincipal.ApplicationId |
    Write-Verbose -ErrorAction SilentlyContinue
    $NewRole = Get-AzRoleAssignment -ObjectId $ServicePrincipal.Id -ErrorAction SilentlyContinue
    $Retries++;
}

$NewRole

```

Provide certificate through automated PowerShell script

Whenever you sign in as a service principal, you need to provide the tenant ID of the directory for your AD app. A

tenant is an instance of Azure AD.

```
Param (

    [Parameter(Mandatory=$true)]
    [String] $CertPath,

    [Parameter(Mandatory=$true)]
    [String] $CertPlainPassword,

    [Parameter(Mandatory=$true)]
    [String] $ApplicationId,

    [Parameter(Mandatory=$true)]
    [String] $TenantId
)

$CertPassword = ConvertTo-SecureString $CertPlainPassword -AsPlainText -Force
$PFXCert = New-Object `-
    -TypeName System.Security.Cryptography.X509Certificates.X509Certificate2 `-
    -ArgumentList @($CertPath, $CertPassword)
$Thumbprint = $PFXCert.Thumbprint

Connect-AzAccount -ServicePrincipal `-
    -CertificateThumbprint $Thumbprint `-
    -ApplicationId $ApplicationId `-
    -TenantId $TenantId
```

The application ID and tenant ID aren't sensitive, so you can embed them directly in your script. If you need to retrieve the tenant ID, use:

```
(Get-AzSubscription -SubscriptionName "Contoso Default").TenantId
```

If you need to retrieve the application ID, use:

```
(Get-AzADApplication -DisplayNameStartWith {display-name}).ApplicationId
```

Change credentials

To change the credentials for an AD app, either because of a security compromise or a credential expiration, use the [Remove-AzADAppCredential](#) and [New-AzADAppCredential](#) cmdlets.

To remove all the credentials for an application, use:

```
Get-AzADApplication -DisplayName exampleapp | Remove-AzADAppCredential
```

To add a certificate value, create a self-signed certificate as shown in this article. Then, use:

```
Get-AzADApplication -DisplayName exampleapp | New-AzADAppCredential `-
    -CertValue $keyValue `-
    -EndDate $cert.NotAfter `-
    -StartDate $cert.NotBefore
```

Debug

You may get the following errors when creating a service principal:

- "**Authentication_Unauthorized**" or "**No subscription found in the context.**" - You see this error when your account doesn't have the [required permissions](#) on the Azure AD to register an app. Typically, you see this error when only admin users in your Azure Active Directory can register apps, and your account isn't an admin. Ask your administrator to either assign you to an administrator role, or to enable users to register apps.
- Your account "**does not have authorization to perform action** '**Microsoft.Authorization/roleAssignments/write**' over scope '**/subscriptions/{guid}**'." - You see this error when your account doesn't have sufficient permissions to assign a role to an identity. Ask your subscription administrator to add you to User Access Administrator role.

Next steps

- To set up a service principal with password, see [Create an Azure service principal with Azure PowerShell](#).
- For a more detailed explanation of applications and service principals, see [Application Objects and Service Principal Objects](#).
- For more information about Azure AD authentication, see [Authentication Scenarios for Azure AD](#).

How to: Use the portal to create an Azure AD application and service principal that can access resources

11/4/2019 • 6 minutes to read • [Edit Online](#)

This article shows you how to create a new Azure Active Directory (Azure AD) application and service principal that can be used with the role-based access control. When you have code that needs to access or modify resources, you can create an identity for the app. This identity is known as a service principal. You can then assign the required permissions to the service principal. This article shows you how to use the portal to create the service principal. It focuses on a single-tenant application where the application is intended to run within only one organization. You typically use single-tenant applications for line-of-business applications that run within your organization.

IMPORTANT

Instead of creating a service principal, consider using managed identities for Azure resources for your application identity. If your code runs on a service that supports managed identities and accesses resources that support Azure AD authentication, managed identities are a better option for you. To learn more about managed identities for Azure resources, including which services currently support it, see [What is managed identities for Azure resources?](#).

Create an Azure Active Directory application

Let's jump straight into creating the identity. If you run into a problem, check the [required permissions](#) to make sure your account can create the identity.

1. Sign in to your Azure Account through the [Azure portal](#).
2. Select **Azure Active Directory**.
3. Select **App registrations**.
4. Select **New registration**.
5. Name the application. Select a supported account type, which determines who can use the application. Under **Redirect URI**, select **Web** for the type of application you want to create. Enter the URI where the access token is sent to. You can't create credentials for a [Native application](#). You can't use that type for an automated application. After setting the values, select **Register**.

Dashboard > Microsoft - App registrations > Register an application

Register an application

⚠️ If you are building an application for external users that will be distributed by Microsoft, you must register as a first party application to meet all security, privacy, and compliance policies. [Read our decision guide](#)

*** Name**
The user-facing display name for this application (this can be changed later).
 ✓

Supported account types
Who can use this application or access this API?
 Accounts in this organizational directory only (Microsoft)
 Accounts in any organizational directory
 Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)
[Help me choose...](#)

Redirect URI (optional)
We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.
 ✓

By proceeding, you agree to the Microsoft Platform Policies [\[link\]](#)

Register

You've created your Azure AD application and service principal.

Assign the application to a role

To access resources in your subscription, you must assign the application to a role. Decide which role offers the right permissions for the application. To learn about the available roles, see [RBAC: Built in Roles](#).

You can set the scope at the level of the subscription, resource group, or resource. Permissions are inherited to lower levels of scope. For example, adding an application to the Reader role for a resource group means it can read the resource group and any resources it contains.

1. In the Azure portal, select the level of scope you wish to assign the application to. For example, to assign a role at the subscription scope, search for and select **Subscriptions**, or select **Subscriptions** on the **Home** page.

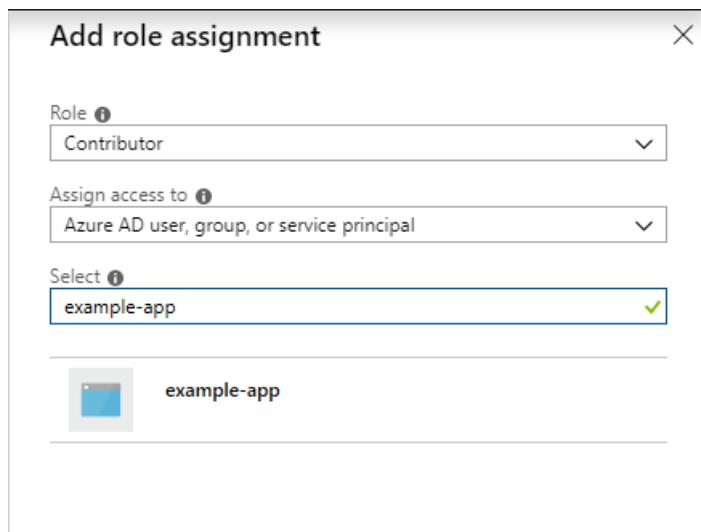
The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with 'Azure services' and 'Recent resources' sections. Below the sidebar, there's a 'Navigate' section with three buttons: 'Subscriptions' (highlighted with a red box), 'Resource groups', and 'All resources'. The main content area is titled 'Subscriptions' and shows a list of items under 'Services'. The first item, 'Subscriptions', is also highlighted with a red box. Other items listed include 'Event Grid Subscriptions', 'Resource groups', and 'Manage subscriptions in the Billing/Account Center'. Below this is a 'Resources' section with a single item: 'APEX C+L - Aquent Vendor Subscriptions'. Under 'Resource Groups', it says 'No results were found.' There's also a 'Documentation' section with links to scaling with multiple subscriptions, Azure subscription limits, and creating an additional subscription.

2. Select the particular subscription to assign the application to.

The screenshot shows the 'Subscriptions' page in the Azure portal. At the top, there's a breadcrumb trail: 'Dashboard > Subscriptions'. The main title is 'Subscriptions' with a 'Microsoft' subtitle. Below that is a 'Add' button. The page displays a list of subscriptions. One subscription, 'Internal testing subscription', is highlighted with a red box. The list includes columns for 'SUBSCRIPTION' and 'SUBSCRIPTION ID'. There are filters at the top: 'My role' (set to '8 selected'), an 'Apply' button, and a checked checkbox for 'Show only subscriptions selected in the global subscriptions filter'. A search bar is also present.

If you don't see the subscription you're looking for, select **global subscriptions filter**. Make sure the subscription you want is selected for the portal.

3. Select **Access control (IAM)**.
4. Select **Add role assignment**.
5. Select the role you wish to assign to the application. For example, to allow the application to execute actions like **reboot**, **start** and **stop** instances, select the **Contributor** role. Read more about the [available roles](#) By default, Azure AD applications aren't displayed in the available options. To find your application, search for the name and select it.



6. Select **Save** to finish assigning the role. You see your application in the list of users assigned to a role for that scope.

Your service principal is set up. You can start using it to run your scripts or apps. The next section shows how to get values that are needed when signing in programmatically.

Get values for signing in

When programmatically signing in, you need to pass the tenant ID with your authentication request. You also need the ID for your application and an authentication key. To get those values, use the following steps:

1. Select **Azure Active Directory**.
2. From **App registrations** in Azure AD, select your application.
3. Copy the Directory (tenant) ID and store it in your application code.

Home > Microsoft - App registrations > example-app

example-app

Search (Ctrl+ /)

Delete Endpoints

Welcome to the new and improved App registrations. Looking to learn how it's changed from

Display name	: example-app
Application (client) ID	: 1A7D7E7E-1A7E-4B41-B7F3-4E04f001cf
Directory (tenant) ID	: 72f93d7f-6c44-4eaf-a218-1a6d61a64717
Object ID	: 12345678-9ABC-4DEF-89AB-123456789012

Copy to clipboard

4. Copy the **Application ID** and store it in your application code.

Home > Microsoft - App registrations > example-app

example-app

Search (Ctrl+ /)

Delete Endpoints

Welcome to the new and improved App registrations. Looking to learn how it's changed from

Display name	: example-app
Application (client) ID	: 1A7D7E7E-1A7E-4B41-B7F3-4E04f001cf
Directory (tenant) ID	: 72f93d7f-6c44-4eaf-a218-1a6d61a64717
Object ID	: 12345678-9ABC-4DEF-89AB-123456789012

Copy to clipboard

Certificates and secrets

Daemon applications can use two forms of credentials to authenticate with Azure AD: certificates and application secrets. We recommend using a certificate, but you can also create a new application secret.

Upload a certificate

You can use an existing certificate if you have one. Optionally, you can create a self-signed certificate for testing purposes. Open PowerShell and run [New-SelfSignedCertificate](#) with the following parameters to create a self-signed certificate in the user certificate store on your computer:

```
$cert = New-SelfSignedCertificate -Subject "CN=DaemonConsoleCert" -CertStoreLocation "Cert:\CurrentUser\My" -KeyExportPolicy Exportable -KeySpec Signature
```

Export this certificate to a file using the [Manage User Certificate](#) MMC snap-in accessible from the Windows Control Panel.

To upload the certificate:

1. Select **Certificates & secrets**.
2. Select **Upload certificate** and select the certificate (an existing certificate or the self-signed certificate you exported).

Certificates

Certificates can be used as secrets to prove the application's identity when requesting a token. Also can be referred to as public keys.

Upload certificate

THUMPRINT	START DATE	EXPIRES

No certificates have been added for this application.

3. Select **Add**.

After registering the certificate with your application in the application registration portal, you need to enable the client application code to use the certificate.

Create a new application secret

If you choose not to use a certificate, you can create a new application secret.

1. Select **Certificates & secrets**.
2. Select **Client secrets -> New client secret**.
3. Provide a description of the secret, and a duration. When done, select **Add**.

After saving the client secret, the value of the client secret is displayed. Copy this value because you aren't able to retrieve the key later. You provide the key value with the application ID to sign in as the application. Store the key value where your application can retrieve it.

Client secrets

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

New client secret

DESCRIPTION	EXPIRES	VALUE
demo secret	5/14/2020	-nWu9HVZ7Rnj.2y7XSkVvUngZ][x9Z:e [REDACTED]

Configure access policies on resources

Keep in mind, you might need to configure addition permissions on resources that your application needs to access. For example, you must also [update a key vault's access policies](#) to give your application access to keys, secrets, or certificates.

1. In the [Azure portal](#), navigate to your key vault and select **Access policies**.
2. Select **Add access policy**, then select the key, secret, and certificate permissions you want to grant your application. Select the service principal you created previously.
3. Select **Add** to add the access policy, then **Save** to commit your changes.

The screenshot shows the 'example-key-vault - Access policies' page in the Azure portal. The left sidebar has 'Access policies' selected. At the top right, there are 'Save', 'Discard', and 'Refresh' buttons. A message says 'Please click 'Save' button to commit your changes.' Below it, 'Enable Access to:' has three checkboxes: 'Azure Virtual Machines for deployment', 'Azure Resource Manager for template deployment', and 'Azure Disk Encryption for volume encryption'. A red box highlights the '+ Add Access Policy...' button. The 'Current Access Policies' table lists one entry: 'example-app' (Category: APPLICATION). Action buttons for 'List', 'Delete', and 'Edit' are shown.

Required permissions

You must have sufficient permissions to register an application with your Azure AD tenant, and assign the application to a role in your Azure subscription.

Check Azure AD permissions

1. Select **Azure Active Directory**.
2. Note your role. If you have the **User** role, you must make sure that non-administrators can register applications.

The screenshot shows the 'Microsoft - Overview' page in Azure Active Directory. The left sidebar has 'Users' selected. At the top right, there are 'Switch directory' and 'Delete directory' buttons. The main area shows 'microsoft.onmicrosoft.com' and 'Microsoft Azure AD Premium P2'. Under 'Sign-ins', it says 'Only global administrators, security administrators, security readers, and report readers can view sign-ins. [More info](#)'. A red box highlights the 'Your role User' section.

3. In the left pane, select **User settings**.
4. Check the **App registrations** setting. This value can only be set by an administrator. If set to **Yes**, any user

in the Azure AD tenant can register an app.

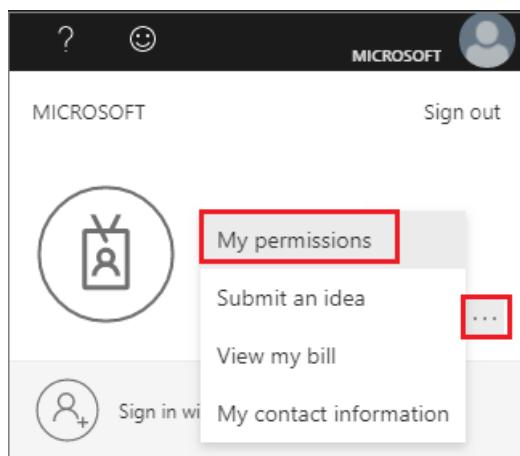
If the app registrations setting is set to **No**, only users with an administrator role may register these types of applications. See [available roles](#) and [role permissions](#) to learn about available administrator roles and the specific permissions in Azure AD that are given to each role. If your account is assigned to the User role, but the app registration setting is limited to admin users, ask your administrator to either assign you to one of the administrator roles that can create and manage all aspects of app registrations, or to enable users to register apps.

Check Azure subscription permissions

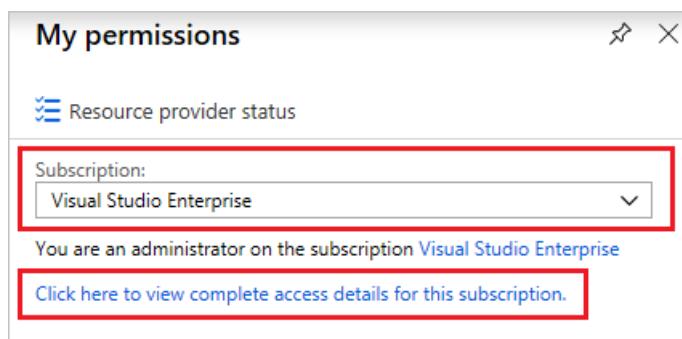
In your Azure subscription, your account must have `Microsoft.Authorization/*/Write` access to assign an AD app to a role. This action is granted through the [Owner](#) role or [User Access Administrator](#) role. If your account is assigned to the **Contributor** role, you don't have adequate permission. You receive an error when attempting to assign the service principal to a role.

To check your subscription permissions:

1. Select your account in the upper right corner, and select... -> **My permissions**.



2. From the drop-down list, select the subscription you want to create the service principal in. Then, select [Click here to view complete access details for this subscription](#).



3. Select **Role assignments** to view your assigned roles, and determine if you have adequate permissions to assign an AD app to a role. If not, ask your subscription administrator to add you to User Access Administrator role. In the following image, the user is assigned to the Owner role, which means that user has adequate permissions.

Add **Remove** **Roles** **Refresh** **Help**

Name **Search by name or email** Type **All** Role **2 selected** Scope **All scopes**

2 items (1 Users, 1 Service Principals)

<input type="checkbox"/> NAME	TYPE	ROLE
OWNER  Example User example@contoso.org	User	Owner, Service administrator

Next steps

- To learn about specifying security policies, see [Azure Role-based Access Control](#).
- For a list of available actions that can be granted or denied to users, see [Azure Resource Manager Resource Provider operations](#).

How to: Restrict your app to a set of users

10/23/2019 • 4 minutes to read • [Edit Online](#)

Applications registered in an Azure Active Directory (Azure AD) tenant are, by default, available to all users of the tenant who authenticate successfully.

Similarly, in case of a [multi-tenant](#) app, all users in the Azure AD tenant where this app is provisioned will be able to access this application once they successfully authenticate in their respective tenant.

Tenant administrators and developers often have requirements where an app must be restricted to a certain set of users. Developers can accomplish the same by using popular authorization patterns like Role Based Access Control (RBAC), but this approach requires a significant amount of work on part of the developer.

Azure AD allows tenant administrators and developers to restrict an app to a specific set of users or security groups in the tenant.

Supported app configurations

The option to restrict an app to a specific set of users or security groups in a tenant works with the following types of applications:

- Applications configured for federated single sign-on with SAML-based authentication
- Application proxy applications that use Azure AD pre-authentication
- Applications built directly on the Azure AD application platform that use OAuth 2.0/OpenID Connect authentication after a user or admin has consented to that application.

NOTE

This feature is available for web app/web API and enterprise applications only. Apps that are registered as [native](#) cannot be restricted to a set of users or security groups in the tenant.

Update the app to enable user assignment

There are two ways to create an application with enabled user assignment. One requires the **Global Administrator** role, the second does not.

Enterprise applications (requires the Global Adminstrator role)

1. Go to the [Azure portal](#) and sign in as a **Global Administrator**.
2. On the top bar, select the signed-in account.
3. Under **Directory**, select the Azure AD tenant where the app will be registered.
4. In the navigation on the left, select **Azure Active Directory**. If Azure Active Directory is not available in the navigation pane, follow these steps:
 - a. Select **All services** at the top of the main left-hand navigation menu.
 - b. Type in **Azure Active Directory** in the filter search box, and then select the **Azure Active Directory** item from the result.
5. In the **Azure Active Directory** pane, select **Enterprise Applications** from the **Azure Active Directory** left-hand navigation menu.

6. Select **All Applications** to view a list of all your applications.

If you do not see the application you want show up here, use the various filters at the top of the **All applications** list to restrict the list or scroll down the list to locate your application.

7. Select the application you want to assign a user or security group to from the list.

8. On the application's **Overview** page, select **Properties** from the application's left-hand navigation menu.

9. Locate the setting **User assignment required?** and set it to **Yes**. When this option is set to **Yes**, users must first be assigned to this application before they can access it.

10. Select **Save** to save this configuration change.

App registration

1. Go to the [Azure portal](#).
2. On the top bar, select the signed-in account.
3. Under **Directory**, select the Azure AD tenant where the app will be registered.
4. In the navigation on the left, select **Azure Active Directory**.
5. In the **Azure Active Directory** pane, select **App Registrations** from the **Azure Active Directory** left-hand navigation menu.
6. Create or select the app you want to manage. You need to be **Owner** of this app registration.
7. On the application's **Overview** page, follow the **Managed application in local directory** link under the essentials in the top of the page. This will take you to the *managed Enterprise Application* of your app registration.
8. From the navigation blade on the left, select **Properties**.
9. Locate the setting **User assignment required?** and set it to **Yes**. When this option is set to **Yes**, users must first be assigned to this application before they can access it.
10. Select **Save** to save this configuration change.

Assign users and groups to the app

Once you've configured your app to enable user assignment, you can go ahead and assign users and groups to the app.

1. Select the **Users and groups** pane in the application's left-hand navigation menu.
2. At the top of the **Users and groups** list, select the **Add user** button to open the **Add Assignment** pane.
3. Select the **Users** selector from the **Add Assignment** pane.

A list of users and security groups will be shown along with a textbox to search and locate a certain user or group. This screen allows you to select multiple users and groups in one go.

4. Once you are done selecting the users and groups, press the **Select** button on bottom to move to the next part.
5. Press the **Assign** button on the bottom to finish the assignments of users and groups to the app.
6. Confirm that the users and groups you added are showing up in the updated **Users and groups** list.

How to: Add app roles in your application and receive them in the token

11/4/2019 • 4 minutes to read • [Edit Online](#)

Role-based access control (RBAC) is a popular mechanism to enforce authorization in applications. When using RBAC, an administrator grants permissions to roles, and not to individual users or groups. The administrator can then assign roles to different users and groups to control who has access to what content and functionality.

Using RBAC with Application Roles and Role Claims, developers can securely enforce authorization in their apps with little effort on their part.

Another approach is to use Azure AD Groups and Group Claims, as shown in [WebApp-GroupClaims-DotNet](#). Azure AD Groups and Application Roles are by no means mutually exclusive; they can be used in tandem to provide even finer grained access control.

Declare roles for an application

These application roles are defined in the [Azure portal](#) in the application's registration manifest. When a user signs into the application, Azure AD emits a `roles` claim for each role that the user has been granted individually to the user and from their group membership. Assignment of users and groups to roles can be done through the portal's UI, or programmatically using [Microsoft Graph](#).

Declare app roles using Azure portal

1. Sign in to the [Azure portal](#).
2. On the top bar, select your account, and then **Switch Directory**.
3. Once the **Directory + subscription** pane opens, choose the Active Directory tenant where you wish to register your application, from the **Favorites** or **All Directories** list.
4. Select **All services** in the left-hand nav, and choose **Azure Active Directory**.
5. In the **Azure Active Directory** pane, select **App registrations (Legacy)** to view a list of all your applications.

If you do not see the application you want show up here, use the various filters at the top of the **App registrations (Legacy)** list to restrict the list or scroll down the list to locate your application.

6. Select the application you want to define app roles in.
7. In the blade for your application, select **Manifest**.
8. Edit the app manifest by locating the `appRoles` setting and adding all your Application Roles.

NOTE

Each app role definition in this manifest must have a different valid GUID for the `id` property.

The `value` property of each app role definition should exactly match the strings that are used in the code in the application. The `value` property can't contain spaces. If it does, you'll receive an error when you save the manifest.

9. Save the manifest.

Examples

The following example shows the `appRoles` that you can assign to `users`.

NOTE

The `id` must be a unique GUID.

```
"appId": "8763f1c4-f988-489c-a51e-158e9ef97d6a",
"appRoles": [
  {
    "allowedMemberTypes": [
      "User"
    ],
    "displayName": "Writer",
    "id": "d1c2ade8-98f8-45fd-aa4a-6d06b947c66f",
    "isEnabled": true,
    "description": "Writers Have the ability to create tasks.",
    "value": "Writer"
  }
],
"availableToOtherTenants": false,
```

NOTE

The `displayName` cannot contain spaces.

You can define app roles to target `users`, `applications`, or both. When available to `applications`, app roles appear as application permissions in the **Required Permissions** blade. The following example shows an app role targeted towards an `Application`.

```
"appId": "8763f1c4-f988-489c-a51e-158e9ef97d6a",
"appRoles": [
  {
    "allowedMemberTypes": [
      "Application"
    ],
    "displayName": "ConsumerApps",
    "id": "47fbb575-859a-4941-89c9-0f7a6c30beac",
    "isEnabled": true,
    "description": "Consumer apps have access to the consumer data.",
    "value": "Consumer"
  }
],
"availableToOtherTenants": false,
```

The number of roles defined affects the limits that the application manifest has. They have been discussed in detail on the [manifest limits](#) page.

Assign users and groups to roles

Once you've added app roles in your application, you can assign users and groups to these roles.

1. In the **Azure Active Directory** pane, select **Enterprise applications** from the **Azure Active Directory** left-hand navigation menu.
2. Select **All applications** to view a list of all your applications.

If you do not see the application you want show up here, use the various filters at the top of the **All**

applications list to restrict the list or scroll down the list to locate your application.

3. Select the application in which you want to assign users or security group to roles.
4. Select the **Users and groups** pane in the application's left-hand navigation menu.
5. At the top of the **Users and groups** list, select the **Add user** button to open the **Add Assignment** pane.
6. Select the **Users and groups** selector from the **Add Assignment** pane.

A list of users and security groups will be shown along with a textbox to search and locate a certain user or group. This screen allows you to select multiple users and groups in one go.

7. Once you are done selecting the users and groups, press the **Select** button on bottom to move to the next part.
8. Choose the **Select Role** selector from the **Add assignment** pane. All the roles declared earlier in the app manifest will show up.
9. Choose a role and press the **Select** button.
10. Press the **Assign** button on the bottom to finish the assignments of users and groups to the app.
11. Confirm that the users and groups you added are showing up in the updated **Users and groups** list.

More information

- [Authorization in a web app using Azure AD application roles & role claims \(Sample\)](#)
- [Using Security Groups and Application Roles in your apps \(Video\)](#)
- [Azure Active Directory, now with Group Claims and Application Roles](#)
- [Azure Active Directory app manifest](#)
- [AAD Access tokens](#)
- [AAD `id_tokens`](#)

Branding guidelines for applications

10/23/2019 • 5 minutes to read • [Edit Online](#)

When developing applications with Azure Active Directory (Azure AD), you'll need to direct your customers when they want to use their work or school account (managed in Azure AD), or their personal account for sign-up and sign-in to your application.

In this article, you will:

- Learn about the two kinds of user accounts managed by Microsoft and how to refer to Azure AD accounts in your application
- Find out what you need to do add the Microsoft logo for use in your app
- Download the official **Sign in** or **Sign in with Microsoft** images to use in your app
- Learn about the branding and navigation do's and don'ts

Personal accounts vs. work or school accounts from Microsoft

Microsoft manages two kinds of user accounts:

- **Personal accounts** (formerly known as Windows Live ID). These accounts represent the relationship between *individual* users and Microsoft, and are used to access consumer devices and services from Microsoft. These accounts are intended for personal use.
- **Work or school accounts.** These accounts are managed by Microsoft on behalf of organizations that use Azure Active Directory. These accounts are used to sign in to Office 365 and other business services from Microsoft.

Microsoft work or school accounts are typically assigned to end users (employees, students, federal employees) by their organizations (company, school, government agency). These accounts master in the cloud (in the Azure AD platform) or sync to Azure AD from an on-premises directory, such as Windows Server Active Directory. Microsoft is the *custodian* of the work or school accounts, but the accounts are owned and controlled by the organization.

Referring to Azure AD accounts in your application

Microsoft doesn't expose end users to the Azure or the Active Directory brand names, and neither should you.

- Once users are signed in, use the organization's name and logo as much as possible. This is better than using generic terms like "your organization."
- When users aren't signed in, refer to their accounts as "Work or school accounts" and use the Microsoft logo to convey that Microsoft manages these accounts. Don't use terms like "enterprise account," "business account," or "corporate account," which create user confusion.

User account pictogram

In an earlier version of these guidelines, we recommended using a "blue badge" pictogram. Based on user and developer feedback, we now recommend the use of the Microsoft logo instead. The Microsoft logo will help users understand that they can reuse the account they use with Office 365 or other Microsoft business services to sign into your app.

Signing up and signing in with Azure AD

Your app may present separate paths for sign-up and sign-in and the following sections provide visual guidance for both scenarios.

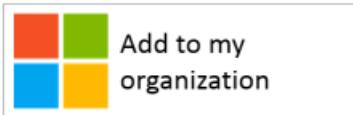
If your app supports end-user sign-up (for example, free to trial or freemium model): You can show a **sign-in** button that allows users to access your app with their work account or their personal account. Azure AD will show a consent prompt the first time they access your app.

If your app requires permissions that only admins can consent to, or if your app requires organizational licensing: Separate admin acquisition from user sign-in. The “**get this app**” button will redirect admins to sign in then ask them to grant consent on behalf of users in their organization, which has the added benefit of suppressing end-user consent prompts to your app.

Visual guidance for app acquisition

Your “get the app” link must redirect the user to the Azure AD grant access (authorize) page, to allow an organization’s administrator to authorize your app to have access to their organization’s data, which is hosted by Microsoft. Details on how to request access are discussed in the [Integrating Applications with Azure Active Directory](#) article.

After admins consent to your app, they can choose to add it to their users’ Office 365 app launcher experience (accessible from the waffle and from <https://portal.office.com/myapps>). If you want to advertise this capability, you can use terms like “Add this app to your organization” and show a button like the following example:



However, we recommend that you write explanatory text instead of relying on buttons. For example:

If you already use Office 365 or other business service from Microsoft, you can grant <your_app_name> access to your organization’s data. This will allow your users to access <your_app_name> with their existing work accounts.

To download the official Microsoft logo for use in your app, right-click the one you want to use and then save it to your computer.

ASSET	PNG FORMAT	SVG FORMAT
Microsoft logo		

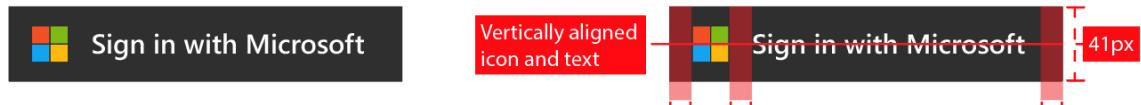
Visual guidance for sign-in

Your app should display a sign-in button that redirects users to the sign-in endpoint that corresponds to the protocol you use to integrate with Azure AD. The following section provides details on what that button should look like.

Pictogram and “Sign in with Microsoft”

It’s the association of the Microsoft logo and the “Sign in with Microsoft” terms that uniquely represent Azure AD amongst other identity providers your app may support. If you don’t have enough space for “Sign in with Microsoft,” it’s ok to shorten it to “Sign in.” You can use a light or dark color scheme for the buttons.

The following diagram shows the Microsoft-recommended redlines when using the assets with your app. The redlines apply to “Sign in with Microsoft” or the shorter “Sign in” version.



Vertically aligned icon and text

Sign in with Microsoft

41px

Font: **Segoe UI Regular 15px**

Font weight: **600**

Font color: **#FFFFFF**

Background: **#2F2F2F**



Vertically aligned icon and text

Sign in with Microsoft

41px

Font: **Segoe UI Regular 15px**

Font weight: **600**

Font color: **#5E5E5E**

Background: **#FFFFFF**

Border: **1px #8C8C8C**

To download the official images for use in your app, right-click the one you want to use and then save it to your computer.

ASSET	PNG FORMAT	SVG FORMAT
Sign in with Microsoft (dark theme)		
Sign in with Microsoft (light theme)		
Sign in (dark theme)		
Sign in (light theme)		

Branding Do's and Don'ts

DO use "work or school account" in combination with the "Sign in with Microsoft" button to provide additional explanation to help end users recognize whether they can use it. **DON'T** use other terms such as "enterprise account", "business account" or "corporate account".

DON'T use "Office 365 ID" or "Azure ID." Office 365 is also the name of a consumer offering from Microsoft, which doesn't use Azure AD for authentication.

DON'T alter the Microsoft logo.

DON'T expose end users to the Azure or Active Directory brands. It's ok however to use these terms with

developers, IT pros, and admins.

Navigation Do's and Don'ts

DO provide a way for users to sign out and switch to another user account. While most people have a single personal account from Microsoft/Facebook/Google/Twitter, people are often associated with more than one organization. Support for multiple signed-in users is coming soon.

How to: Configure terms of service and privacy statement for an app

7/1/2019 • 2 minutes to read • [Edit Online](#)

Developers who build and manage apps that integrate with Azure Active Directory (Azure AD) and Microsoft accounts should include links to the app's terms of service and privacy statement. The terms of service and privacy statement are surfaced to users through the user consent experience. They help your users know that they can trust your app. The terms of service and privacy statement are especially critical for user-facing multi-tenant apps—apps that are used by multiple directories or are available to any Microsoft account.

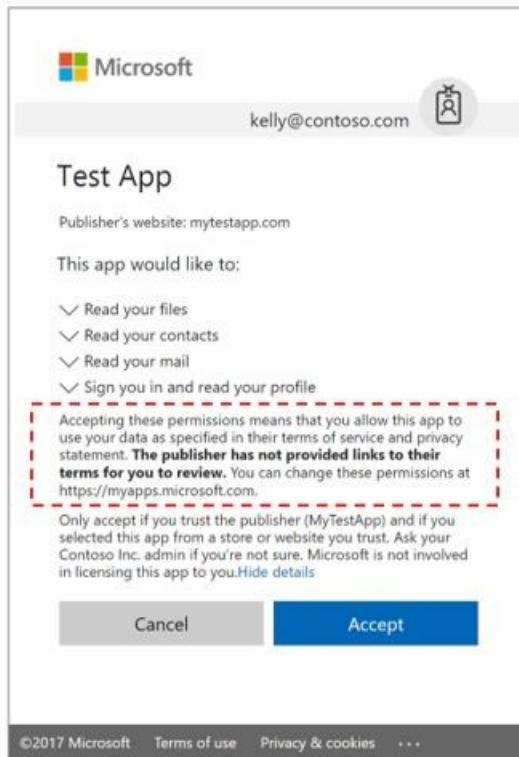
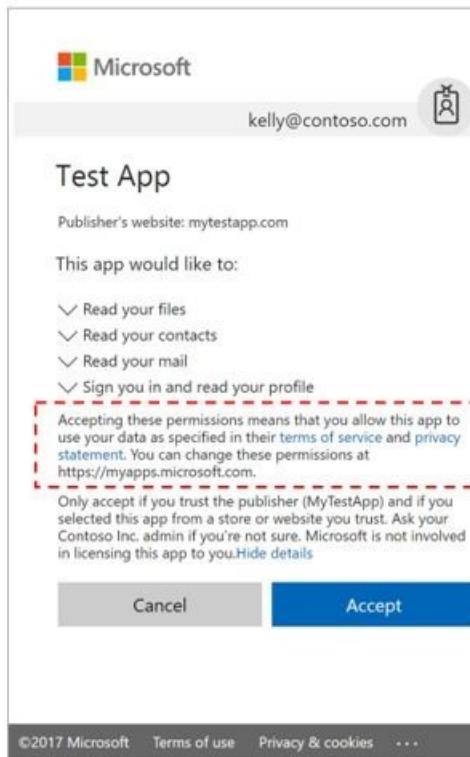
You are responsible for creating the terms of service and privacy statement documents for your app, and for providing the URLs to these documents. For multi-tenant apps that fail to provide these links, the user consent experience for your app will show an alert, which may discourage users from consenting to your app.

NOTE

- Single-tenant apps will not show an alert.
- If one or both of the two links are missing, your app will show an alert.

User consent experience

The following examples show the user consent experience when the terms of service and privacy statement are configured and when these links are not configured.



Privacy statement and terms of service have been provided

Privacy statement and terms of service have not been provided

Formatting links to the terms of service and privacy statement

documents

Before you add links to your app's terms of service and privacy statement documents, make sure the URLs follow these guidelines.

GUIDELINE	DESCRIPTION
Format	Valid URL
Valid schemas	HTTP and HTTPS We recommend HTTPS
Max length	2048 characters

Examples: <https://myapp.com/terms-of-service> and <https://myapp.com/privacy-statement>

Adding links to the terms of service and privacy statement

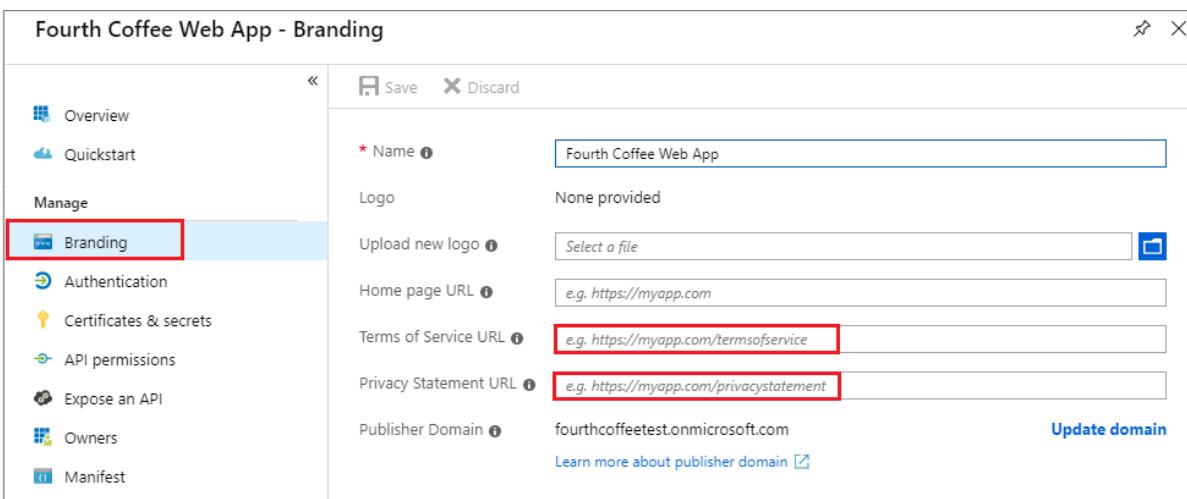
When the terms of service and privacy statement are ready, you can add links to these documents in your app using one of these methods:

- [Through the Azure portal](#)
- [Using the app object JSON](#)
- [Using the MS Graph beta REST API](#)

Using the Azure portal

Follow these steps in the Azure portal.

1. Sign in to the [Azure portal](#).
2. Navigate to the **App Registrations** section and select your app.
3. Open the **Branding** pane.
4. Fill out the **Terms of Service URL** and **Privacy Statement URL** fields.
5. Save your changes.



Using the app object JSON

If you prefer to modify the app object JSON directly, you can use the manifest editor in the Azure portal or Application Registration Portal to include links to your app's terms of service and privacy statement.

```
"informationalUrls": {  
    "termsOfService": "<your_terms_of_service_url>",  
    "privacy": "<your_privacy_statement_url>"  
}
```

Using the MSGraph beta REST API

To programmatically update all your apps, you can use the MSGraph beta REST API to update all your apps to include links to the terms of service and privacy statement documents.

```
PATCH https://graph.microsoft.com/beta/applications/{application id}  
{  
    "appId": "{your application id}",  
    "info": {  
        "termsOfServiceUrl": "<your_terms_of_service_url>",  
        "supportUrl": null,  
        "privacyStatementUrl": "<your_privacy_statement_url>",  
        "marketingUrl": null,  
        "logoUrl": null  
    }  
}
```

NOTE

- Be careful not to overwrite any pre-existing values you have assigned to any of these fields: `supportUrl`, `marketingUrl`, and `logoUrl`
- The MSGraph beta REST API will only work when you sign in with an Azure AD account. Personal Microsoft accounts are not supported.

Adding an Azure Active Directory by using Connected Services in Visual Studio

11/12/2019 • 2 minutes to read • [Edit Online](#)

By using Azure Active Directory (Azure AD), you can support Single Sign-On (SSO) for ASP.NET MVC web applications, or Active Directory Authentication in Web API services. With Azure AD Authentication, your users can use their accounts from Azure Active Directory to connect to your web applications. The advantages of Azure AD Authentication with Web API include enhanced data security when exposing an API from a web application. With Azure AD, you do not have to manage a separate authentication system with its own account and user management.

This article and its companion articles provide details of using the Visual Studio Connected Service feature for Active Directory. The capability is available in Visual Studio 2015 and later.

At present, the Active Directory connected service does not support ASP.NET Core applications.

Prerequisites

- Azure account: if you don't have an Azure account, you can [sign up for a free trial](#) or [activate your Visual Studio subscriber benefits](#).
- **Visual Studio 2015** or later. [Download Visual Studio now](#).

Connect to Azure Active Directory using the Connected Services dialog

1. In Visual Studio, create or open an ASP.NET MVC project, or an ASP.NET Web API project. You can use the MVC, Web API, Single-Page Application, Azure API App, Azure Mobile App, and Azure Mobile Service templates.
2. Select the **Project > Add Connected Service...** menu command, or double-click the **Connected Services** node found under the project in Solution Explorer.
3. On the **Connected Services** page, select **Authentication with Azure Active Directory**.

Connected Services

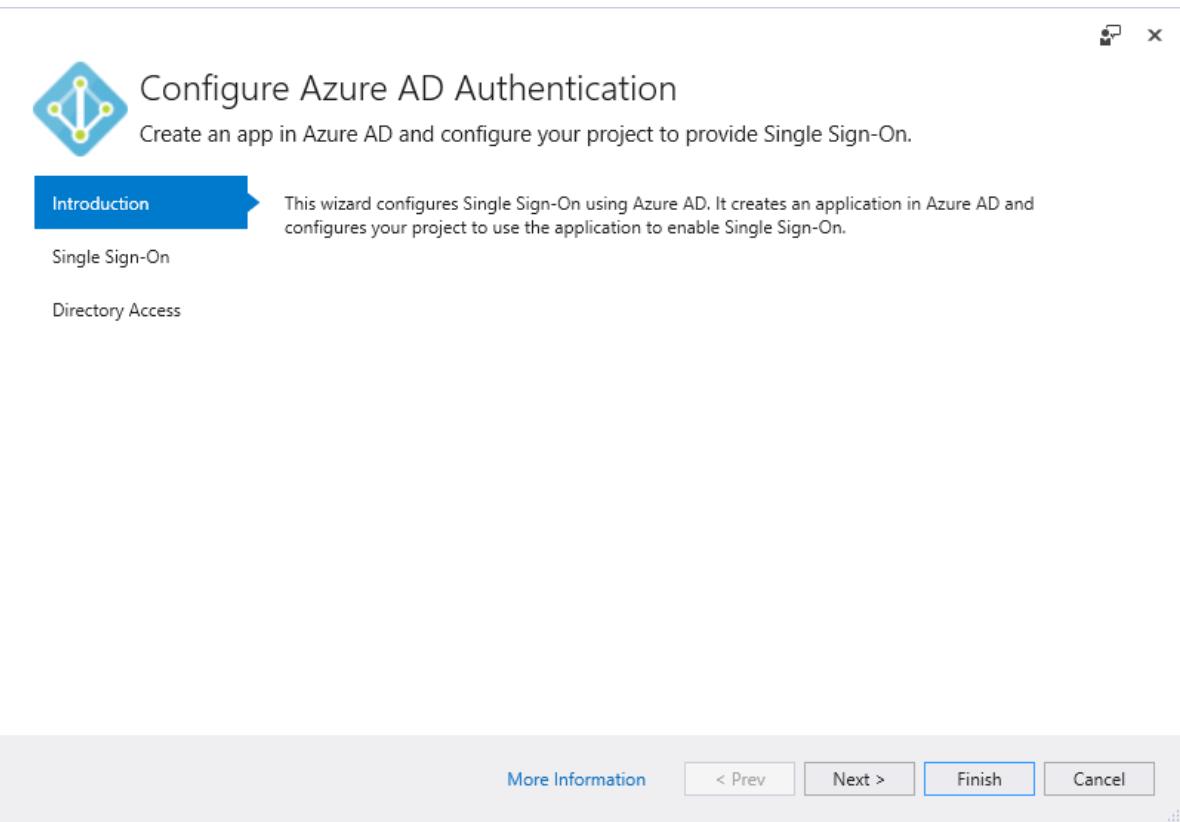
Add code and dependencies for one of these services to your application

Cloud Storage with Azure Storage
Store and access data with Azure Storage services like Blobs, Queues, and Tables.

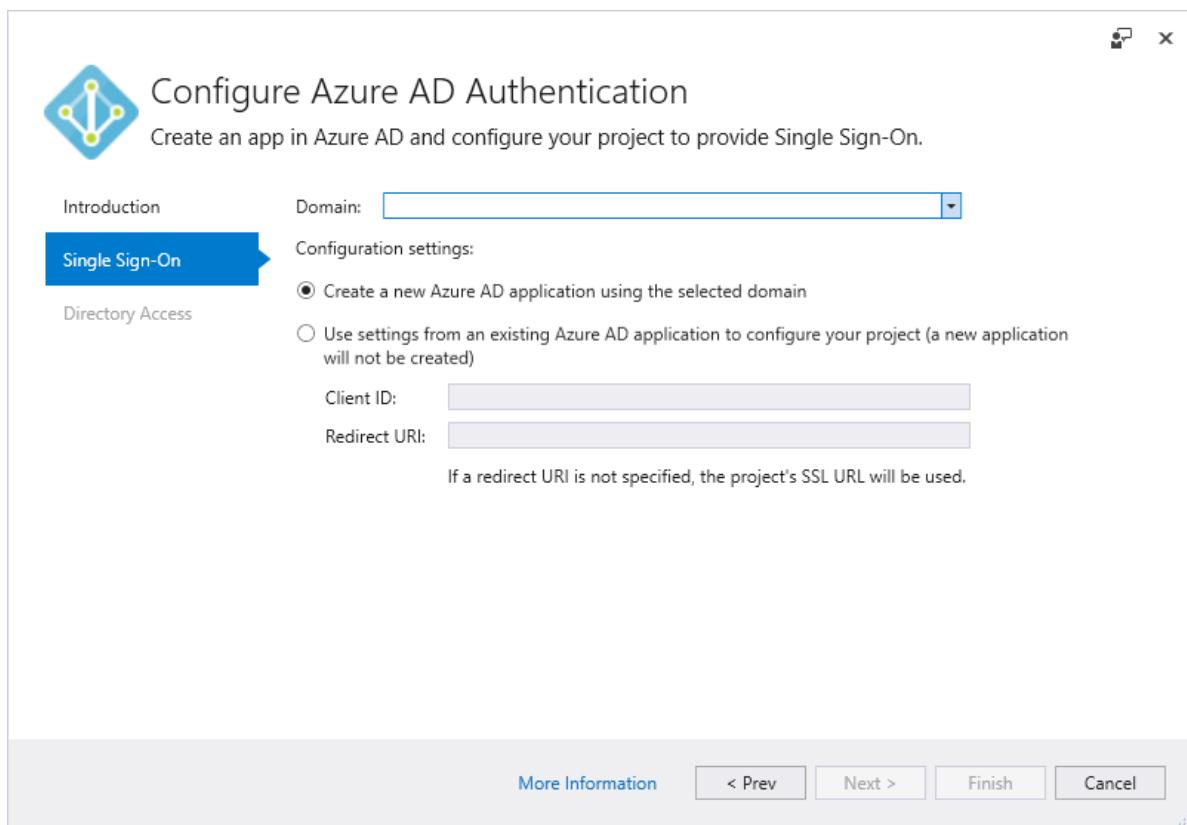
Access Office 365 Services with Microsoft Graph
Use the Microsoft Graph API to integrate your applications with Office 365 services.

Authentication with Azure Active Directory
Configure Single Sign-On in your application using Azure AD.

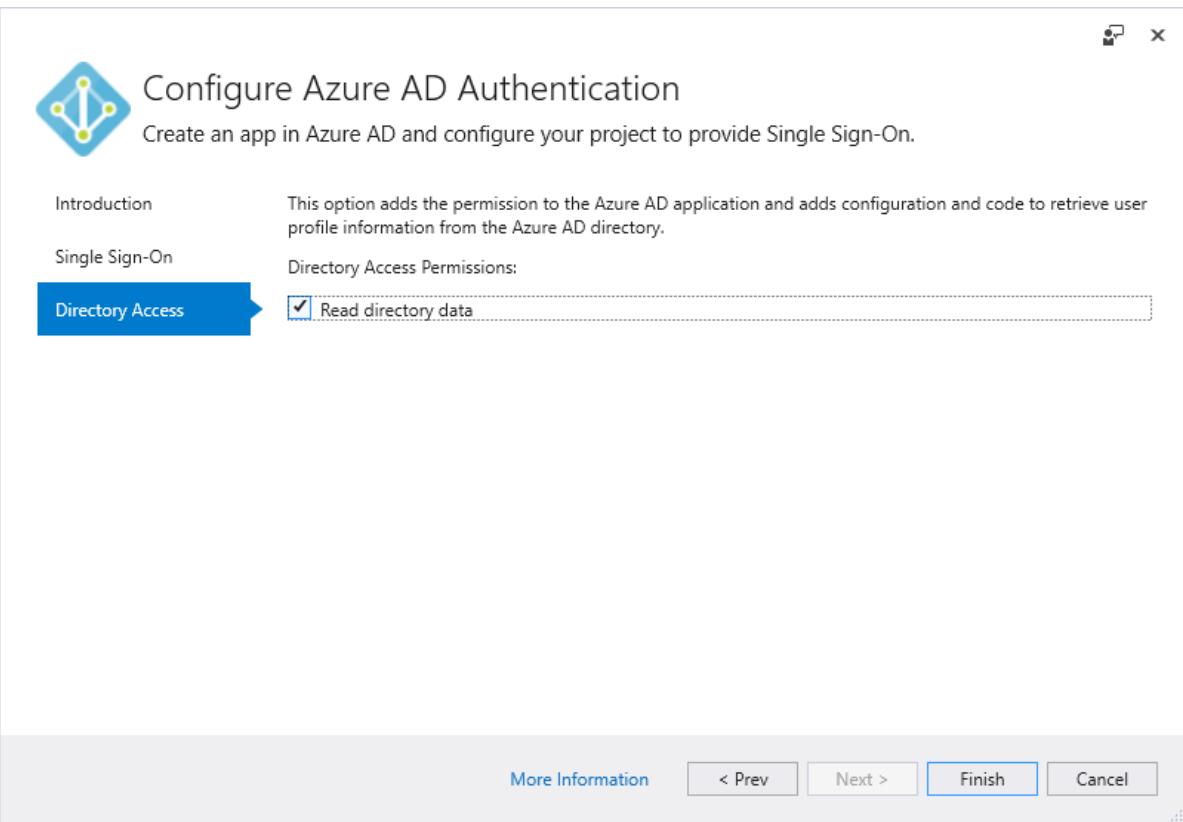
4. On the **Introduction** page, select **Next**. If you see errors on this page, refer to [Diagnosing errors with the Azure Active Directory Connected Service](#).



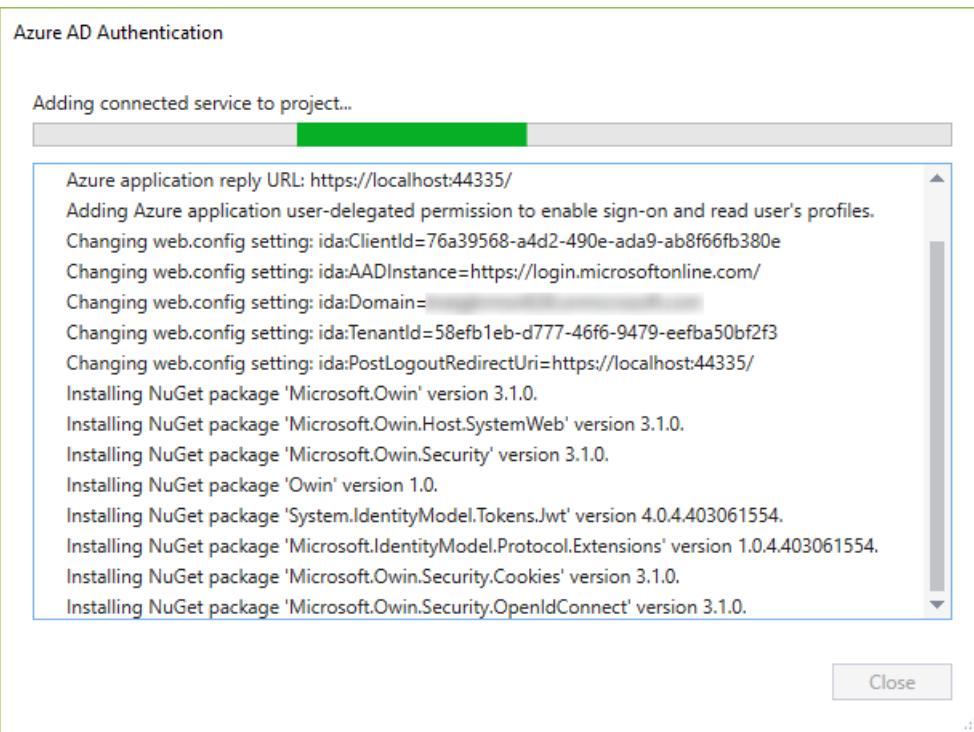
5. On the **Single-Sign On** page, select a domain from the **Domain** drop-down list. The list contains all domains accessible by the accounts listed in the Account Settings dialog of Visual Studio (**File > Account Settings...**). As an alternative, you can enter a domain name if you don't find the one you're looking for, such as `mydomain.onmicrosoft.com`. You can choose the option to create an Azure Active Directory app or use the settings from an existing Azure Active Directory app. Select **Next** when done.



6. On the **Directory Access** page, select the **Read directory data** option as desired. Developers typically include this option.



7. Select **Finish** to start modifications to your project to enable Azure AD authentication. Visual Studio shows progress during this time:



8. When the process is complete, Visual Studio opens your browser to one of the following articles, as appropriate to your project type:

- [Get started with .NET MVC projects](#)
- [Get started with WebAPI projects](#)

9. You can also see the Active Directory domain on the [Azure portal](#).

How your project is modified

When you add the connected service the wizard, Visual Studio adds Azure Active Directory and associated

references to your project. Configuration files and code files in your project are also modified to add support for Azure AD. The specific modifications that Visual Studio makes depend on the project type. See the following articles for details:

- [What happened to my .NET MVC project?](#)
- [What happened to my Web API project?](#)

Next steps

- [Authentication scenarios for Azure Active Directory](#)
- [Add sign-in with Microsoft to an ASP.NET web app](#)

Getting Started with Azure Active Directory (ASP.NET MVC Projects)

10/23/2019 • 2 minutes to read • [Edit Online](#)

This article provides additional guidance after you've added Active Directory to an ASP.NET MVC project through the **Project > Connected Services** command of Visual Studio. If you've not already added the service to your project, you can do so at any time.

See [What happened to my MVC project?](#) for the changes made to your project when adding the connected service.

Requiring authentication to access controllers

All controllers in your project were adorned with the `[Authorize]` attribute. This attribute requires the user to be authenticated before accessing these controllers. To allow the controller to be accessed anonymously, remove this attribute from the controller. If you want to set the permissions at a more granular level, apply the attribute to each method that requires authorization instead of applying it to the controller class.

Adding SignIn / SignOut Controls

To add the SignIn/SignOut controls to your view, you can use the `_LoginPartial.cshtml` partial view to add the functionality to one of your views. Here is an example of the functionality added to the standard `_Layout.cshtml` view. (Note the last element in the div with class `navbar-collapse`):

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")

</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @Html.Partial("_LoginPartial")
            </div>
        </div>
        <div class="container body-content">
            @RenderBody()
            <hr />
            <footer>
                <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
            </footer>
        </div>
    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>

```

Next steps

- [Authentication scenarios for Azure Active Directory](#)
- [Add sign-in with Microsoft to an ASP.NET web app](#)

What happened to my MVC project (Visual Studio Azure Active Directory connected service)?

8/7/2019 • 3 minutes to read • [Edit Online](#)

This article identifies the exact changes made to an ASP.NET MVC project when adding the [Azure Active Directory connected service using Visual Studio](#).

For information on working with the connected service, see [Getting Started](#).

Added references

Affects the project file *.NET references) and `packages.config` (NuGet references).

TYPE	REFERENCE
.NET; NuGet	Microsoft.IdentityModel.Protocol.Extensions
.NET; NuGet	Microsoft.Owin
.NET; NuGet	Microsoft.Owin.Host.SystemWeb
.NET; NuGet	Microsoft.Owin.Security
.NET; NuGet	Microsoft.Owin.Security.Cookies
.NET; NuGet	Microsoft.Owin.Security.OpenIdConnect
.NET; NuGet	Owin
.NET	System.IdentityModel
.NET; NuGet	System.IdentityModel.Tokens.Jwt
.NET	System.Runtime.Serialization

Additional references if you selected the **Read directory data** option:

TYPE	REFERENCE
.NET; NuGet	EntityFramework
.NET	EntityFramework.SqlServer (Visual Studio 2015 only)
.NET; NuGet	Microsoft.Azure.ActiveDirectory.GraphClient
.NET; NuGet	Microsoft.Data.Edm
.NET; NuGet	Microsoft.Data.OData

TYPE	REFERENCE
.NET; NuGet	Microsoft.Data.Services.Client
.NET; NuGet	Microsoft.IdentityModel.Clients.ActiveDirectory
.NET	Microsoft.IdentityModel.Clients.ActiveDirectory.WindowsForms (Visual Studio 2015 only)
.NET; NuGet	System.Spatial

The following references are removed (ASP.NET 4 projects only, as in Visual Studio 2015):

TYPE	REFERENCE
.NET; NuGet	Microsoft.AspNet.Identity.Core
.NET; NuGet	Microsoft.AspNet.Identity.EntityFramework
.NET; NuGet	Microsoft.AspNet.Identity.Owin

Project file changes

- Set the property `IISExpressSSLPort` to a distinct number.
- Set the property `WebProject_DirectoryAccessLevelKey` to 0, or 1 if you selected the **Read directory data** option.
- Set the property `IISUrl` to `https://localhost:<port>/` where `<port>` matches the `IISExpressSSLPort` value.

web.config or app.config changes

- Added the following configuration entries:

```

<appSettings>
    <add key="ida:ClientId" value="<ClientId from the new Azure AD app>" />
    <add key="ida:AADInstance" value="https://login.microsoftonline.com/" />
    <add key="ida:Domain" value="<your selected Azure domain>" />
    <add key="ida:TenantId" value="<the Id of your selected Azure AD tenant>" />
    <add key="ida:PostLogoutRedirectUri" value="<project start page, such as https://localhost:44335>" />
</appSettings>

```

- Added `<dependentAssembly>` elements under the `<runtime><assemblyBinding>` node for `System.IdentityModel.Tokens.Jwt` and `Microsoft.IdentityModel.Protocol.Extensions`.

Additional changes if you selected the **Read directory data** option:

- Added the following configuration entry under `<appSettings>`:

```

<add key="ida:ClientSecret" value="<Azure AD app's new client secret>" />

```

- Added the following elements under `<configuration>`; values for the project-mdf-file and project-catalog-id will vary:

```

<configSections>
  <!-- For more information on Entity Framework configuration, visit https://go.microsoft.com/fwlink/?LinkID=237468 -->
  <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
    EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    requirePermission="false" />
</configSections>

<connectionStrings>
  <add name="DefaultConnection" connectionString="Data Source=
    (localdb)\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\<project-mdf-file>.mdf;Initial Catalog=
    <project-catalog-id>;Integrated Security=True" providerName="System.Data.SqlClient" />
</connectionStrings>

<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
    EntityFramework">
    <parameters>
      <parameter value="mssqllocaldb" />
    </parameters>
  </defaultConnectionFactory>
  <providers>
    <provider invariantName="System.Data.SqlClient"
      type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
  </providers>
</entityFramework>

```

- Added `<dependentAssembly>` elements under the `<runtime><assemblyBinding>` node for `Microsoft.Data.Services.Client`, `Microsoft.Data.Edm`, and `Microsoft.Data.OData`.

Code changes and additions

- Added the `[Authorize]` attribute to `Controllers/HomeController.cs` and any other existing controllers.
- Added an authentication startup class, `App_Start/Startup.Auth.cs`, containing startup logic for Azure AD authentication. If you selected the **Read directory data** option, this file also contains code to receive an OAuth code and exchange it for an access token.
- Added a controller class, `Controllers/AccountController.cs`, containing `SignIn` and `SignOut` methods.
- Added a partial view, `Views/Shared/_LoginPartial.cshtml`, containing an action link for `SignIn` and `SignOut`.
- Added a partial view, `Views/Account/SignoutCallback.cshtml`, containing HTML for sign-out UI.
- Updated the `Startup.Configuration` method to include a call to `ConfigureAuth(app)` if the class already existed; otherwise added a `Startup` class that includes calls the method.
- Added `Connected Services/AzureAD/ConnectedService.json` (Visual Studio 2017) or `Service References/Azure AD/ConnectedService.json` (Visual Studio 2015), containing information that Visual Studio uses to track the addition of the connected service.
- If you selected the **Read directory data** option, added `Models/ADALTokenCache.cs` and `Models/ApplicationDbContext.cs` to support token caching. Also added an additional controller and view to illustrate accessing user profile information using Azure graph APIs: `Controllers/UserProfileController.cs`, `Views/UserProfile/Index.cshtml`, and `Views/UserProfile/Relogin.cshtml`

File backup (Visual Studio 2015)

When adding the connected service, Visual Studio 2015 backs up changed and removed files. All affected files

are saved in the folder `Backup/AzureAD`. Visual Studio 2017 and later does not create backups.

- `Startup.cs`
- `App_Start\IdentityConfig.cs`
- `App_Start\Startup.Auth.cs`
- `Controllers\AccountController.cs`
- `Controllers\ManageController.cs`
- `Models\IdentityModels.cs`
- `Models\ManageViewModels.cs`
- `Views\Shared_LoginPartial.cshtml`

Changes on Azure

- Created an Azure AD Application in the domain that you selected when adding the connected service.
- Updated the app to include the **Read directory data** permission if that option was selected.

[Learn more about Azure Active Directory.](#)

Next steps

- [Authentication scenarios for Azure Active Directory](#)
- [Add sign-in with Microsoft to an ASP.NET web app](#)

Get Started with Azure Active Directory (WebAPI projects)

8/7/2019 • 2 minutes to read • [Edit Online](#)

This article provides additional guidance after you've added Active Directory to an ASP.NET WebAPI project through the **Project > Connected Services** command of Visual Studio. If you've not already added the service to your project, you can do so at any time.

See [What happened to my WebAPI project?](#) for the changes made to your project when adding the connected service.

Requiring authentication to access controllers

All controllers in your project were adorned with the `[Authorize]` attribute. This attribute requires the user to be authenticated before accessing the APIs defined by these controllers. To allow the controller to be accessed anonymously, remove this attribute from the controller. If you want to set the permissions at a more granular level, apply the attribute to each method that requires authorization instead of applying it to the controller class.

Next steps

- [Authentication scenarios for Azure Active Directory](#)
- [Add sign-in with Microsoft to an ASP.NET web app](#)

What happened to my WebAPI project (Visual Studio Azure Active Directory connected service)

8/7/2019 • 2 minutes to read • [Edit Online](#)

This article identifies the exact changes made to ASP.NET WebAPI, ASP.NET Single-Page Application, and ASP.NET Azure API projects when adding the [Azure Active Directory connected service using Visual Studio](#). Also applies to the ASP.NET Azure Mobile Service projects in Visual Studio 2015.

For information on working with the connected service, see [Getting Started](#).

Added references

Affects the project file *.NET references) and `packages.config` (NuGet references).

TYPE	REFERENCE
.NET; NuGet	Microsoft.Owin
.NET; NuGet	Microsoft.Owin.Host.SystemWeb
.NET; NuGet	Microsoft.Owin.Security
.NET; NuGet	Microsoft.Owin.Security.ActiveDirectory
.NET; NuGet	Microsoft.Owin.Security.Jwt
.NET; NuGet	Microsoft.Owin.Security.OAuth
.NET; NuGet	Owin
.NET; NuGet	System.IdentityModel.Tokens.Jwt

Additional references if you selected the **Read directory data** option:

TYPE	REFERENCE
.NET; NuGet	EntityFramework
.NET	EntityFramework.SqlServer (Visual Studio 2015 only)
.NET; NuGet	Microsoft.Azure.ActiveDirectory.GraphClient
.NET; NuGet	Microsoft.Data.Edm
.NET; NuGet	Microsoft.Data.OData
.NET; NuGet	Microsoft.Data.Services.Client
.NET; NuGet	Microsoft.IdentityModel.Clients.ActiveDirectory

TYPE	REFERENCE
.NET	Microsoft.IdentityModel.Clients.ActiveDirectory.WindowsForms (Visual Studio 2015 only)
.NET; NuGet	System.Spatial

The following references are removed (ASP.NET 4 projects only, as in Visual Studio 2015):

TYPE	REFERENCE
.NET; NuGet	Microsoft.AspNet.Identity.Core
.NET; NuGet	Microsoft.AspNet.Identity.EntityFramework
.NET; NuGet	Microsoft.AspNet.Identity.Owin

Project file changes

- Set the property `IISExpressSSLPort` to a distinct number.
- Set the property `WebProject_DirectoryAccessLevelKey` to 0, or 1 if you selected the **Read directory data** option.
- Set the property `IISUrl` to `https://localhost:<port>/` where `<port>` matches the `IISExpressSSLPort` value.

web.config or app.config changes

- Added the following configuration entries:

```
<appSettings>
    <add key="ida:ClientId" value="<ClientId from the new Azure AD app>" />
    <add key="ida:Tenant" value="<your selected Azure domain>" />
    <add key="ida:Audience" value="<your selected domain + / + project name>" />
</appSettings>
```

- Visual Studio 2017 only: Also added the following entry under `<appSettings>`

```
<add key="ida:MetadataAddress" value="<domain URL + /federationmetadata/2007-06/federationmetadata.xml>" />
```

- Added `<dependentAssembly>` elements under the `<runtime><assemblyBinding>` node for `System.IdentityModel.Tokens.Jwt`.
- If you selected the **Read directory data** option, added the following configuration entry under `<appSettings>`:

```
<add key="ida:Password" value="<Your Azure AD app's new password>" />
```

Code changes and additions

- Added the `[Authorize]` attribute to `Controllers/ValueController.cs` and any other existing controllers.

- Added an authentication startup class, `App_Start/Startup.Auth.cs`, containing startup logic for Azure AD authentication, or modified it accordingly. If you selected the **Read directory data** option, this file also contains code to receive an OAuth code and exchange it for an access token.
- (Visual Studio 2015 with ASP.NET 4 app only) Removed `App_Start/IdentityConfig.cs` and added `Controllers/AccountController.cs`, `Models/IdentityModel.cs`, and `Providers/ApplicationAuthProvider.cs`.
- Added `Connected Services/AzureAD/ConnectedService.json` (Visual Studio 2017) or `Service References/Azure AD/ConnectedService.json` (Visual Studio 2015), containing information that Visual Studio uses to track the addition of the connected service.

File backup (Visual Studio 2015)

When adding the connected service, Visual Studio 2015 backs up changed and removed files. All affected files are saved in the folder `Backup/AzureAD`. Visual Studio 2017 does not create backups.

- `Startup.cs`
- `App_Start\IdentityConfig.cs`
- `App_Start\Startup.Auth.cs`
- `Controllers\AccountController.cs`
- `Controllers\ManageController.cs`
- `Models\IdentityModels.cs`
- `Models\ApplicationOAuthProvider.cs`

Changes on Azure

- Created an Azure AD Application in the domain that you selected when adding the connected service.
- Updated the app to include the **Read directory data** permission if that option was selected.

[Learn more about Azure Active Directory.](#)

Next steps

- [Authentication scenarios for Azure Active Directory](#)
- [Add sign-in with Microsoft to an ASP.NET web app](#)

Diagnosing errors with the Azure Active Directory Connected Service

11/12/2019 • 2 minutes to read • [Edit Online](#)

While detecting previous authentication code, the Azure Active Director connect server detected an incompatible authentication type.

To correctly detect previous authentication code in a project, the project must be built. If you see this error and you don't have a previous authentication code in your project, rebuild and try again.

Project types

The connected service checks the type of project you're developing so it can inject the right authentication logic into the project. If there's any controller that derives from `ApiController` in the project, the project is considered a WebAPI project. If there are only controllers that derive from `MVC.Controller` in the project, the project is considered an MVC project. The connected service doesn't support any other project type.

Compatible authentication code

The connected service also checks for authentication settings that have been previously configured or are compatible with the service. If all settings are present, it's considered a re-entrant case, and the connected service opens display the settings. If only some of the settings are present, it's considered an error case.

In an MVC project, the connected service checks for any of the following settings, which result from previous use of the service:

```
<add key="ida:ClientId" value="" />
<add key="ida:Tenant" value="" />
<add key="ida:AADInstance" value="" />
<add key="ida:PostLogoutRedirectUri" value="" />
```

Also, the connected service checks for any of the following settings in a Web API project, which result from previous use of the service:

```
<add key="ida:ClientId" value="" />
<add key="ida:Tenant" value="" />
<add key="ida:Audience" value="" />
```

Incompatible authentication code

Finally, the connected service attempts to detect versions of authentication code that have been configured with previous versions of Visual Studio. If you received this error, it means your project contains an incompatible authentication type. The connected service detects the following types of authentication from previous versions of Visual Studio:

- Windows Authentication
- Individual User Accounts
- Organizational Accounts

To detect Windows Authentication in an MVC project, the connected looks for the `<authentication>` element in your `web.config` file.

```
<configuration>
  <system.web>
    <authentication mode="Windows" />
  </system.web>
</configuration>
```

To detect Windows Authentication in a Web API project, the connected service looks for the `IISExpressWindowsAuthentication` element in your project's `.csproj` file:

```
<Project>
  <PropertyGroup>
    <IISExpressWindowsAuthentication>enabled</IISExpressWindowsAuthentication>
  </PropertyGroup>
</Project>
```

To detect Individual User Accounts authentication, the connected service looks for the package element in your `packages.config` file.

```
<packages>
  <package id="Microsoft.AspNet.Identity.EntityFramework" version="2.1.0" targetFramework="net45" />
</packages>
```

To detect an old form of Organizational Account authentication, the connected service looks for the following element in `web.config`:

```
<configuration>
  <appSettings>
    <add key="ida:Realm" value="***" />
  </appSettings>
</configuration>
```

To change the authentication type, remove the incompatible authentication type and try adding the connected service again.

For more information, see [Authentication Scenarios for Azure AD](#).

How to get AppSource Certified for Azure Active Directory

10/15/2019 • 3 minutes to read • [Edit Online](#)

Microsoft AppSource is a destination for business users to discover, try, and manage line-of-business SaaS applications (standalone SaaS and add-on to existing Microsoft SaaS products).

To list a standalone SaaS application on AppSource, your application must accept single sign-on from work accounts from any company or organization that has Azure Active Directory (Azure AD). The sign-in process must use the [OpenID Connect](#) or [OAuth 2.0](#) protocols. SAML integration is not accepted for AppSource certification.

Guides and code samples

If you want to learn about how to integrate your application with Azure AD using Open ID connect, follow our guides and code samples in the [Azure Active Directory developer's guide](#).

Multi-tenant applications

A *multi-tenant application* is an application that accepts sign-ins from users from any company or organization that have Azure AD without requiring a separate instance, configuration, or deployment. AppSource recommends that applications implement multi-tenancy to enable the *single-click* free trial experience.

To enable multi-tenancy on your application, follow these steps:

1. Set `Multi-Tenanted` property to `Yes` on your application registration's information in the [Azure portal](#). By default, applications created in the Azure portal are configured as *single-tenant*.
2. Update your code to send requests to the `common` endpoint. To do this, update the endpoint from `https://login.microsoftonline.com/{yourtenant}` to `https://login.microsoftonline.com/common*`.
3. For some platforms, like ASP .NET, you need also to update your code to accept multiple issuers.

For more information about multi-tenancy, see [How to sign in any Azure Active Directory \(Azure AD\) user using the multi-tenant application pattern](#).

Single-tenant applications

A *single-tenant application* is an application that only accepts sign-ins from users of a defined Azure AD instance. External users (including work or school accounts from other organizations, or personal accounts) can sign in to a single-tenant application after adding each user as a guest account to the Azure AD instance that the application is registered.

You can add users as guest accounts to Azure AD through the [Azure AD B2B collaboration](#) and you can do this [programmatically](#). When using B2B, users can create a self-service portal that does not require an invitation to sign in. For more info, see [Self-service portal for Azure AD B2B collaboration sign-up](#).

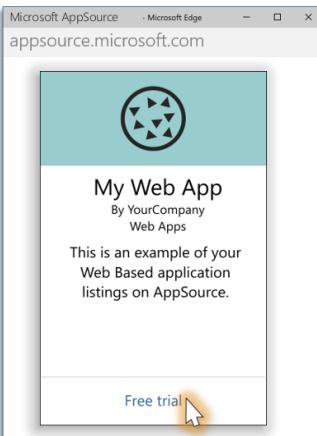
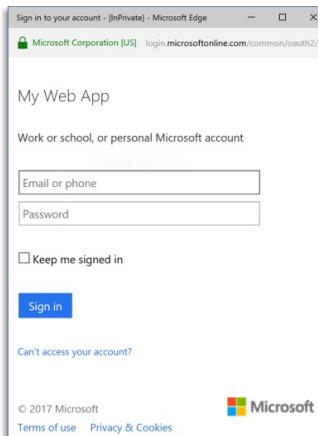
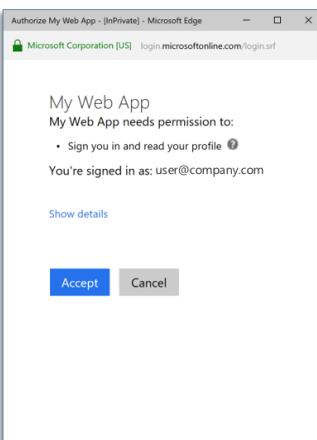
Single-tenant applications can enable the *Contact Me* experience, but if you want to enable the single-click/free trial experience that AppSource recommends, enable multi-tenancy on your application instead.

AppSource trial experiences

Free trial (customer-led trial experience)

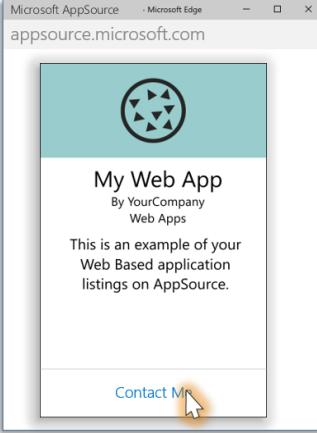
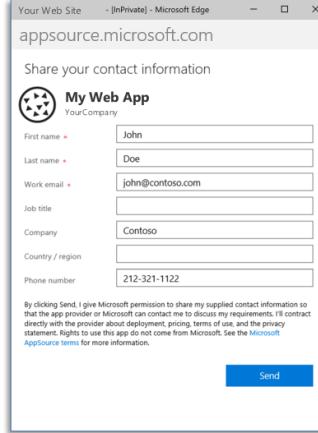
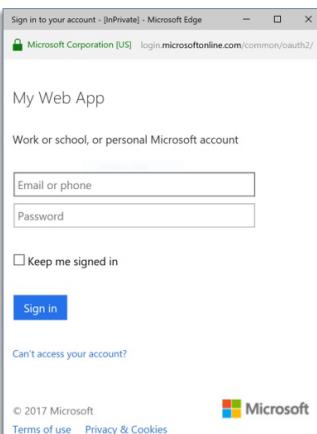
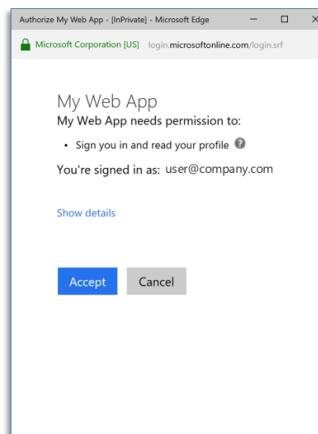
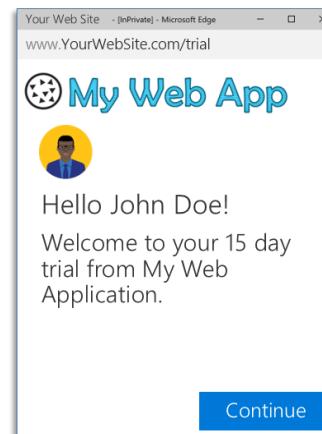
The customer-led trial is the experience that AppSource recommends as it offers a single-click access to your

application. The following example shows what this experience looks like:

<p>1.</p>  <p>Microsoft AppSource - Microsoft Edge appsource.microsoft.com</p> <p>My Web App By YourCompany Web Apps</p> <p>This is an example of your Web Based application listings on AppSource.</p> <p>Free trial</p>	<p>2.</p>  <p>Your Web Site - [InPrivate] - Microsoft Edge www.YourWebSite.com/signin</p> <p>My Web App</p>	<p>3.</p>  <p>Sign in to your account - [InPrivate] - Microsoft Edge Microsoft Corporation [US] login.microsoftonline.com/common/oauth2/</p> <p>My Web App</p> <p>Work or school, or personal Microsoft account</p> <p>Email or phone Password</p> <p><input type="checkbox"/> Keep me signed in</p> <p>Sign in</p> <p>Can't access your account?</p> <p>© 2017 Microsoft Terms of use Privacy & Cookies</p> <p>Microsoft</p>
<ul style="list-style-type: none">User finds your application in AppSource Web SiteSelects 'Free trial' option	<ul style="list-style-type: none">AppSource redirects user to a URL in your web siteYour web site starts the <i>single-sign-on</i> process automatically (on page load)	<ul style="list-style-type: none">User is redirected to Microsoft Sign-in pageUser provides credentials to sign in
<p>4.</p>  <p>Authorize My Web App - [InPrivate] - Microsoft Edge Microsoft Corporation [US] login.microsoftonline.com/login.srf</p> <p>My Web App My Web App needs permission to: • Sign in and read your profile ⓘ You're signed in as: user@company.com</p> <p>Show details</p> <p>Accept Cancel</p>	<p>5.</p>  <p>Your Web Site - [InPrivate] - Microsoft Edge www.YourWebSite.com/trial</p> <p>My Web App</p> <p>Hello John Doe! Welcome to your 15 day trial from My Web Application.</p> <p>Continue</p>	<ul style="list-style-type: none">User gives consent for your application
		<ul style="list-style-type: none">Sign-in completes and user is redirected back to your web siteUser starts the free trial

Contact me (partner-led trial experience)

You can use the partner trial experience when a manual or a long-term operation needs to happen to provision the user/company--for example, your application needs to provision virtual machines, database instances, or operations that take much time to complete. In this case, after the user selects the **Request Trial** button and fills out a form, AppSource sends you the user's contact information. When you receive this information, you then provision the environment and send the instructions to the user on how to access the trial experience:

<p>1.</p> 	<p>2.</p> 	<p>3.</p> <table border="1" data-bbox="1028 181 1429 586"> <tbody> <tr> <td></td><td>You receive user information</td></tr> <tr> <td></td><td>Setup environment</td></tr> <tr> <td></td><td>Contact user with trial info</td></tr> </tbody> </table>		You receive user information		Setup environment		Contact user with trial info
	You receive user information							
	Setup environment							
	Contact user with trial info							
<ul style="list-style-type: none"> User finds your application in AppSource web site Selects 'Contact Me' option 	<ul style="list-style-type: none"> Fills out a form with contact information 	<ul style="list-style-type: none"> You receive user's information and setup trial instance You send the hyperlink to access your application to the user 						
<p>4.</p> 	<p>5.</p> 	<p>6.</p> 						
<ul style="list-style-type: none"> User accesses your application and complete the single-sign-on process 	<ul style="list-style-type: none"> User gives consent for your application 	<ul style="list-style-type: none"> Sign-in completes and user is redirected back to your web site User starts the free trial 						

More information

For more information about the AppSource trial experience, see [this video](#).

Next Steps

- For more information on building applications that support Azure AD sign-ins, see [Authentication scenarios for Azure AD](#).
- For information on how to list your SaaS application in AppSource, go see [AppSource Partner Information](#)

Get support

For Azure AD integration, we use [Stack Overflow](#) with the community to provide support.

We highly recommend you ask your questions on Stack Overflow first and browse existing issues to see if someone has asked your question before. Make sure that your questions or comments are tagged with

[\[azure-active-directory\]](#) and [\[appsource\]](#).

Use the following comments section to provide feedback and help us refine and shape our content.

List your application in the Azure Active Directory application gallery

10/15/2019 • 4 minutes to read • [Edit Online](#)

This article shows how to list an application in the Azure Active Directory (Azure AD) application gallery, implement single sign-on (SSO), and manage the listing.

What is the Azure AD application gallery?

- Customers find the best possible single sign-on experience.
- Configuration of the application is simple and minimal.
- A quick search finds your application in the gallery.
- Free, Basic, and Premium Azure AD customers can all use this integration.
- Mutual customers get a step-by-step configuration tutorial.
- Customers who use the System for Cross-domain Identity Management ([SCIM](#)) can use provisioning for the same app.

Prerequisites

- For federated applications (Open ID and SAML/WS-Fed), the application must support the software-as-a-service (SaaS) model for getting listed in the Azure AD app gallery. The enterprise gallery applications must support multiple customer configurations and not any specific customer.
- For Open ID Connect, the application must be multitenanted and the [Azure AD consent framework](#) must be properly implemented for the application. The user can send the sign-in request to a common endpoint so that any customer can provide consent to the application. You can control user access based on the tenant ID and the user's UPN received in the token.
- For SAML 2.0/WS-Fed, your application must have the capability to do the SAML/WS-Fed SSO integration in SP or IDP mode. Make sure this capability is working correctly before you submit the request.
- For password SSO, make sure that your application supports form authentication so that password vaulting can be done to get single sign-on to work as expected.
- You need a permanent account for testing with at least two users registered.

Submit the request in the portal

After you've tested that your application integration works with Azure AD, submit your request for access in the [Application Network portal](#). If you have an Office 365 account, use that to sign in to this portal. If not, use your Microsoft account, such as Outlook or Hotmail, to sign in.

If the following page appears after you sign in, contact the [Azure AD SSO Integration Team](#). Provide the email account that you want to use for submitting the request. The Azure AD team will add the account in the Microsoft Application Network portal.

That didn't work

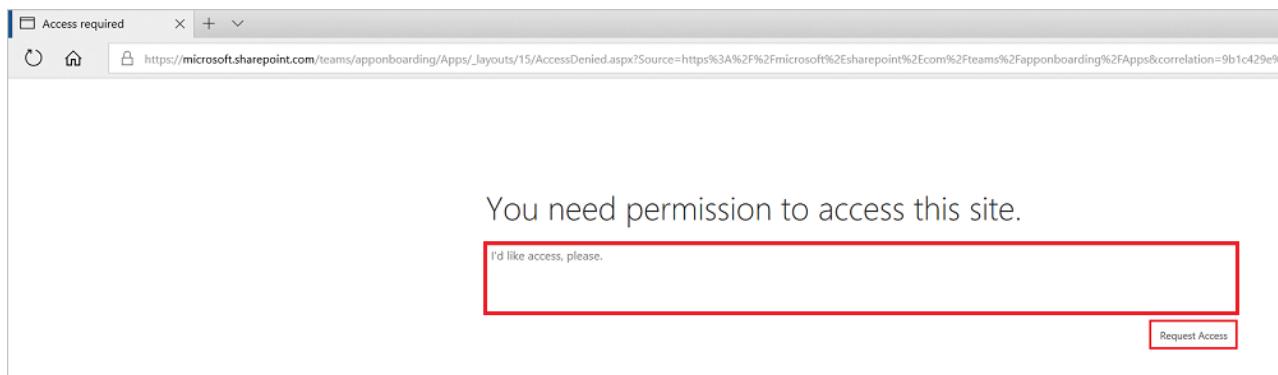
We're sorry, but [REDACTED] can't be found in the microsoft.sharepoint.com directory. Please try again later, while we try to automatically fix this for you.

Here are a few ideas:

- ⊕ [Click here to sign in with a different account to this site.](#)
This will sign you out of all other Office 365 services that you're signed into at this time.
- ⊕ [If you're using this account on another site and don't want to sign out, start your browser in Private Browsing mode for this site \(show me how\).](#)

After the account is added, you can sign in to the Microsoft Application Network portal.

If the following page appears after you sign in, provide a business justification for needing access in the text box. Then select **Request Access**.



Our team reviews the details and gives you access accordingly. After your request is approved, you can sign in to the portal and submit the request by selecting the **Submit Request (ISV)** tile on the home page.

Welcome to Microsoft Application Network

Azure AD Application Gallery FAQs

We recommend partners to go through the FAQs if you need any assistance while submitting your request on Microsoft Application Network portal. Please click on FAQs tile to see more details.

What is Azure AD App Gallery?

Azure AD is a cloud based Identity service. Azure AD app gallery is a common store where all the application connectors are published for single sign-on and user provisioning. Our mutual customers who are using Azure AD as Identity provider look for different SaaS application connectors which are published here. IT administrator adds connector from the app gallery and configure and use it for Single sign-on and provisioning. Azure AD supports all major federation protocols like SAML 2.0, OpenID Connect, OAuth and WS-Fed for single sign-on.

Note: For provisioning, Azure AD support Graph APIs and SCIM 1.0, 1.1 and 2.0 protocol.

Benefits of listing my application

Provide best possible single sign-on experience to our customers

Pre-requisites

To list an application in the Azure AD gallery, the application first needs to implement one of the federation protocol supported by Azure AD. The supported protocols are SAML 2.0, OpenID Connect, OAuth 2.0 and WS-Fed.

Related Links

- Application access and single sign-on with Azure AD
- Integrated application benefits
- List your application in Azure AD
- SAML Protocol reference
- Developing multi-tenant apps with Azure AD
- Overview of Consent Framework
- Azure AD app gallery terms and conditions
- Notices and procedure for making claims of copyright infringement

NOTE

If you have any issues with access, contact the [Azure AD SSO Integration Team](#).

Implement SSO by using the federation protocol

To list an application in the Azure AD app gallery, you first need to implement one of the following federation protocols supported by Azure AD. You also need to agree to the Azure AD application gallery terms and conditions. Read the terms and conditions of the Azure AD application gallery on [this website](#).

- **OpenID Connect:** To integrate your application with Azure AD by using the Open ID Connect protocol,

follow the [developers' instructions](#).

The screenshot shows the 'Application Registration Form' page. On the left, there is a sidebar with four buttons: 'Submit Request (ISV)', 'Track Request(s)', 'Announcement', and 'FAQs'. Below these is another button for 'App requests by Customers'. The main content area has a note: 'Please select the appropriate option(s) for your listing in Azure AD gallery. You can find the instructions [here](#) for all the listing options.' It also includes a note: 'Note-> This listing is only for SaaS applications. We can list the applications in the gallery only if the product is made for multiple customers and not for specific customer.' A question asks 'What type of request do you want to submit?' with two options: 'List my application in the gallery' (selected) and 'Update my application's listing in the gallery'. Below this is a note: 'New/Update gallery app listing with Federated/Password Single Sign On and/or user provisioning'. Another question asks 'What feature would you like to enable when listing your application in the gallery?' with three options: 'Federated SSO' (selected), 'Password SSO', and 'User Provisioning'. A note below asks 'Please select your Application Federation Protocol' with two options: 'SAML 2.0/WS-Fed' (selected) and 'Open ID Connect & OAuth 2.0'.

- If you want to add your application to list in the gallery by using OpenID Connect, select **OpenID Connect & OAuth 2.0** as shown.
- If you have any issues with access, contact the [Azure AD SSO Integration Team](#).
- **SAML 2.0 or WS-Fed:** If your app supports SAML 2.0, you can integrate it directly with an Azure AD tenant by following the [instructions to add a custom application](#).

The screenshot shows the 'Application Registration Form' page. On the left, there is a sidebar with four buttons: 'Submit Request (ISV)', 'Track Request(s)', 'Announcement', and 'FAQs'. Below these is another button for 'App requests by Customers'. The main content area has a note: 'Please select the appropriate option(s) for your listing in Azure AD gallery. You can find the instructions [here](#) for all the listing options.' It also includes a note: 'Note-> This listing is only for SaaS applications. We can list the applications in the gallery only if the product is made for multiple customers and not for specific customer.' A question asks 'What type of request do you want to submit?' with two options: 'List my application in the gallery' (selected) and 'Update my application's listing in the gallery'. Below this is a note: 'New/Update gallery app listing with Federated/Password Single Sign On and/or user provisioning'. Another question asks 'What feature would you like to enable when listing your application in the gallery?' with three options: 'Federated SSO' (selected), 'Password SSO', and 'User Provisioning'. A note below asks 'Please select your Application Federation Protocol' with two options: 'SAML 2.0/WS-Fed' (selected) and 'Open ID Connect & OAuth 2.0'.

- If you want to add your application to list in the gallery by using **SAML 2.0 or WS-Fed**, select **SAML 2.0/WS-Fed** as shown.
- If you have any issues with access, contact the [Azure AD SSO Integration Team](#).

Implement SSO by using the password SSO

Create a web application that has an HTML sign-in page to configure [password-based single sign-on](#). Password-based SSO, also referred to as password vaulting, enables you to manage user access and passwords to web applications that don't support identity federation. It's also useful for scenarios in which several users need to share a single account, such as to your organization's social media app accounts.

[Home](#)

The screenshot shows the 'Application Registration Form' page. On the left, there is a sidebar with four buttons: 'Submit Request (ISV)', 'Track Request(s)', 'Announcement', and 'App requests by Customers'. The 'Submit Request (ISV)' button is highlighted with a blue background. The main content area has a heading 'Application Registration Form'. Below it, a note says: 'Please select the appropriate option(s) for your listing in Azure AD gallery. You can find the instructions [here](#) for all the listing options.' A note below that says: 'Note-> This listing is only for SaaS applications. We can list the applications in the gallery only if the product is made for multiple customers and not for specific customer.' A question 'What type of request do you want to submit?' has two options: 'List my application in the gallery' (selected) and 'Update my application's listing in the gallery'. Below this, a note says: 'New/Update gallery app listing with Federated/Password Single Sign On and/or user provisioning'. Another question 'What feature would you like to enable when listing your application in the gallery?' has three options: 'Federated SSO', 'Password SSO' (selected), and 'User Provisioning'.

- If you want to add your application to list in the gallery by using password SSO, select **Password SSO** as shown.
- If you have any issues with access, contact the [Azure AD SSO Integration Team](#).

Request for user provisioning

Follow the process shown in the following image to request user provisioning.

[Home](#)

The screenshot shows the 'Application Registration Form' page. The sidebar and notes are identical to the previous screenshot. The 'What type of request do you want to submit?' section has 'List my application in the gallery' selected. The 'What feature would you like to enable when listing your application in the gallery?' section has 'User Provisioning' selected (highlighted with a red border).

Update or remove an existing listing

To update or remove an existing application in the Azure AD app gallery, you first need to submit the request in the [Application Network portal](#). If you have an Office 365 account, use that to sign in to this portal. If not, use your Microsoft account, such as Outlook or Hotmail, to sign in.

- Select the appropriate option as shown in the following image.

- To update an existing application, select the appropriate option as per your requirement.
- To remove an existing application from the Azure AD app gallery, select **Remove my application listing from the gallery**.
- If you have any issues with access, contact the [Azure AD SSO Integration Team](#).

List requests by customers

Customers can submit a request to list an application by selecting **App requests by Customers > Submit new request**.

Here's the flow of customer-requested applications.

CUSTOMER	Azure AD Team	ISV Partner	Azure AD Team
Submit the request to list the application in gallery	Notify the ISV Partner about customer request	Verify the request and submit listing request with all necessary details	Approve the request and start working on it
STEP-1	STEP-2	STEP-3	STEP-4

Timelines

The timeline for the process of listing a SAML 2.0 or WS-Fed application in the gallery is 7 to 10 business days.

ISV Partner	Azure AD Team	ISV Partner	Azure AD Team	Azure AD Team	Azure AD Team	Azure AD Team
Submit request listing	Approve the request	Provide sandbox environment for testing	Test integration and share result	Prepare tutorial for integration	Publish Tutorial	Publish Application in the gallery
1 Business Day	1 Business Day	1 Business Day	3 Business Days	2 Business Days	2 Business Days	2 Business Days

The timeline for the process of listing an OpenID Connect application in the gallery is 2 to 5 business days.

ISV Partner	Azure AD Team	ISV Partner	Azure AD Team	Azure AD Team
Submit request listing	Approve the request	Provide sandbox environment for testing	Test integration and share result	Publish Application in the gallery
1 Business Day	1 Business Day	1 Business Day	2 Business Days	2 Business Days

Escalations

For any escalations, send email to the [Azure AD SSO Integration Team](#) at SaaSApplicationIntegrations@service.microsoft.com, and we'll respond as soon as possible.

How to: Migrate from the Azure Access Control Service

8/6/2019 • 20 minutes to read • [Edit Online](#)

Microsoft Azure Access Control Service (ACS), a service of Azure Active Directory (Azure AD), will be retired on November 7, 2018. Applications and services that currently use Access Control must be fully migrated to a different authentication mechanism by then. This article describes recommendations for current customers, as you plan to deprecate your use of Access Control. If you don't currently use Access Control, you don't need to take any action.

Overview

Access Control is a cloud authentication service that offers an easy way to authenticate and authorize users for access to your web applications and services. It allows many features of authentication and authorization to be factored out of your code. Access Control is primarily used by developers and architects of Microsoft .NET clients, ASP.NET web applications, and Windows Communication Foundation (WCF) web services.

Use cases for Access Control can be broken down into three main categories:

- Authenticating to certain Microsoft cloud services, including Azure Service Bus and Dynamics CRM. Client applications obtain tokens from Access Control to authenticate to these services to perform various actions.
- Adding authentication to web applications, both custom and prepackaged (like SharePoint). By using Access Control "passive" authentication, web applications can support sign-in with a Microsoft account (formerly Live ID), and with accounts from Google, Facebook, Yahoo, Azure AD, and Active Directory Federation Services (AD FS).
- Securing custom web services with tokens issued by Access Control. By using "active" authentication, web services can ensure that they allow access only to known clients that have authenticated with Access Control.

Each of these use cases and their recommended migration strategies are discussed in the following sections.

WARNING

In most cases, significant code changes are required to migrate existing apps and services to newer technologies. We recommend that you immediately begin planning and executing your migration to avoid any potential outages or downtime.

Access Control has the following components:

- A secure token service (STS), which receives authentication requests and issues security tokens in return.
- The Azure classic portal, where you create, delete, and enable and disable Access Control namespaces.
- A separate Access Control management portal, where you customize and configure Access Control namespaces.
- A management service, which you can use to automate the functions of the portals.
- A token transformation rule engine, which you can use to define complex logic to manipulate the output format of tokens that Access Control issues.

To use these components, you must create one or more Access Control namespaces. A *namespace* is a dedicated instance of Access Control that your applications and services communicate with. A namespace is isolated from all other Access Control customers. Other Access Control customers use their own namespaces. A namespace in Access Control has a dedicated URL that looks like this:

```
https://<mynamespace>.accesscontrol.windows.net
```

All communication with the STS and management operations are done at this URL. You use different paths for different purposes. To determine whether your applications or services use Access Control, monitor for any traffic to https://<namespace>.accesscontrol.windows.net. Any traffic to this URL is handled by Access Control, and needs to be discontinued.

The exception to this is any traffic to https://accounts.accesscontrol.windows.net. Traffic to this URL is already handled by a different service and **is not** affected by the Access Control deprecation.

For more information about Access Control, see [Access Control Service 2.0 \(archived\)](#).

Find out which of your apps will be impacted

Follow the steps in this section to find out which of your apps will be impacted by ACS retirement.

Download and install ACS PowerShell

1. Go to the PowerShell Gallery and download [Acs.Namespaces](#).
2. Install the module by running

```
Install-Module -Name Acs.Namespaces
```

3. Get a list of all possible commands by running

```
Get-Command -Module Acs.Namespaces
```

To get help on a specific command, run:

```
Get-Help [Command-Name] -Full
```

where [Command-Name] is the name of the ACS command.

List your ACS namespaces

1. Connect to ACS using the **Connect-AcsAccount** cmdlet.

You may need to run `Set-ExecutionPolicy -ExecutionPolicy Bypass` before you can execute commands and be the admin of those subscriptions in order to execute the commands.

2. List your available Azure subscriptions using the **Get-AcsSubscription** cmdlet.
3. List your ACS namespaces using the **Get-AcsNamespace** cmdlet.

Check which applications will be impacted

1. Use the namespace from the previous step and go to https://<namespace>.accesscontrol.windows.net

For example, if one of the namespaces is contoso-test, go to

```
https://contoso-test.accesscontrol.windows.net
```

2. Under **Trust relationships**, select **Relying party applications** to see the list of apps that will be impacted by ACS retirement.
3. Repeat steps 1-2 for any other ACS namespace(s) that you have.

Retirement schedule

As of November 2017, all Access Control components are fully supported and operational. The only restriction is that you [can't create new Access Control namespaces via the Azure classic portal](#).

Here's the schedule for deprecating Access Control components:

- **November 2017:** The Azure AD admin experience in the Azure classic portal [is retired](#). At this point, namespace management for Access Control is available at a new, dedicated URL:
`https://manage.windowsazure.com?restoreClassic=true`. Use this URL to view your existing namespaces, enable and disable namespaces, and to delete namespaces, if you choose to.
- **April 2, 2018:** The Azure classic portal is completely retired, meaning Access Control namespace management is no longer available via any URL. At this point, you can't disable or enable, delete, or enumerate your Access Control namespaces. However, the Access Control management portal will be fully functional and located at
`https://<namespace>.accesscontrol.windows.net`. All other components of Access Control continue to operate normally.
- **November 7, 2018:** All Access Control components are permanently shut down. This includes the Access Control management portal, the management service, STS, and the token transformation rule engine. At this point, any requests sent to Access Control (located at <namespace>.accesscontrol.windows.net) fail. You should have migrated all existing apps and services to other technologies well before this time.

NOTE

A policy disables namespaces that have not requested a token for a period of time. As of early September 2018, this period of time is currently at 14 days of inactivity, but this will be shortened to 7 days of inactivity in the coming weeks. If you have Access Control namespaces that are currently disabled, you can [download and install ACS PowerShell](#) to re-enable the namespace(s).

Migration strategies

The following sections describe high-level recommendations for migrating from Access Control to other Microsoft technologies.

Clients of Microsoft cloud services

Each Microsoft cloud service that accepts tokens that are issued by Access Control now supports at least one alternate form of authentication. The correct authentication mechanism varies for each service. We recommend that you refer to the specific documentation for each service for official guidance. For convenience, each set of documentation is provided here:

SERVICE	GUIDANCE
Azure Service Bus	Migrate to shared access signatures
Azure Service Bus Relay	Migrate to shared access signatures
Azure Managed Cache	Migrate to Azure Cache for Redis
Azure DataMarket	Migrate to the Cognitive Services APIs
BizTalk Services	Migrate to the Logic Apps feature of Azure App Service
Azure Media Services	Migrate to Azure AD authentication

SERVICE	GUIDANCE
Azure Backup	Upgrade the Azure Backup agent

SharePoint customers

SharePoint 2013, 2016, and SharePoint Online customers have long used ACS for authentication purposes in cloud, on premises, and hybrid scenarios. Some SharePoint features and use cases will be affected by ACS retirement, while others will not. The below table summarizes migration guidance for some of the most popular SharePoint feature that leverage ACS:

FEATURE	GUIDANCE
Authenticating users from Azure AD	Previously, Azure AD did not support SAML 1.1 tokens required by SharePoint for authentication, and ACS was used as an intermediary that made SharePoint compatible with Azure AD token formats. Now, you can connect SharePoint directly to Azure AD using Azure AD App Gallery SharePoint on premises app .
App authentication & server-to-server authentication in SharePoint on premises	Not affected by ACS retirement; no changes necessary.
Low trust authorization for SharePoint add-ins (provider hosted and SharePoint hosted)	Not affected by ACS retirement; no changes necessary.
SharePoint cloud hybrid search	Not affected by ACS retirement; no changes necessary.

Web applications that use passive authentication

For web applications that use Access Control for user authentication, Access Control provides the following features and capabilities to web application developers and architects:

- Deep integration with Windows Identity Foundation (WIF).
- Federation with Google, Facebook, Yahoo, Azure Active Directory, and AD FS accounts, and Microsoft accounts.
- Support for the following authentication protocols: OAuth 2.0 Draft 13, WS-Trust, and Web Services Federation (WS-Federation).
- Support for the following token formats: JSON Web Token (JWT), SAML 1.1, SAML 2.0, and Simple Web Token (SWT).
- A home realm discovery experience, integrated into WIF, that allows users to pick the type of account they use to sign in. This experience is hosted by the web application and is fully customizable.
- Token transformation that allows rich customization of the claims received by the web application from Access Control, including:
 - Pass through claims from identity providers.
 - Adding additional custom claims.
 - Simple if-then logic to issue claims under certain conditions.

Unfortunately, there isn't one service that offers all of these equivalent capabilities. You should evaluate which capabilities of Access Control you need, and then choose between using [Azure Active Directory](#), [Azure Active Directory B2C](#) (Azure AD B2C), or another cloud authentication service.

Migrate to Azure Active Directory

A path to consider is integrating your apps and services directly with Azure AD. Azure AD is the cloud-based identity provider for Microsoft work or school accounts. Azure AD is the identity provider for Office 365, Azure, and much more. It provides similar federated authentication capabilities to Access Control, but doesn't support all

Access Control features.

The primary example is federation with social identity providers, such as Facebook, Google, and Yahoo. If your users sign in with these types of credentials, Azure AD is not the solution for you.

Azure AD also doesn't necessarily support the exact same authentication protocols as Access Control. For example, although both Access Control and Azure AD support OAuth, there are subtle differences between each implementation. Different implementations require you to modify code as part of a migration.

However, Azure AD does provide several potential advantages to Access Control customers. It natively supports Microsoft work or school accounts hosted in the cloud, which are commonly used by Access Control customers.

An Azure AD tenant can also be federated to one or more instances of on-premises Active Directory via AD FS. This way, your app can authenticate cloud-based users and users that are hosted on-premises. It also supports the WS-Federation protocol, which makes it relatively straightforward to integrate with a web application by using WIF.

The following table compares the features of Access Control that are relevant to web applications with those features that are available in Azure AD.

At a high level, *Azure Active Directory is probably the best choice for your migration if you let users sign in only with their Microsoft work or school accounts.*

CAPABILITY	ACCESS CONTROL SUPPORT	AZURE AD SUPPORT
Types of accounts		
Microsoft work or school accounts	Supported	Supported
Accounts from Windows Server Active Directory and AD FS	- Supported via federation with an Azure AD tenant - Supported via direct federation with AD FS	Only supported via federation with an Azure AD tenant
Accounts from other enterprise identity management systems	- Possible via federation with an Azure AD tenant - Supported via direct federation	Possible via federation with an Azure AD tenant
Microsoft accounts for personal use	Supported	Supported via the Azure AD v2.0 OAuth protocol, but not over any other protocols
Facebook, Google, Yahoo accounts	Supported	Not supported whatsoever
Protocols and SDK compatibility		
WIF	Supported	Supported, but limited instructions are available
WS-Federation	Supported	Supported
OAuth 2.0	Support for Draft 13	Support for RFC 6749, the most modern specification
WS-Trust	Supported	Not supported
Token formats		

CAPABILITY	ACCESS CONTROL SUPPORT	AZURE AD SUPPORT
JWT	Supported In Beta	Supported
SAML 1.1	Supported	Preview
SAML 2.0	Supported	Supported
SWT	Supported	Not supported
Customizations		
Customizable home realm discovery/account-picking UI	Downloadable code that can be incorporated into apps	Not supported
Upload custom token-signing certificates	Supported	Supported
Customize claims in tokens	<ul style="list-style-type: none"> - Pass through input claims from identity providers - Get access token from identity provider as a claim - Issue output claims based on values of input claims - Issue output claims with constant values 	<ul style="list-style-type: none"> - Cannot pass through claims from federated identity providers - Cannot get access token from identity provider as a claim - Cannot issue output claims based on values of input claims - Can issue output claims with constant values - Can issue output claims based on properties of users synced to Azure AD
Automation		
Automate configuration and management tasks	Supported via Access Control Management Service	Supported via Microsoft Graph and Azure AD Graph API

If you decide that Azure AD is the best migration path for your applications and services, you should be aware of two ways to integrate your app with Azure AD.

To use WS-Federation or WIF to integrate with Azure AD, we recommend following the approach described in [Configure federated single sign-on for a non-gallery application](#). The article refers to configuring Azure AD for SAML-based single sign-on, but also works for configuring WS-Federation. Following this approach requires an Azure AD Premium license. This approach has two advantages:

- You get the full flexibility of Azure AD token customization. You can customize the claims that are issued by Azure AD to match the claims that are issued by Access Control. This especially includes the user ID or Name Identifier claim. To continue to receive consistent user identifiers for your users after you change technologies, ensure that the user IDs issued by Azure AD match those issued by Access Control.
- You can configure a token-signing certificate that is specific to your application, and with a lifetime that you control.

NOTE

This approach requires an Azure AD Premium license. If you are an Access Control customer and you require a premium license for setting up single-sign on for an application, contact us. We'll be happy to provide developer licenses for you to use.

An alternative approach is to follow [this code sample](#), which gives slightly different instructions for setting up WS-Federation. This code sample does not use WIF, but rather, the ASP.NET 4.5 OWIN middleware. However, the instructions for app registration are valid for apps using WIF, and don't require an Azure AD Premium license.

If you choose this approach, you need to understand [signing key rollover in Azure AD](#). This approach uses the Azure AD global signing key to issue tokens. By default, WIF does not automatically refresh signing keys. When Azure AD rotates its global signing keys, your WIF implementation needs to be prepared to accept the changes. For more information, see [Important information about signing key rollover in Azure AD](#).

If you can integrate with Azure AD via the OpenID Connect or OAuth protocols, we recommend doing so. We have extensive documentation and guidance about how to integrate Azure AD into your web application available in our [Azure AD developer guide](#).

Migrate to Azure Active Directory B2C

The other migration path to consider is Azure AD B2C. Azure AD B2C is a cloud authentication service that, like Access Control, allows developers to outsource their authentication and identity management logic to a cloud service. It's a paid service (with free and premium tiers) that is designed for consumer-facing applications that might have up to millions of active users.

Like Access Control, one of the most attractive features of Azure AD B2C is that it supports many different types of accounts. With Azure AD B2C, you can sign in users by using their Microsoft account, or Facebook, Google, LinkedIn, GitHub, or Yahoo accounts, and more. Azure AD B2C also supports "local accounts," or username and passwords that users create specifically for your application. Azure AD B2C also provides rich extensibility that you can use to customize your sign-in flows.

However, Azure AD B2C doesn't support the breadth of authentication protocols and token formats that Access Control customers might require. You also can't use Azure AD B2C to get tokens and query for additional information about the user from the identity provider, Microsoft or otherwise.

The following table compares the features of Access Control that are relevant to web applications with those that are available in Azure AD B2C. At a high level, *Azure AD B2C is probably the right choice for your migration if your application is consumer facing, or if it supports many different types of accounts*.

CAPABILITY	ACCESS CONTROL SUPPORT	AZURE AD B2C SUPPORT
Types of accounts		
Microsoft work or school accounts	Supported	Supported via custom policies
Accounts from Windows Server Active Directory and AD FS	Supported via direct federation with AD FS	Supported via SAML federation by using custom policies
Accounts from other enterprise identity management systems	Supported via direct federation through WS-Federation	Supported via SAML federation by using custom policies
Microsoft accounts for personal use	Supported	Supported

CAPABILITY	ACCESS CONTROL SUPPORT	AZURE AD B2C SUPPORT
Facebook, Google, Yahoo accounts	Supported	Facebook and Google supported natively, Yahoo supported via OpenID Connect federation by using custom policies
Protocols and SDK compatibility		
Windows Identity Foundation (WIF)	Supported	Not supported
WS-Federation	Supported	Not supported
OAuth 2.0	Support for Draft 13	Support for RFC 6749, the most modern specification
WS-Trust	Supported	Not supported
Token formats		
JWT	Supported In Beta	Supported
SAML 1.1	Supported	Not supported
SAML 2.0	Supported	Not supported
SWT	Supported	Not supported
Customizations		
Customizable home realm discovery/account-picking UI	Downloadable code that can be incorporated into apps	Fully customizable UI via custom CSS
Upload custom token-signing certificates	Supported	Custom signing keys, not certificates, supported via custom policies
Customize claims in tokens	<ul style="list-style-type: none"> - Pass through input claims from identity providers - Get access token from identity provider as a claim - Issue output claims based on values of input claims - Issue output claims with constant values 	<ul style="list-style-type: none"> - Can pass through claims from identity providers; custom policies required for some claims - Cannot get access token from identity provider as a claim - Can issue output claims based on values of input claims via custom policies - Can issue output claims with constant values via custom policies
Automation		
Automate configuration and management tasks	Supported via Access Control Management Service	<ul style="list-style-type: none"> - Creation of users allowed via Azure AD Graph API - Cannot create B2C tenants, applications, or policies programmatically

If you decide that Azure AD B2C is the best migration path for your applications and services, begin with the

following resources:

- [Azure AD B2C documentation](#)
- [Azure AD B2C custom policies](#)
- [Azure AD B2C pricing](#)

Migrate to Ping Identity or Auth0

In some cases, you might find that Azure AD and Azure AD B2C aren't sufficient to replace Access Control in your web applications without making major code changes. Some common examples might include:

- Web applications that use WIF or WS-Federation for sign-in with social identity providers such as Google or Facebook.
- Web applications that perform direct federation to an enterprise identity provider over the WS-Federation protocol.
- Web applications that require the access token issued by a social identity provider (such as Google or Facebook) as a claim in the tokens issued by Access Control.
- Web applications with complex token transformation rules that Azure AD or Azure AD B2C can't reproduce.
- Multi-tenant web applications that use ACS to centrally manage federation to many different identity providers

In these cases, you might want to consider migrating your web application to another cloud authentication service.

We recommend exploring the following options. Each of the following options offer capabilities similar to Access Control:

	<p>Auth0 is a flexible cloud identity service that has created high-level migration guidance for customers of Access Control, and supports nearly every feature that ACS does.</p>
	<p>Ping Identity offers two solutions similar to ACS. PingOne is a cloud identity service that supports many of the same features as ACS, and PingFederate is a similar on premises identity product that offers more flexibility. Refer to Ping's ACS retirement guidance for more details on using these products.</p>

Our aim in working with Ping Identity and Auth0 is to ensure that all Access Control customers have a migration path for their apps and services that minimizes the amount of work required to move from Access Control.

Web services that use active authentication

For web services that are secured with tokens issued by Access Control, Access Control offers the following features and capabilities:

- Ability to register one or more *service identities* in your Access Control namespace. Service identities can be used to request tokens.
- Support for the OAuth WRAP and OAuth 2.0 Draft 13 protocols for requesting tokens, by using the following types of credentials:
 - A simple password that's created for the service identity
 - A signed SWT by using a symmetric key or X509 certificate
 - A SAML token issued by a trusted identity provider (typically, an AD FS instance)
- Support for the following token formats: JWT, SAML 1.1, SAML 2.0, and SWT.
- Simple token transformation rules.

Service identities in Access Control are typically used to implement server-to-server authentication.

Migrate to Azure Active Directory

Our recommendation for this type of authentication flow is to migrate to [Azure Active Directory](#). Azure AD is the cloud-based identity provider for Microsoft work or school accounts. Azure AD is the identity provider for Office 365, Azure, and much more.

You can also use Azure AD for server-to-server authentication by using the Azure AD implementation of the OAuth client credentials grant. The following table compares the capabilities of Access Control in server-to-server authentication with those that are available in Azure AD.

CAPABILITY	ACCESS CONTROL SUPPORT	AZURE AD SUPPORT
How to register a web service	Create a relying party in the Access Control management portal	Create an Azure AD web application in the Azure portal
How to register a client	Create a service identity in Access Control management portal	Create another Azure AD web application in the Azure portal
Protocol used	- OAuth WRAP protocol - OAuth 2.0 Draft 13 client credentials grant	OAuth 2.0 client credentials grant
Client authentication methods	- Simple password - Signed SWT - SAML token from a federated identity provider	- Simple password - Signed JWT
Token formats	- JWT - SAML 1.1 - SAML 2.0 - SWT	JWT only
Token transformation	- Add custom claims - Simple if-then claim issuance logic	Add custom claims
Automate configuration and management tasks	Supported via Access Control Management Service	Supported via Microsoft Graph and Azure AD Graph API

For guidance about implementing server-to-server scenarios, see the following resources:

- Service-to-Service section of the [Azure AD developer guide](#)
- [Daemon code sample by using simple password client credentials](#)
- [Daemon code sample by using certificate client credentials](#)

Migrate to Ping Identity or Auth0

In some cases, you might find that the Azure AD client credentials and the OAuth grant implementation aren't sufficient to replace Access Control in your architecture without major code changes. Some common examples might include:

- Server-to-server authentication using token formats other than JWTs.
- Server-to-server authentication using an input token provided by an external identity provider.
- Server-to-server authentication with token transformation rules that Azure AD cannot reproduce.

In these cases, you might consider migrating your web application to another cloud authentication service. We recommend exploring the following options. Each of the following options offer capabilities similar to Access Control:

	<p>Auth0 is a flexible cloud identity service that has created high-level migration guidance for customers of Access Control, and supports nearly every feature that ACS does.</p>
	<p>Ping Identity offers two solutions similar to ACS. PingOne is a cloud identity service that supports many of the same features as ACS, and PingFederate is a similar on premises identity product that offers more flexibility. Refer to Ping's ACS retirement guidance for more details on using these products.</p>

Our aim in working with Ping Identity and Auth0 is to ensure that all Access Control customers have a migration path for their apps and services that minimizes the amount of work required to move from Access Control.

Passthrough authentication

For passthrough authentication with arbitrary token transformation, there is no equivalent Microsoft technology to ACS. If that is what your customers need, Auth0 might be the one who provides the closest approximation solution.

Questions, concerns, and feedback

We understand that many Access Control customers won't find a clear migration path after reading this article. You might need some assistance or guidance in determining the right plan. If you would like to discuss your migration scenarios and questions, please leave a comment on this page.

How to: Reactivate disabled Access Control Service namespaces

7/1/2019 • 3 minutes to read • [Edit Online](#)

On November 2017, we announced that Microsoft Azure Access Control Service (ACS), a service of Azure Active Directory (Azure AD), is being retired on November 7, 2018.

Since then, we've sent emails to the ACS subscriptions' admin email about the ACS retirement 12 months, 9 months, 6 months, 3 months, 1 month, 2 weeks, 1 week, and 1 day before the retirement date of November 7, 2018.

On October 3, 2018, we announced (through email and [a blog post](#)) an extension offer to customers who can't finish their migration before November 7, 2018. The announcement also had instructions for requesting the extension.

Why your namespace is disabled

If you haven't opted in for the extension, we'll start to disable ACS namespaces starting November 7, 2018. You must have requested the extension to February 4, 2019 already; otherwise, you will not be able to enable the namespaces through PowerShell.

NOTE

You must be a service administrator or co-administrator of the subscription to run the PowerShell commands and request an extension.

Find and enable your ACS namespaces

You can use ACS PowerShell to list all your ACS namespaces and reactivate ones that have been disabled.

1. Download and install ACS PowerShell:

- a. Go to the PowerShell Gallery and download [Acs.Namespaces](#).
- b. Install the module:

```
Install-Module -Name Acs.Namespaces
```

- c. Get a list of all possible commands:

```
Get-Command -Module Acs.Namespaces
```

To get help on a specific command, run:

```
Get-Help [Command-Name] -Full
```

where `[Command-Name]` is the name of the ACS command.

2. Connect to ACS using the **Connect-AcsAccount** cmdlet.

You may need to change your execution policy by running **Set-ExecutionPolicy** before you can run the command.

3. List your available Azure subscriptions using the **Get-AcsSubscription** cmdlet.
4. List your ACS namespaces using the **Get-AcsNamespace** cmdlet.
5. Confirm that the namespaces are disabled by confirming that `State` is `Disabled`.

Name	ManagementUrl	Region	State
contoso-test	https://contoso-test.accesscontrol.windows.net	EUWE	Disabled

You can also use `nslookup {your-namespace}.accesscontrol.windows.net` to confirm if the domain is still active.

6. Enable your ACS namespace(s) using the **Enable-AcsNamespace** cmdlet.

Once you've enabled your namespace(s), you can request an extension so that the namespace(s) won't be disabled again before February 4, 2019. After that date, all requests to ACS will fail.

Request an extension

We are taking new extension requests starting on January 21, 2019.

We will start disabling namespaces for customers who have requested extensions to February 4, 2019. You can still re-enable namespaces through PowerShell, but the namespaces will be disabled again after 48 hours.

After March 4, 2019, customers will no longer be able to re-enable any namespaces through PowerShell.

Further extensions will no longer be automatically approved. If you need additional time to migrate, contact [Azure support](#) to provide a detailed migration timeline.

To request an extension

1. Sign in to the Azure portal and create a [new support request](#).
2. Fill in the new support request form as shown in the following example.

SUPPORT REQUEST FIELD	VALUE
Issue type	Technical
Subscription	Set to your subscription
Service	All services
Resource	General question/Resource not available
Problem type	ACS to SAS Migration
Subject	Describe the issue

New support request (preview)

Basics Solutions Details Review + create

Create a new support request to get assistance with billing, subscription, technical or quota management issues. Complete the Basics tab by selecting the options that best describe your problem. Providing detailed, accurate information can help to solve your issues faster. Looking for the old experience? Click [here](#)

<p>* Issue type</p>	<input type="text" value="Technical"/>
<p>* Subscription</p>	<input)="" type="text" value="Microsoft Azure Internal Consumption (c4132b19-5782-43..."/> Can't find your subscription? Show more 
<p>* Service</p>	<input type="radio"/> My services <input checked="" type="radio"/> All services <input type="text" value="Service Bus"/>
<p>* Resource</p>	<input type="text" value="General question / Resource not available"/>
<p>* Problem type</p>	<input type="text" value="ACS to SAS Migration"/>
<p>* Subject</p>	<input type="text" value="In a few words, describe your issue"/>

Help and support

- If you run into any issues after following this how-to, contact [Azure support](#).
- If you have questions or feedback about ACS retirement, contact us at acsfeedback@microsoft.com.

Next steps

- Review the information about ACS retirement in [How to: Migrate from the Azure Access Control Service](#).

How to: Use the Azure AD Graph API

10/15/2019 • 6 minutes to read • [Edit Online](#)

IMPORTANT

We strongly recommend that you use [Microsoft Graph](#) instead of Azure AD Graph API to access Azure Active Directory resources. Our development efforts are now concentrated on Microsoft Graph and no further enhancements are planned for Azure AD Graph API. There are a very limited number of scenarios for which Azure AD Graph API might still be appropriate; for more information, see the [Microsoft Graph](#) or the [Azure AD Graph](#) blog post and [Migrate Azure AD Graph apps to Microsoft Graph](#).

The Azure Active Directory (Azure AD) Graph API provides programmatic access to Azure AD through OData REST API endpoints. Applications can use Azure AD Graph API to perform create, read, update, and delete (CRUD) operations on directory data and objects. For example, you can use Azure AD Graph API to create a new user, view or update user's properties, change user's password, check group membership for role-based access, disable, or delete the user. For more information on Azure AD Graph API features and application scenarios, see [Azure AD Graph API](#) and [Azure AD Graph API prerequisites](#). Azure AD Graph API only works with work or school/organization accounts.

This article applies to Azure AD Graph API. For similar info related to Microsoft Graph API, see [Use the Microsoft Graph API](#).

How to construct a Graph API URL

In Graph API, to access directory data and objects (in other words, resources or entities) against which you want to perform CRUD operations, you can use URLs based on the Open Data (OData) Protocol. The URLs used in Graph API consist of four main parts: service root, tenant identifier, resource path, and query string options:

`https://graph.windows.net/{tenant-identifier}/{resource-path}?[query-parameters]`. Take the example of the following URL: `https://graph.windows.net/contoso.com/groups?api-version=1.6`.

- **Service Root:** In Azure AD Graph API, the service root is always <https://graph.windows.net>.
- **Tenant identifier:** This section can be a verified (registered) domain name, in the preceding example, contoso.com. It can also be a tenant object ID or the "myorganization" or "me" alias. For more information, see [Addressing entities and operations in Azure AD Graph API](#).
- **Resource path:** This section of a URL identifies the resource to be interacted with (users, groups, a particular user, or a particular group, etc.) In the example above, it is the top level "groups" to address that resource set. You can also address a specific entity, for example "users/{objectId}" or "users/userPrincipalName".
- **Query parameters:** A question mark (?) separates the resource path section from the query parameters section. The "api-version" query parameter is required on all requests in Azure AD Graph API. Azure AD Graph API also supports the following OData query options: **\$filter**, **\$orderby**, **\$expand**, **\$top**, and **\$format**. The following query options are not currently supported: **\$count**, **\$inlinecount**, and **\$skip**. For more information, see [Supported Queries, Filters, and Paging Options in Azure AD Graph API](#).

Graph API versions

You specify the version for a Graph API request in the "api-version" query parameter. For version 1.5 and later, you use a numerical version value; api-version=1.6. For earlier versions, you use a date string that adheres to the format YYYY-MM-DD; for example, api-version=2013-11-08. For preview features, use the string "beta"; for example, api-version=beta. For more information about differences between Graph API versions, see [Azure AD](#)

Graph API versioning.

Graph API metadata

To return the Azure AD Graph API metadata file, add the “\$metadata” segment after the tenant-identifier in the URL. For example, the following URL returns metadata for a demo company:

[https://graph.windows.net/GraphDir1.OnMicrosoft.com/\\$metadata?api-version=1.6](https://graph.windows.net/GraphDir1.OnMicrosoft.com/$metadata?api-version=1.6). You can enter this URL in the address bar of a web browser to see the metadata. The CSDL metadata document returned describes the entities and complex types, their properties, and the functions and actions exposed by the version of Graph API you requested. Omitting the api-version parameter returns metadata for the most recent version.

Common queries

[Azure AD Graph API common queries](#) lists common queries that can be used with the Azure AD Graph, including queries that can be used to access top-level resources in your directory and queries to perform operations in your directory.

For example, <https://graph.windows.net/contoso.com/tenantDetails?api-version=1.6> returns company information for directory contoso.com.

Or <https://graph.windows.net/contoso.com/users?api-version=1.6> lists all user objects in the directory contoso.com.

Using the Azure AD Graph Explorer

You can use the Azure AD Graph Explorer for the Azure AD Graph API to query the directory data as you build your application.

The following screenshot is the output you would see if you were to navigate to the Azure AD Graph Explorer, sign in, and enter <https://graph.windows.net/GraphDir1.OnMicrosoft.com/users?api-version=1.6> to display all the users in the signed-in user's directory:

The screenshot shows the Azure AD Graph Explorer web application. At the top, there is a navigation bar with icons for back, forward, refresh, and search, followed by the URL 'graphexplorer.azurewebsites.net/#'. Below the URL is a header bar with tabs for 'Documentation' and 'Logout'. The main area has a title 'Azure AD Graph Explorer' and a status bar showing 'GET' and 'https://graph.windows.net/myorganization/users?api-version=1.6'. The status bar also includes '134 ms', 'api-version=1.6', and 'Go' buttons. The central content area displays a JSON response for a user object. The JSON structure includes properties like 'odata.type', 'objectType', 'objectId', 'deletionTimestamp', 'accountEnabled', 'signInNames', 'assignedLicenses', and 'assignedPlans'. The 'assignedPlans' section contains details such as 'assignedTimestamp', 'capabilityStatus', 'service', and 'servicePlanId'. At the bottom of the page, there is a 'Response Headers' section.

Load the Azure AD Graph Explorer: To load the tool, navigate to <https://graphexplorer.azurewebsites.net/>. Click **Login** and sign-in with your Azure AD account credentials to run the Azure AD Graph Explorer against your tenant. If you run Azure AD Graph Explorer against your own tenant, either you or your administrator needs to consent during sign-in. If you have an Office 365 subscription, you automatically have an Azure AD tenant. The credentials you use to sign in to Office 365 are, in fact, Azure AD accounts, and you can use these credentials with Azure AD Graph Explorer.

Run a query: To run a query, type your query in the request text box and click **GET** or click the **enter** key. The results are displayed in the response box. For example,

`https://graph.windows.net/myorganization/groups?api-version=1.6` lists all group objects in the signed-in user's directory.

Note the following features and limitations of the Azure AD Graph Explorer:

- Autocomplete capability on resource sets. To see this functionality, click on the request text box (where the company URL appears). You can select a resource set from the dropdown list.
- Request history.
- Supports the "me" and "myorganization" addressing aliases. For example, you can use `https://graph.windows.net/me?api-version=1.6` to return the user object of the signed-in user or `https://graph.windows.net/myorganization/users?api-version=1.6` to return all users in the signed-in user's directory.
- Supports full CRUD operations against your own directory using `POST`, `GET`, `PATCH` and `DELETE`.
- A response headers section. This section can be used to help troubleshoot issues that occur when running queries.
- A JSON viewer for the response with expand and collapse capabilities.
- No support for displaying or uploading a thumbnail photo.

Using Fiddler to write to the directory

For the purposes of this Quickstart guide, you can use the Fiddler Web Debugger to practice performing 'write' operations against your Azure AD directory. For example, you can get and upload a user's profile photo (which is not possible with Azure AD Graph Explorer). For more information and to install Fiddler, see <https://www.telerik.com/fiddler>.

In the example below, you use Fiddler Web Debugger to create a new security group 'MyTestGroup' in your Azure AD directory.

Obtain an access token: To access Azure AD Graph, clients are required to successfully authenticate to Azure AD first. For more information, see [Authentication scenarios for Azure AD](#).

Compose and run a query: Complete the following steps:

1. Open Fiddler Web Debugger and switch to the **Composer** tab.
2. Since you want to create a new security group, select **Post** as the HTTP method from the pull-down menu.
For more information about operations and permissions on a group object, see [Group](#) within the [Azure AD Graph REST API reference](#).
3. In the field next to **Post**, type in the following request URL:

`https://graph.windows.net/{mytenantdomain}/groups?api-version=1.6`.

NOTE

You must substitute {mytenantdomain} with the domain name of your own Azure AD directory.

4. In the field directly below Post pull-down, type the following HTTP header:

```
Host: graph.windows.net
Authorization: Bearer <your access token>
Content-Type: application/json
```

NOTE

Substitute your <your access token> with the access token for your Azure AD directory.

5. In the **Request body** field, type the following JSON:

```
{  
    "displayName": "MyTestGroup",  
    "mailNickname": "MyTestGroup",  
    "mailEnabled": "false",  
    "securityEnabled": true  
}
```

For more information about creating groups, see [Create Group](#).

For more information on Azure AD entities and types that are exposed by Graph and information about the operations that can be performed on them with Graph, see [Azure AD Graph REST API reference](#).

Next steps

- Learn more about the [Azure AD Graph API](#)
- Learn more about [Azure AD Graph API permission scopes](#)

Azure Active Directory Authentication Libraries

10/15/2019 • 3 minutes to read • [Edit Online](#)

The Azure Active Directory Authentication Library (ADAL) v1.0 enables application developers to authenticate users to cloud or on-premises Active Directory (AD), and obtain tokens for securing API calls. ADAL makes authentication easier for developers through features such as:

- Configurable token cache that stores access tokens and refresh tokens
- Automatic token refresh when an access token expires and a refresh token is available
- Support for asynchronous method calls

NOTE

Looking for the Azure AD v2.0 libraries (MSAL)? Checkout the [MSAL library guide](#).

Microsoft-supported Client Libraries

PLATFORM	LIBRARY	DOWNLOAD	SOURCE CODE	SAMPLE	REFERENCE
.NET Client, Windows Store, UWP, Xamarin iOS and Android	ADAL .NET v3	NuGet	GitHub	Desktop app	Reference
.NET Client, Windows Store, Windows Phone 8.1	ADAL .NET v2	NuGet	GitHub	Desktop app	
JavaScript	ADAL.js	GitHub	GitHub	Single-page app	
iOS, macOS	ADAL	GitHub	GitHub	iOS app	Reference
Android	ADAL	Maven	GitHub	Android app	JavaDocs
Node.js	ADAL	npm	GitHub	Node.js web app	Reference
Java	ADAL4J	Maven	GitHub	Java web app	Reference
Python	ADAL	GitHub	GitHub	Python web app	Reference

Microsoft-supported Server Libraries

PLATFORM	LIBRARY	DOWNLOAD	SOURCE CODE	SAMPLE	REFERENCE
.NET	OWIN for AzureAD	NuGet	GitHub	MVC App	

Platform	Library	Download	Source Code	Sample	Reference
.NET	OWIN for OpenIDConnect	NuGet	GitHub	Web App	
.NET	OWIN for WS-Federation	NuGet	GitHub	MVC Web App	
.NET	Identity Protocol Extensions for .NET 4.5	NuGet	GitHub		
.NET	JWT Handler for .NET 4.5	NuGet	GitHub		
Node.js	Azure AD Passport	npm	GitHub	Web API	

Scenarios

Here are three common scenarios for using ADAL in a client that accesses a remote resource:

Authenticating users of a native client application running on a device

In this scenario, a developer has a mobile client or desktop application that needs to access a remote resource, such as a web API. The web API does not allow anonymous calls and must be called in the context of an authenticated user. The web API is pre-configured to trust access tokens issued by a specific Azure AD tenant. Azure AD is pre-configured to issue access tokens for that resource. To invoke the web API from the client, the developer uses ADAL to facilitate authentication with Azure AD. The most secure way to use ADAL is to have it render the user interface for collecting user credentials (rendered as browser window).

ADAL makes it easy to authenticate the user, obtain an access token and refresh token from Azure AD, and then call the web API using the access token.

For a code sample that demonstrates this scenario using authentication to Azure AD, see [Native Client WPF Application to Web API](#).

Authenticating a confidential client application running on a web server

In this scenario, a developer has an application running on a server that needs to access a remote resource, such as a web API. The web API does not allow anonymous calls, so it must be called from an authorized service. The web API is pre-configured to trust access tokens issued by a specific Azure AD tenant. Azure AD is pre-configured to issue access tokens for that resource to a service with client credentials (client ID and secret). ADAL facilitates authentication of the service with Azure AD returning an access token that can be used to call the web API. ADAL also handles managing the lifetime of the access token by caching it and renewing it as necessary. For a code sample that demonstrates this scenario, see [Daemon console Application to Web API](#).

Authenticating a confidential client application running on a server, on behalf of a user

In this scenario, a developer has a web application running on a server that needs to access a remote resource, such as a web API. The web API does not allow anonymous calls, so it must be called from an authorized service on behalf of an authenticated user. The web API is pre-configured to trust access tokens issued by a specific Azure AD tenant, and Azure AD is pre-configured to issue access tokens for that resource to a service with client credentials. Once the user is authenticated in the web application, the application can get an authorization code for the user from Azure AD. The web application can then use ADAL to obtain an access token and refresh token on behalf of a user using the authorization code and client credentials associated with the application from Azure AD. Once the web application is in possession of the access token, it can call the web API until the token

expires. When the token expires, the web application can use ADAL to get a new access token by using the refresh token that was previously received. For a code sample that demonstrates this scenario, see [Native client to Web API to Web API](#).

See Also

- [The Azure Active Directory developer's guide](#)
- [Authentication scenarios for Azure Active directory](#)
- [Azure Active Directory code samples](#)

Azure Active Directory Graph API

8/7/2019 • 4 minutes to read • [Edit Online](#)

IMPORTANT

As of February 2019, we started the process to deprecate some earlier versions of Azure Active Directory Graph API in favor of the Microsoft Graph API.

For details, updates, and time frames, see [Microsoft Graph or the Azure AD Graph](#) in the Office Dev Center.

Moving forward, applications should use the Microsoft Graph API.

This article applies to Azure AD Graph API. For similar info related to Microsoft Graph API, see [Use the Microsoft Graph API](#).

The Azure Active Directory Graph API provides programmatic access to Azure AD through REST API endpoints. Applications can use Azure AD Graph API to perform create, read, update, and delete (CRUD) operations on directory data and objects. For example, Azure AD Graph API supports the following common operations for a user object:

- Create a new user in a directory
- Get a user's detailed properties, such as their groups
- Update a user's properties, such as their location and phone number, or change their password
- Check a user's group membership for role-based access
- Disable a user's account or delete it entirely

Additionally, you can perform similar operations on other objects such as groups and applications. To call Azure AD Graph API on a directory, your application must be registered with Azure AD. Your application must also be granted access to Azure AD Graph API. This access is normally achieved through a user or admin consent flow.

To begin using the Azure Active Directory Graph API, see the [Azure AD Graph API quickstart guide](#), or view the [interactive Azure AD Graph API reference documentation](#).

Features

Azure AD Graph API provides the following features:

- **REST API Endpoints:** Azure AD Graph API is a RESTful service comprised of endpoints that are accessed using standard HTTP requests. Azure AD Graph API supports XML or Javascript Object Notation (JSON) content types for requests and responses. For more information, see [Azure AD Graph REST API reference](#).
- **Authentication with Azure AD:** Every request to Azure AD Graph API must be authenticated by appending a JSON Web Token (JWT) in the Authorization header of the request. This token is acquired by making a request to Azure AD's token endpoint and providing valid credentials. You can use the OAuth 2.0 client credentials flow or the authorization code grant flow to acquire a token to call the Graph. For more information, [OAuth 2.0 in Azure AD](#).
- **Role-Based Authorization (RBAC):** Security groups are used to perform RBAC in Azure AD Graph API. For example, if you want to determine whether a user has access to a specific resource, the application can call the [Check group membership \(transitive\)](#) operation, which returns true or false.
- **Differential Query:** Differential query allows you to track changes in a directory between two time periods

without having to make frequent queries to Azure AD Graph API. This type of request will return only the changes made between the previous differential query request and the current request. For more information, see [Azure AD Graph API differential query](#).

- **Directory Extensions:** You can add custom properties to directory objects without requiring an external data store. For example, if your application requires a Skype ID property for each user, you can register the new property in the directory and it will be available for use on every user object. For more information, see [Azure AD Graph API directory schema extensions](#).
- **Secured by permission scopes:** Azure AD Graph API exposes permission scopes that enable secure access to Azure AD data using OAuth 2.0. It supports a variety of client app types, including:
 - user interfaces that are given delegated access to data via authorization from the signed-in user (delegated)
 - service/daemon applications that operate in the background without a signed-in user being present and use application-defined role-based access controlBoth delegated and application permissions represent a privilege exposed by the Azure AD Graph API and can be requested by client applications through application registration permissions features in the [Azure portal](#). [Azure AD Graph API permission scopes](#) provides information on what's available for use by your client application.

Scenarios

Azure AD Graph API enables many application scenarios. The following scenarios are the most common:

- **Line of Business (Single Tenant) Application:** In this scenario, an enterprise developer works for an organization that has an Office 365 subscription. The developer is building a web application that interacts with Azure AD to perform tasks such as assigning a license to a user. This task requires access to the Azure AD Graph API, so the developer registers the single tenant application in Azure AD and configures read and write permissions for Azure AD Graph API. Then the application is configured to use either its own credentials or those of the currently sign-in user to acquire a token to call the Azure AD Graph API.
- **Software as a Service Application (Multi-Tenant):** In this scenario, an independent software vendor (ISV) is developing a hosted multi-tenant web application that provides user management features for other organizations that use Azure AD. These features require access to directory objects, so the application needs to call the Azure AD Graph API. The developer registers the application in Azure AD, configures it to require read and write permissions for Azure AD Graph API, and then enables external access so that other organizations can consent to use the application in their directory. When a user in another organization authenticates to the application for the first time, they are shown a consent dialog with the permissions the application is requesting. Granting consent will then give the application those requested permissions to Azure AD Graph API in the user's directory. For more information on the consent framework, see [Overview of the consent framework](#).

Next steps

To begin using the Azure Active Directory Graph API, see the following topics:

- [Azure AD Graph API quickstart guide](#)
- [Azure AD Graph REST documentation](#)

Azure Active Directory app manifest

10/15/2019 • 11 minutes to read • [Edit Online](#)

The application manifest contains a definition of all the attributes of an application object in the Microsoft identity platform. It also serves as a mechanism for updating the application object. For more info on the Application entity and its schema, see the [Graph API Application entity documentation](#).

You can configure an app's attributes through the Azure portal or programmatically using [REST API](#) or [PowerShell](#). However, there are some scenarios where you'll need to edit the app manifest to configure an app's attribute. These scenarios include:

- If you registered the app as Azure AD multi-tenant and personal Microsoft accounts, you can't change the supported Microsoft accounts in the UI. Instead, you must use the application manifest editor to change the supported account type.
- If you need to define permissions and roles that your app supports, you must modify the application manifest.

Configure the app manifest

To configure the application manifest:

1. Sign in to the [Azure portal](#).
2. Select the **Azure Active Directory** service, and then select **App registrations**.
3. Select the app you want to configure.
4. From the app's **Overview** page, select the **Manifest** section. A web-based manifest editor opens, allowing you to edit the manifest within the portal. Optionally, you can select **Download** to edit the manifest locally, and then use **Upload** to reapply it to your application.

Manifest reference

NOTE

If you can't see the **Example value** column after the **Description**, maximize your browser window and scroll/swipe until you see the **Example value** column.

KEY	VALUE TYPE	DESCRIPTION	EXAMPLE VALUE
<code>accessTokenAcceptedVersion</code>	Nullable Int32	<p>Specifies the access token version expected by the resource. This changes the version and format of the JWT produced independent of the endpoint or client used to request the access token.</p> <p>The endpoint used, v1.0 or v2.0, is chosen by the client and only impacts the version of id_tokens. Resources need to explicitly configure <code>accessTokenAcceptedVersion</code> to indicate the supported access token format.</p> <p>Possible values for <code>accessTokenAcceptedVersion</code> are 1, 2, or null. If the value is null, this defaults to 1, which corresponds to the v1.0 endpoint.</p> <p>If <code>signInAudience</code> is <code>AzureADandPersonalMicrosoftAccount</code>, the value must be 2</p>	2
<code>addIns</code>	Collection	Defines custom behavior that a consuming service can use to call an app in specific contexts. For example, applications that can render file streams may set the addIns property for its "FileHandler" functionality. This will let services like Office 365 call the application in the context of a document the user is working on.	{ "id": "968A844F-7A47-430C-9163-07AE7C31D407", "type": "FileHandler", "properties": [{"key": "version", "value": "2"}] }

KEY	VALUE TYPE	DESCRIPTION	EXAMPLE VALUE
<code>allowPublicClient</code>	Boolean	Specifies the fallback application type. Azure AD infers the application type from the replyUrlsWithType by default. There are certain scenarios where Azure AD cannot determine the client app type (e.g. ROPC flow where HTTP request happens without a URL redirection). In those cases Azure AD will interpret the application type based on the value of this property. If this value is set to true the fallback application type is set as public client, such as an installed app running on a mobile device. The default value is false which means the fallback application type is confidential client such as web app.	<code>false</code>
<code>availableToOtherTenants</code>	Boolean	true if the application is shared with other tenants; otherwise, false. <i>Note: This is available only in App registrations (Legacy) experience. Replaced by <code>signInAudience</code> in the App registrations experience.</i>	
<code>appId</code>	String	Specifies the unique identifier for the app that is assigned to an app by Azure AD.	<code>"601790de-b632-4f57-9523-ee7cb6ceba95"</code>
<code>appRoles</code>	Collection	Specifies the collection of roles that an app may declare. These roles can be assigned to users, groups, or service principals. For more examples and info, see Add app roles in your application and receive them in the token	<pre>[{ "allowedMemberTypes": ["User"], "description":"Read-only access to device information", "displayName":"Read Only", "id":guid, "isEnabled":true, "value":"ReadOnly" }]</pre>
<code>displayName</code>	String	The display name for the app. <i>Note: This is available only in App registrations (Legacy) experience. Replaced by <code>name</code> in the App registrations experience.</i>	<code>"MyRegisteredApp"</code>
<code>errorUrl</code>	String	Unsupported.	
<code>groupMembershipClaims</code>	String	Configures the <code>groups</code> claim issued in a user or OAuth 2.0 access token that the app expects. To set this attribute, use one of the following valid string values: - <code>"None"</code> - <code>"SecurityGroup"</code> (for security groups and Azure AD roles) - <code>"A11"</code> (this will get all of the security groups, distribution groups, and Azure AD directory roles that the signed-in user is a member of.)	<code>"SecurityGroup"</code>
<code>homepage</code>	String	The URL to the application's homepage. <i>Note: This is available only in App registrations (Legacy) experience. Replaced by <code>signInUrl</code> in the App registrations experience.</i>	<code>"https://MyRegisteredApp"</code>
<code>objectId</code>	String	The unique identifier for the app in the directory. <i>Note: This is available only in App registrations (Legacy) experience. Replaced by <code>id</code> in the App registrations experience.</i>	<code>"f7f9acf-ae0c-4d6c-b489-0a81dc1652dd"</code>

KEY	VALUE TYPE	DESCRIPTION	EXAMPLE VALUE
<code>optionalClaims</code>	String	The optional claims returned in the token by the security token service for this specific app. At this time, apps that support both personal accounts and Azure AD (registered through the app registration portal) cannot use optional claims. However, apps registered for just Azure AD using the v2.0 endpoint can get the optional claims they requested in the manifest. For more info, see optional claims .	<code>null</code>
<code>id</code>	String	The unique identifier for the app in the directory. This ID is not the identifier used to identify the app in any protocol transaction. It's used for the referencing the object in directory queries.	<code>"f7f9acfc-ae0c-4d6c-b489-0a81dc1652dd"</code>
<code>identifierUris</code>	String Array	User-defined URI(s) that uniquely identify a Web app within its Azure AD tenant, or within a verified custom domain if the app is multi-tenant.	<code>["https://MyRegisteredApp"]</code>
<code>informationalUrls</code>	String	Specifies the links to the app's terms of service and privacy statement. The terms of service and privacy statement are surfaced to users through the user consent experience. For more info, see How to: Add Terms of service and privacy statement for registered Azure AD apps .	<code>{ "marketing": "https://MyRegisteredApp/marketing", "privacy": "https://MyRegisteredApp/privacy", "support": "https://MyRegisteredApp/support", "termsOfService": "https://MyRegisteredApp/terms" }</code>
<code>keyCredentials</code>	Collection	Holds references to app-assigned credentials, string-based shared secrets and X.509 certificates. These credentials are used when requesting access tokens (when the app is acting as a client rather than as resource).	<code>[{ "customKeyIdentifier": null, "enddate": "2018-09-13T00:00:00Z", "keyId": "<guid>", "startdate": "2017-09-12T00:00:00Z", "type": "AsymmetricX509Cert", "usage": "Verify", "value": null }]</code>
<code>knownClientApplications</code>	String Array	Used for bundling consent if you have a solution that contains two parts: a client app and a custom web API app. If you enter the appId of the client app into this value, the user will only have to consent once to the client app. Azure AD will know that consenting to the client means implicitly consenting to the web API and will automatically provision service principals for both the client and web API at the same time. Both the client and the web API app must be registered in the same tenant.	<code>["f7f9acfc-ae0c-4d6c-b489-0a81dc1652dd"]</code>
<code>logoUrl</code>	String	Read only value that points to the CDN URL to logo that was uploaded in the portal.	<code>"https://MyRegisteredAppLogo"</code>
<code>logoutUrl</code>	String	The URL to log out of the app.	<code>"https://MyRegisteredAppLogout"</code>
<code>name</code>	String	The display name for the app.	<code>"MyRegisteredApp"</code>
<code>oauth2AllowImplicitFlow</code>	Boolean	Specifies whether this web app can request OAuth2.0 implicit flow access tokens. The default is false. This flag is used for browser-based apps, like Javascript single-page apps. To learn more, enter <code>OAuth 2.0 implicit grant flow</code> in the table of contents and see the topics about implicit flow.	<code>false</code>

KEY	VALUE TYPE	DESCRIPTION	EXAMPLE VALUE
<code>oauth2AllowIdTokenImplicitFlow</code>	Boolean	Specifies whether this web app can request OAuth2.0 implicit flow ID tokens. The default is false. This flag is used for browser-based apps, like Javascript single-page apps.	<code>false</code>
<code>oauth2Permissions</code>	Collection	Specifies the collection of OAuth 2.0 permission scopes that the web API (resource) app exposes to client apps. These permission scopes may be granted to client apps during consent.	[{ "adminConsentDescription": "Allow the app to access resources on behalf of the signed-in user.", "adminConsentDisplayName": "Access resource1", "id": "<guid>", "isEnabled": true, "type": "User", "userConsentDescription": "Allow the app to access resource1 on your behalf.", "userConsentDisplayName": "Access resources", "value": "user_impersonation" }]
<code>oauth2RequiredPostResponse</code>	Boolean	Specifies whether, as part of OAuth 2.0 token requests, Azure AD will allow POST requests, as opposed to GET requests. The default is false, which specifies that only GET requests will be allowed.	<code>false</code>
<code>parentalControlSettings</code>	String	<code>countriesBlockedForMinors</code> specifies the countries in which the app is blocked for minors. <code>legalAgeGroupRule</code> specifies the legal age group rule that applies to users of the app. Can be set to <code>Allow</code> , <code>RequireConsentForPrivacyServices</code> , <code>RequireConsentForMinors</code> , <code>RequireConsentForKids</code> , or <code>BlockMinors</code> .	{ "countriesBlockedForMinors": [], "legalAgeGroupRule": "Allow" }
<code>passwordCredentials</code>	Collection	See the description for the <code>keyCredentials</code> property.	[{ "customKeyIdentifier": null, "endDate": "2018-10-19T17:59:59.6521653Z", "keyId": "<guid>", "startDate": "2016-10-19T17:59:59.6521653Z", "value": null }]
<code>preAuthorizedApplications</code>	Collection	Lists applications and requested permissions for implicit consent. Requires an admin to have provided consent to the application. <code>preAuthorizedApplications</code> do not require the user to consent to the requested permissions. Permissions listed in <code>preAuthorizedApplications</code> do not require user consent. However, any additional requested permissions not listed in <code>preAuthorizedApplications</code> require user consent.	[{ "appId": "abcdefg2-000a-1111-a0e5-812ed8dd72e8", "permissionIds": ["8748f7db-21fe-4c83-8ab5-53033933c8f1"] }]
<code>publicClient</code>	Boolean	Specifies whether this application is a public client (such as an installed application running on a mobile device). <i>Note: This is available only in App registrations (Legacy) experience. Replaced by <code>allowPublicClient</code> in the App registrations experience.</i>	
<code>publisherDomain</code>	String	The verified publisher domain for the application. Read-only.	https://www.contoso.com

KEY	VALUE TYPE	DESCRIPTION	EXAMPLE VALUE
<code>replyUrls</code>	String array	This multi-value property holds the list of registered redirect_uri values that Azure AD will accept as destinations when returning tokens. <i>Note: This is available only in App registrations (Legacy) experience. Replaced by <code>replyUrlsWithType</code> in the App registrations experience.</i>	
<code>replyUrlsWithType</code>	Collection	This multi-value property holds the list of registered redirect_uri values that Azure AD will accept as destinations when returning tokens. Each URI value should contain an associated app type value. Supported type values are: <ul style="list-style-type: none">• <code>Web</code>• <code>InstalledClient</code> Learn more about replyUrl restrictions and limitations .	<pre>"replyUrlsWithType": [{ "url": "https://localhost:4400/service", "type": "InstalledClient" }]</pre>
<code>requiredResourceAccess</code>	Collection	With dynamic consent, <code>requiredResourceAccess</code> drives the admin consent experience and the user consent experience for users who are using static consent. However, this does not drive the user consent experience for the general case. <code>resourceAppId</code> is the unique identifier for the resource that the app requires access to. This value should be equal to the appid declared on the target resource app. <code>resourceAccess</code> is an array that lists the OAuth2.0 permission scopes and app roles that the app requires from the specified resource. Contains the <code>id</code> and <code>type</code> values of the specified resources.	<pre>[{ "resourceAppId": "00000002-0000-0000-0000-000000000000", "resourceAccess": [{ "id": "311a71cc-e848-46a1-bdf8-97ff7156d8e6", "type": "Scope" }]]</pre>
<code>samlMetadataUrl</code>	String	The URL to the SAML metadata for the app.	<code>https://MyRegisteredAppSAMLMetadata</code>
<code>signInUrl</code>	String	Specifies the URL to the app's home page.	<code>https://MyRegisteredApp</code>
<code>signInAudience</code>	String	Specifies what Microsoft accounts are supported for the current application. Supported values are: <ul style="list-style-type: none">• AzureADMyOrg - Users with a Microsoft work or school account in my organization's Azure AD tenant (i.e. single tenant)• AzureADMultipleOrgs - Users with a Microsoft work or school account in any organization's Azure AD tenant (i.e. multi-tenant)• AzureADandPersonalMicrosoftAccount - Users with a personal Microsoft account, or a work or school account in any organization's Azure AD tenant	<code>AzureADandPersonalMicrosoftAccount</code>
<code>tags</code>	String Array	Custom strings that can be used to categorize and identify the application.	<pre>["ProductionApp"]</pre>

Common issues

Manifest limits

An application manifest has multiple attributes that are referred to as collections; for example, `approles`, `keycredentials`, `knownClientApplications`, `identifierUris`, `redirectUris`, `requiredResourceAccess`, and `oauth2Permissions`. Within the complete application manifest for any application, the total number of entries in all the collections combined has been capped at 1200. If you already have 100 redirect URIs specified in the application manifest, then you're only left with 1100 remaining entries to use across all other collections combined that make up the manifest.

NOTE

In case you try to add more than 1200 entries in the application manifest, you may see an error **"Failed to update application xxxxxx. Error details: The size of the manifest has exceeded its limit. Please reduce the number of values and retry your request."**

Unsupported attributes

The application manifest represents the schema of the underlying application model in Azure AD. As the underlying schema evolves, the manifest editor will be updated to reflect the new schema from time to time. As a result, you may notice new attributes showing up in the application manifest. In rare occasions, you may notice a syntactic or semantic change in the existing attributes or you may find an attribute that existed previously are not supported anymore. For example, you will see new attributes in the [App registrations](#) which are known with a different name in the App registrations (Legacy) experience.

APP REGISTRATIONS (LEGACY)	APP REGISTRATIONS
availableToOtherTenants	signInAudience
displayName	name
errorUrl	-
homepage	signInUrl
objectId	Id
publicClient	allowPublicClient
replyUrls	replyUrlsWithType

For descriptions for these attributes, see the [manifest reference](#) section.

When you try to upload a previously downloaded manifest, you may see one of the following errors. This is likely because the manifest editor now supports a newer version of the schema, which doesn't match with the one you're trying to upload.

- "Failed to update xxxxxx application. Error detail: Invalid object identifier 'undefined'. []."
- "Failed to update xxxxxx application. Error detail: One or more property values specified are invalid. []."
- "Failed to update xxxxxx application. Error detail: Not allowed to set availableToOtherTenants in this api version for update. []."
- "Failed to update xxxxxx application. Error detail: Updates to 'replyUrls' property is not allowed for this application. Use 'replyUrlsWithType' property instead. []."
- "Failed to update xxxxxx application. Error detail: A value without a type name was found and no expected type is available. When the model is specified, each value in the payload must have a type which can be either specified in the payload, explicitly by the caller or implicitly inferred from the parent value. []"

When you see one of these errors, we recommend the following:

1. Edit the attributes individually in the manifest editor instead of uploading a previously downloaded manifest. Use the [manifest reference](#) table to understand the syntax and semantics of old and new attributes so that you can successfully edit the attributes you're interested in.
2. If your workflow requires you to save the manifests in your source repository for use later, we suggest rebasing the saved manifests in your repository with the one you see in the [App registrations](#) experience.

Next steps

- For more info on the relationship between an app's application and service principal object(s), see [Application and service principal objects in Azure AD](#).
- See the [Microsoft identity platform developer glossary](#) for definitions of some of the core Microsoft identity platform developer concepts.

Use the following comments section to provide feedback that helps refine and shape our content.

Authentication and authorization error codes

8/30/2019 • 21 minutes to read • [Edit Online](#)

Looking for info about the AADSTS error codes that are returned from the Azure Active Directory (Azure AD) security token service (STS)? Read this document to find AADSTS error descriptions, fixes, and some suggested workarounds.

NOTE

This information is preliminary and subject to change. Have a question or can't find what you're looking for? Create a GitHub issue or see [Support and help options for developers](#) to learn about other ways you can get help and support.

This documentation is provided for developer and admin guidance, but should never be used by the client itself. Error codes are subject to change at any time in order to provide more granular error messages that are intended to help the developer while building their application. Apps that take a dependency on text or error code numbers will be broken over time.

Lookup current error code information

Error codes and messages are subject to change. For the most current info, take a look at the <https://login.microsoftonline.com/error> page to find AADSTS error descriptions, fixes, and some suggested workarounds.

Search on the numeric part of the returned error code. For example, if you received the error code "AADSTS16000" then do a search in <https://login.microsoftonline.com/error> for "16000". You can also link directly to a specific error by adding the error code number to the URL: <https://login.microsoftonline.com/error?code=16000>.

AADSTS error codes

ERROR	DESCRIPTION
AADSTS16000	SelectUserAccount - This is an interrupt thrown by Azure AD, which results in UI that allows the user to select from among multiple valid SSO sessions. This error is fairly common and may be returned to the application if <code>prompt=none</code> is specified.
AADSTS16001	UserAccountSelectionInvalid - You'll see this error if the user clicks on a tile that the session select logic has rejected. When triggered, this error allows the user to recover by picking from an updated list of tiles/sessions, or by choosing another account. This error can occur because of a code defect or race condition.
AADSTS16002	AppSessionSelectionInvalid - The app-specified SID requirement was not met.
AADSTS16003	SsoUserAccountNotFoundInResourceTenant - Indicates that the user hasn't been explicitly added to the tenant.

ERROR	DESCRIPTION
AADSTS17003	CredentialKeyProvisioningFailed - Azure AD can't provision the user key.
AADSTS20001	WsFedSignInResponseError - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS20012	WsFedMessageInvalid - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS20033	FedMetadataInvalidTenantName - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS40008	OAuth2IdPUnretryableServerError - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS40009	OAuth2IdPRefreshTokenRedemptionUserError - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS40010	OAuth2IdPRetryableServerError - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS40015	OAuth2IdPAuthCodeRedemptionUserError - There's an issue with your federated Identity Provider. Contact your IDP to resolve this issue.
AADSTS50000	TokenIssuanceError - There's an issue with the sign-in service. Open a support ticket to resolve this issue.
AADSTS50001	InvalidResource - The resource is disabled or does not exist. Check your app's code to ensure that you have specified the exact resource URL for the resource you are trying to access.
AADSTS50002	NotAllowedTenant - Sign-in failed because of a restricted proxy access on the tenant. If it's your own tenant policy, you can change your restricted tenant settings to fix this issue.
AADSTS50003	MissingSigningKey - Sign-in failed because of a missing signing key or certificate. This might be because there was no signing key configured in the app. Check out the resolutions outlined at https://docs.microsoft.com/azure/active-directory/application-sign-in-problem-federated-sso-gallery#certificate-or-key-not-configured . If you still see issues, contact the app owner or an app admin.
AADSTS50005	DevicePolicyError - User tried to log in to a device from a platform that's currently not supported through Conditional Access policy.
AADSTS50006	InvalidSignature - Signature verification failed because of an invalid signature.

ERROR	DESCRIPTION
AADSTS50007	PartnerEncryptionCertificateMissing - The partner encryption certificate was not found for this app. Open a support ticket with Microsoft to get this fixed.
AADSTS50008	InvalidSamlToken - SAML assertion is missing or misconfigured in the token. Contact your federation provider.
AADSTS50010	AudienceUriValidationFailed - Audience URI validation for the app failed since no token audiences were configured.
AADSTS50011	InvalidReplyTo - The reply address is missing, misconfigured, or does not match reply addresses configured for the app. As a resolution ensure to add this missing reply address to the Azure Active Directory application or have someone with the permissions to manage your application in Active Directory do this for you.
AADSTS50012	<p>AuthenticationFailed - Authentication failed for one of the following reasons:</p> <ul style="list-style-type: none"> • The subject name of the signing certificate is not authorized • A matching trusted authority policy was not found for the authorized subject name • The certificate chain is not valid • The signing certificate is not valid • Policy is not configured on the tenant • Thumbprint of the signing certificate is not authorized • Client assertion contains an invalid signature
AADSTS50013	InvalidAssertion - Assertion is invalid because of various reasons - The token issuer doesn't match the api version within its valid time range -expired -malformed - Refresh token in the assertion is not a primary refresh token.
AADSTS50014	GuestUserInPendingState - The user's redemption is in a pending state. The guest user account is not fully created yet.
AADSTS50015	ViralUserLegalAgeConsentRequiredState - The user requires legal age group consent.
AADSTS50017	<p>CertificateValidationFailed - Certification validation failed, reasons for the following reasons:</p> <ul style="list-style-type: none"> • Cannot find issuing certificate in trusted certificates list • Unable to find expected CrlSegment • Cannot find issuing certificate in trusted certificates list • Delta CRL distribution point is configured without a corresponding CRL distribution point • Unable to retrieve valid CRL segments because of a timeout issue • Unable to download CRL <p>Contact the tenant admin.</p>
AADSTS50020	UserUnauthorized - Users are unauthorized to call this endpoint.

ERROR	DESCRIPTION
AADSTS50027	<p>InvalidJwtToken - Invalid JWT token because of the following reasons:</p> <ul style="list-style-type: none"> • doesn't contain nonce claim, sub claim • subject identifier mismatch • duplicate claim in idToken claims • unexpected issuer • unexpected audience • not within its valid time range • token format is not proper • External ID token from issuer failed signature verification.
AADSTS50029	<p>Invalid URI - domain name contains invalid characters. Contact the tenant admin.</p>
AADSTS50032	<p>WeakRsaKey - Indicates the erroneous user attempt to use a weak RSA key.</p>
AADSTS50033	<p>RetryableError - Indicates a transient error not related to the database operations.</p>
AADSTS50034	<p>UserAccountNotFound - To sign into this application, the account must be added to the directory.</p>
AADSTS50042	<p>UnableToGeneratePairwiseIdentifierWithMissingSalt - The salt required to generate a pairwise identifier is missing in principle. Contact the tenant admin.</p>
AADSTS50043	<p>UnableToGeneratePairwiseIdentifierWithMultipleSalts</p>
AADSTS50048	<p>SubjectMismatchedIssuer - Subject mismatches Issuer claim in the client assertion. Contact the tenant admin.</p>
AADSTS50049	<p>NoSuchInstanceForDiscovery - Unknown or invalid instance.</p>
AADSTS50050	<p>MalformedDiscoveryRequest - The request is malformed.</p>
AADSTS50053	<p>IdsLocked - The account is locked because the user tried to sign in too many times with an incorrect user ID or password.</p>
AADSTS50055	<p>InvalidPasswordExpiredPassword - The password is expired.</p>
AADSTS50056	<p>Invalid or null password - Password does not exist in store for this user.</p>
AADSTS50057	<p>UserDisabled - The user account is disabled. The account has been disabled by an administrator.</p>

ERROR	DESCRIPTION
AADSTS50058	<p>UserInformationNotProvided - This means that a user is not signed in. This is a common error that's expected when a user is unauthenticated and has not yet signed in.</p> <p>If this error is encouraged in an SSO context where the user has previously signed in, this means that the SSO session was either not found or invalid.</p> <p>This error may be returned to the application if prompt=none is specified.</p>
AADSTS50059	<p>MissingTenantRealmAndNoUserInformationProvided - Tenant-identifying information was not found in either the request or implied by any provided credentials. The user can contact the tenant admin to help resolve the issue.</p>
AADSTS50061	<p>SignoutInvalidRequest - The sign-out request is invalid.</p>
AADSTS50064	<p>CredentialAuthenticationError - Credential validation on username or password has failed.</p>
AADSTS50068	<p>SignoutInitiatorNotParticipant - Signout has failed. The app that initiated signout is not a participant in the current session.</p>
AADSTS50070	<p>SignoutUnknownSessionIdentifier - Signout has failed. The signout request specified a name identifier that didn't match the existing session(s).</p>
AADSTS50071	<p>SignoutMessageExpired - The logout request has expired.</p>
AADSTS50072	<p>UserStrongAuthEnrollmentRequiredInterrupt - User needs to enroll for second factor authentication (interactive).</p>
AADSTS50074	<p>UserStrongAuthClientAuthNRequiredInterrupt - Strong authentication is required and the user did not pass the MFA challenge.</p>
AADSTS50076	<p>UserStrongAuthClientAuthNRequired - Due to a configuration change made by the admin, or because you moved to a new location, the user must use multi-factor authentication to access the resource. Retry with a new authorize request for the resource.</p>
AADSTS50079	<p>UserStrongAuthEnrollmentRequired - Due to a configuration change made by the administrator, or because the user moved to a new location, the user is required to use multi-factor authentication.</p>
AADSTS50085	<p>Refresh token needs social IDP login. Have user try signing-in again with username -password</p>
AADSTS50086	<p>SasNonRetryableError</p>
AADSTS50087	<p>SasRetryableError - The service is temporarily unavailable. Try again.</p>

ERROR	DESCRIPTION
AADSTS50089	Flow token expired - Authentication Failed. Have the user try signing-in again with username -password.
AADSTS50097	DeviceAuthenticationRequired - Device authentication is required.
AADSTS50099	PKeyAuthInvalidJwtUnauthorized - The JWT signature is invalid.
AADSTS50105	EntitlementGrantsNotFound - The signed in user is not assigned to a role for the signed in app. Assign the user to the app. For more information: https://docs.microsoft.com/azure/active-directory/application-sign-in-problem-federated-sso-gallery#user-not-assigned-a-role .
AADSTS50107	InvalidRealmUri - The requested federation realm object does not exist. Contact the tenant admin.
AADSTS50120	ThresholdJwtInvalidJwtFormat - Issue with JWT header. Contact the tenant admin.
AADSTS50124	ClaimsTransformationInvalidInputParameter - Claims Transformation contains invalid input parameter. Contact the tenant admin to update the policy.
AADSTS50125	PasswordResetRegistrationRequiredInterrupt - Sign-in was interrupted because of a password reset or password registration entry.
AADSTS50126	InvalidUserNameOrPassword - Error validating credentials due to invalid username or password.
AADSTS50127	BrokerAppNotInstalled - User needs to install a broker app to gain access to this content.
AADSTS50128	Invalid domain name - No tenant-identifying information found in either the request or implied by any provided credentials.
AADSTS50129	DeviceIsNotWorkplaceJoined - Workplace join is required to register the device.
AADSTS50131	ConditionalAccessFailed - Indicates various Conditional Access errors such as bad Windows device state, request blocked due to suspicious activity, access policy, or security policy decisions.
AADSTS50132	SsoArtifactInvalidOrExpired - The session is not valid due to password expiration or recent password change.
AADSTS50133	SsoArtifactRevoked - The session is not valid due to password expiration or recent password change.

ERROR	DESCRIPTION
AADSTS50134	DeviceFlowAuthorizeWrongDatacenter - Wrong data center. To authorize a request that was initiated by an app in the OAuth 2.0 device flow, the authorizing party must be in the same data center where the original request resides.
AADSTS50135	PasswordChangeCompromisedPassword - Password change is required due to account risk.
AADSTS50136	RedirectMsaSessionToApp - Single MSA session detected.
AADSTS50139	SessionMissingMsaOAuth2RefreshToken - The session is invalid due to a missing external refresh token.
AADSTS50140	KmslInterrupt - This error occurred due to "Keep me signed in" interrupt when the user was signing-in. Open a support ticket with Correlation ID, Request ID, and Error code to get more details.
AADSTS50143	Session mismatch - Session is invalid because user tenant does not match the domain hint due to different resource. Open a support ticket with Correlation ID, Request ID, and Error code to get more details.
AADSTS50144	InvalidPasswordExpiredOnPremPassword - User's Active Directory password has expired. Generate a new password for the user or have the user use the self-service reset tool to reset their password.
AADSTS50146	MissingCustomSigningKey - This app is required to be configured with an app-specific signing key. It is either not configured with one, or the key has expired or is not yet valid.
AADSTS50147	MissingCodeChallenge - The size of the code challenge parameter is not valid.
AADSTS50155	DeviceAuthenticationFailed - Device authentication failed for this user.
AADSTS50158	ExternalSecurityChallenge - External security challenge was not satisfied.
AADSTS50161	InvalidExternalSecurityChallengeConfiguration - Claims sent by external provider is not enough or Missing claim requested to external provider.
AADSTS50166	ExternalClaimsProviderThrottled - Failed to send the request to the claims provider.
AADSTS50168	ChromeBrowserSsoInterruptRequired - The client is capable of obtaining an SSO token through the Windows 10 Accounts extension, but the token was not found in the request or the supplied token was expired.
AADSTS50169	InvalidRequestBadRealm - The realm is not a configured realm of the current service namespace.

ERROR	DESCRIPTION
AADSTS50170	MissingExternalClaimsProviderMapping - The external controls mapping is missing.
AADSTS50177	ExternalChallengeNotSupportedForPassthroughUsers - External challenge is not supported for passthrough users.
AADSTS50178	SessionControlNotSupportedForPassthroughUsers - Session control is not supported for passthrough users.
AADSTS50180	WindowsIntegratedAuthMissing - Integrated Windows authentication is needed. Enable the tenant for Seamless SSO.
AADSTS50187	DeviceInformationNotProvided - The service failed to perform device authentication.
AADSTS51000	RequiredFeatureNotEnabled - The feature is disabled.
AADSTS51001	DomainHintMustbePresent - Domain hint must be present with on-premises security identifier or on-premises UPN.
AADSTS51004	UserAccountNotInDirectory - The user account doesn't exist in the directory.
AADSTS51005	TemporaryRedirect - Equivalent to HTTP status 307, which indicates that the requested information is located at the URI specified in the location header. When you receive this status, follow the location header associated with the response. When the original request method was POST, the redirected request will also use the POST method.
AADSTS51006	ForceReauthDueToInsufficientAuth - Integrated Windows authentication is needed. User logged in using a session token that is missing the Integrated Windows authentication claim. Request the user to log in again.
AADSTS52004	DelegationDoesNotExistForLinkedIn - The user has not provided consent for access to LinkedIn resources.
AADSTS53000	DeviceNotCompliant - Conditional Access policy requires a compliant device, and the device is not compliant. The user must enroll their device with an approved MDM provider like Intune.
AADSTS53001	DeviceNotDomainJoined - Conditional Access policy requires a domain joined device, and the device is not domain joined. Have the user use a domain joined device.
AADSTS53002	ApplicationUsedIsNotAnApprovedApp - The app used is not an approved app for Conditional Access. User needs to use one of the apps from the list of approved apps to use in order to get access.
AADSTS53003	BlockedByConditionalAccess - Access has been blocked by Conditional Access policies. The access policy does not allow token issuance.

ERROR	DESCRIPTION
AADSTS53004	ProofUpBlockedDueToRisk - User needs to complete the multi-factor authentication registration process before accessing this content. User should register for multi-factor authentication.
AADSTS54000	MinorUserBlockedLegalAgeGroupRule
AADSTS65001	DelegationDoesNotExist - The user or administrator has not consented to use the application with ID X. Send an interactive authorization request for this user and resource.
AADSTS65004	UserDeclinedConsent - User declined to consent to access the app. Have the user retry the sign-in and consent to the app
AADSTS65005	MisconfiguredApplication - The app required resource access list does not contain apps discoverable by the resource or The client app has requested access to resource, which was not specified in its required resource access list or Graph service returned bad request or resource not found. If the app supports SAML, you may have configured the app with the wrong Identifier (Entity). Try out the resolution listed for SAML using the link below: https://docs.microsoft.com/azure/active-directory/application-sign-in-problem-federated-sso-gallery#no-resource-in-requiredresourceaccess-list
AADSTS67003	ActorNotValidServiceIdentity
AADSTS70000	InvalidGrant - Authentication failed. The refresh token is not valid. Error may be due to the following reasons: <ul style="list-style-type: none"> Token binding header is empty Token binding hash does not match
AADSTS70001	UnauthorizedClient - The application is disabled.
AADSTS70002	InvalidClient - Error validating the credentials. The specified client_secret does not match the expected value for this client. Correct the client_secret and try again. For more info, see Use the authorization code to request an access token .
AADSTS70003	UnsupportedGrantType - The app returned an unsupported grant type.
AADSTS70004	InvalidRedirectUri - The app returned an invalid redirect URI. The redirect address specified by the client does not match any configured addresses or any addresses on the OIDC approve list.
AADSTS70005	UnsupportedResponseType - The app returned an unsupported response type due to the following reasons: <ul style="list-style-type: none"> response type 'token' is not enabled for the app response type 'id_token' requires the 'OpenID' scope - contains an unsupported OAuth parameter value in the encoded wctx

ERROR	DESCRIPTION
AADSTS70007	UnsupportedResponseMode - The app returned an unsupported value of <code>response_mode</code> when requesting a token.
AADSTS70008	ExpiredOrRevokedGrant - The refresh token has expired due to inactivity. The token was issued on XXX and was inactive for a certain amount of time.
AADSTS70011	InvalidScope - The scope requested by the app is invalid.
AADSTS70012	MsaServerError - A server error occurred while authenticating an MSA (consumer) user. Try again. If it continues to fail, open a support ticket
AADSTS70016	AuthorizationPending - OAuth 2.0 device flow error. Authorization is pending. The device will retry polling the request.
AADSTS70018	BadVerificationCode - Invalid verification code due to User typing in wrong user code for device code flow. Authorization is not approved.
AADSTS70019	CodeExpired - Verification code expired. Have the user retry the sign-in.
AADSTS75001	BindingSerializationError - An error occurred during SAML message binding.
AADSTS75003	UnsupportedBindingError - The app returned an error related to unsupported binding (SAML protocol response cannot be sent via bindings other than HTTP POST).
AADSTS75005	Saml2MessageInvalid - Azure AD doesn't support the SAML request sent by the app for SSO.
AADSTS75008	RequestDeniedError - The request from the app was denied since the SAML request had an unexpected destination.
AADSTS75011	NoMatchedAuthnContextInOutputClaims - The authentication method by which the user authenticated with the service doesn't match requested authentication method.
AADSTS75016	Saml2AuthenticationRequestInvalidNameIDPolicy - SAML2 Authentication Request has invalid NameIdPolicy.
AADSTS80001	OnPremiseStoreIsNotAvailable - The Authentication Agent is unable to connect to Active Directory. Make sure that agent servers are members of the same AD forest as the users whose passwords need to be validated and they are able to connect to Active Directory.
AADSTS80002	OnPremisePasswordValidatorRequestTimedout - Password validation request timed out. Make sure that Active Directory is available and responding to requests from the agents.

ERROR	DESCRIPTION
AADSTS80005	OnPremisePasswordValidatorUnpredictableWebException - An unknown error occurred while processing the response from the Authentication Agent. Retry the request. If it continues to fail, open a support ticket to get more details on the error.
AADSTS80007	OnPremisePasswordValidatorErrorOccurredOnPrem - The Authentication Agent is unable to validate user's password. Check the agent logs for more info and verify that Active Directory is operating as expected.
AADSTS80010	OnPremisePasswordValidationEncryptionException - The Authentication Agent is unable to decrypt password.
AADSTS80012	OnPremisePasswordValidationAccountLogonInvalidHours - The users attempted to log on outside of the allowed hours (this is specified in AD).
AADSTS80013	OnPremisePasswordValidationTimeSkew - The authentication attempt could not be completed due to time skew between the machine running the authentication agent and AD. Fix time sync issues.
AADSTS81004	DesktopSsoIdentityInTicketIsNotAuthenticated - Kerberos authentication attempt failed.
AADSTS81005	DesktopSsoAuthenticationPackageNotSupported - The authentication package is not supported.
AADSTS81006	DesktopSsoNoAuthorizationHeader - No authorization header was found.
AADSTS81007	DesktopSsoTenantIsNotOptIn - The tenant is not enabled for Seamless SSO.
AADSTS81009	DesktopSsoAuthorizationHeaderValueWithBadFormat - Unable to validate user's Kerberos ticket.
AADSTS81010	DesktopSsoAuthTokenInvalid - Seamless SSO failed because the user's Kerberos ticket has expired or is invalid.
AADSTS81011	DesktopSsoLookupUserBySidFailed - Unable to find user object based on information in the user's Kerberos ticket.
AADSTS81012	DesktopSsoMismatchBetweenTokenUpnAndChosenUpn - The user trying to sign in to Azure AD is different from the user signed into the device.
AADSTS90002	InvalidTenantName - The tenant name wasn't found in the data store. Check to make sure you have the correct tenant ID.
AADSTS90004	InvalidRequestFormat - The request is not properly formatted.

ERROR	DESCRIPTION
AADSTS90005	InvalidRequestWithMultipleRequirements - Unable to complete the request. The request is not valid because the identifier and login hint can't be used together.
AADSTS90006	ExternalServerRetryableError - The service is temporarily unavailable.
AADSTS90007	InvalidSessionId - Bad request. The passed session ID can't be parsed.
AADSTS90008	TokenForItselfRequiresGraphPermission - The user or administrator hasn't consented to use the application. At the minimum, the application requires access to Azure AD by specifying the sign-in and read user profile permission.
AADSTS90009	TokenForItselfMissingIdenticalAppIdentifier - The application is requesting a token for itself. This scenario is supported only if the resource that's specified is using the GUID-based application ID.
AADSTS90010	NotSupported - Unable to create the algorithm.
AADSTS90012	RequestTimeout - The requested has timed out.
AADSTS90013	InvalidUserInput - The input from the user is not valid.
AADSTS90014	MissingRequiredField - This error code may appear in various cases when an expected field is not present in the credential.
AADSTS90015	QueryStringTooLong - The query string is too long.
AADSTS90016	MissingRequiredClaim - The access token isn't valid. The required claim is missing.
AADSTS90019	MissingTenantRealm - Azure AD was unable to determine the tenant identifier from the request.
AADSTS90022	AuthenticatedInvalidPrincipalNameFormat - The principal name format is not valid, or does not meet the expected <code>name[/host][@realm]</code> format. The principal name is required, host and realm are optional and may be set to null.
AADSTS90023	InvalidRequest - The authentication service request is not valid.
AADSTS9002313	InvalidRequest - Request is malformed or invalid. - The issue here is because there was something wrong with the request to a certain endpoint. The suggestion to this issue is to get a fiddler trace of the error occurring and looking to see if the request is actually properly formatted or not.
AADSTS90024	RequestBudgetExceededError - A transient error has occurred. Try again.

ERROR	DESCRIPTION
AADSTS90033	MsodsServiceUnavailable - The Microsoft Online Directory Service (MSODS) is not available.
AADSTS90036	MsodsServiceUnretryableFailure - An unexpected, non-retryable error from the WCF service hosted by MSODS has occurred. Open a support ticket to get more details on the error.
AADSTS90038	NationalCloudTenantRedirection - The specified tenant 'Y' belongs to the National Cloud 'X'. Current cloud instance 'Z' does not federate with X. A cloud redirect error is returned.
AADSTS90043	NationalCloudAuthCodeRedirection - The feature is disabled.
AADSTS90051	InvalidNationalCloudId - The national cloud identifier contains an invalid cloud identifier.
AADSTS90055	TenantThrottlingError - There are too many incoming requests. This exception is thrown for blocked tenants.
AADSTS90056	<p>BadResourceRequest - To redeem the code for an access token, the app should send a POST request to the <code>/token</code> endpoint. Also, prior to this, you should provide an authorization code and send it in the POST request to the <code>/token</code> endpoint. Refer to this article for an overview of OAuth 2.0 authorization code flow: https://docs.microsoft.com/azure/active-directory/develop/active-directory-protocols-oauth-code. Direct the user to the <code>/authorize</code> endpoint, which will return an authorization_code. By posting a request to the <code>/token</code> endpoint, the user gets the access token. Log in the Azure portal, and check App registrations > Endpoints to confirm that the two endpoints were configured correctly.</p>
AADSTS90072	PassThroughUserMfaError - The external account that the user signs in with doesn't exist on the tenant that they signed into; so the user can't satisfy the MFA requirements for the tenant. The account must be added as an external user in the tenant first. Sign out and sign in with a different Azure AD user account.
AADSTS90081	OrgIdWsFederationMessageInvalid - An error occurred when the service tried to process a WS-Federation message. The message is not valid.
AADSTS90082	OrgIdWsFederationNotSupported - The selected authentication policy for the request isn't currently supported.
AADSTS90084	OrgIdWsFederationGuestNotAllowed - Guest accounts aren't allowed for this site.
AADSTS90085	OrgIdWsFederationSltRedemptionFailed - The service is unable to issue a token because the company object hasn't been provisioned yet.

ERROR	DESCRIPTION
AADSTS90086	OrgIdWsTrustDaTokenExpired - The user DA token is expired.
AADSTS90087	OrgIdWsFederationMessageCreationFromUriFailed - An error occurred while creating the WS-Federation message from the URI.
AADSTS90090	GraphRetryableError - The service is temporarily unavailable.
AADSTS90091	GraphServiceUnreachable
AADSTS90092	GraphNonRetryableError
AADSTS90093	GraphUserUnauthorized - Graph returned with a forbidden error code for the request.
AADSTS90094	AdminConsentRequired - Administrator consent is required.
AADSTS90100	InvalidRequestParameter - The parameter is empty or not valid.
AADSTS90102	AADSTS90102: The 'resource' request parameter is not supported.
AADSTS90101	InvalidEmailAddress - The supplied data isn't a valid email address. The email address must be in the format <code>someone@example.com</code>
AADSTS90102	InvalidUriParameter - The value must be a valid absolute URI.
AADSTS90107	InvalidXml - The request is not valid. Make sure your data doesn't have invalid characters.
AADSTS90114	InvalidExpiryDate - The bulk token expiration timestamp will cause an expired token to be issued.
AADSTS90117	InvalidRequestInput
AADSTS90119	InvalidUserCode - The user code is null or empty.
AADSTS90120	InvalidDeviceFlowRequest - The request was already authorized or declined.
AADSTS90121	InvalidEmptyRequest - Invalid empty request.
AADSTS90123	IdentityProviderAccessDenied - The token can't be issued because the identity or claim issuance provider denied the request.
AADSTS90124	V1ResourceV2GlobalEndpointNotSupported - The resource is not supported over the <code>/common</code> or <code>/consumers</code> endpoints. Use the <code>/organizations</code> or tenant-specific endpoint instead.

ERROR	DESCRIPTION
AADSTS90125	DebugModeEnrollTenantNotFound - The user isn't in the system. Make sure you entered the user name correctly.
AADSTS90126	DebugModeEnrollTenantNotInferred - The user type is not supported on this endpoint. The system can't infer the user's tenant from the user name.
AADSTS90130	NonConvergedAppV2GlobalEndpointNotSupported - The application is not supported over the <code>/common</code> or <code>/consumers</code> endpoints. Use the <code>/organizations</code> or tenant-specific endpoint instead.
AADSTS120000	PasswordChangeIncorrectCurrentPassword
AADSTS120002	PasswordChangeInvalidNewPasswordWeak
AADSTS120003	PasswordChangeInvalidNewPasswordContainsMemberName
AADSTS120004	PasswordChangeOnPremComplexity
AADSTS120005	PasswordChangeOnPremSuccessCloudFail
AADSTS120008	PasswordChangeAsyncJobStateTerminated - A non-retryable error has occurred.
AADSTS120011	PasswordChangeAsyncUpnInferenceFailed
AADSTS120012	PasswordChangeNeedsToHappenOnPrem
AADSTS120013	PasswordChangeOnPremisesConnectivityFailure
AADSTS120014	PasswordChangeOnPremUserAccountLockedOutOrDisabled
AADSTS120015	PasswordChangeADAdminActionRequired
AADSTS120016	PasswordChangeUserNotFoundBySspr
AADSTS120018	PasswordChangePasswordDoesnotComplyFuzzyPolicy
AADSTS120020	PasswordChangeFailure
AADSTS120021	PartnerServiceSsprInternalServiceError
AADSTS130004	NgcKeyNotFound - The user principal doesn't have the NGC ID key configured.
AADSTS130005	NgcInvalidSignature - NGC key signature verified failed.
AADSTS130006	NgcTransportKeyNotFound - The NGC transport key isn't configured on the device.
AADSTS130007	NgcDeviceIsDisabled - The device is disabled.

ERROR	DESCRIPTION
AADSTS130008	NgcDeviceNotFound - The device referenced by the NGC key wasn't found.
AADSTS135010	KeyNotFound
AADSTS140000	InvalidRequestNonce - Request nonce is not provided.
AADSTS140001	InvalidSessionKey - The session key is not valid.
AADSTS165900	InvalidApiRequest - Invalid request.
AADSTS220450	UnsupportedAndroidWebViewVersion - The Chrome WebView version is not supported.
AADSTS220501	InvalidCrlDownload
AADSTS221000	DeviceOnlyTokensNotSupportedByResource - The resource is not configured to accept device-only tokens.
AADSTS240001	BulkAADJTokenUnauthorized - The user isn't authorized to register devices in Azure AD.
AADSTS240002	RequiredClaimIsMissing - The id_token can't be used as <code>urn:ietf:params:oauth:grant-type:jwt-bearer</code> grant.
AADSTS530032	BlockedByConditionalAccessOnSecurityPolicy - The tenant admin has configured a security policy that blocks this request. Check the security policies that are defined on the tenant level to determine if your request meets the policy requirements.
AADSTS700016	UnauthorizedClient_DoesNotMatchRequest - The application wasn't found in the directory/tenant. 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 misconfigured the identifier value for the application or sent your authentication request to the wrong tenant.
AADSTS700020	InteractionRequired - The access grant requires interaction.
AADSTS700022	InvalidMultipleResourcesScope - The provided value for the input parameter scope isn't valid because it contains more than one resource.
AADSTS700023	InvalidResourcelessScope - The provided value for the input parameter scope isn't valid when request an access token.
AADSTS1000000	UserNotBoundError - The Bind API requires the Azure AD user to also authenticate with an external IDP, which hasn't happened yet.
AADSTS1000002	BindCompleteInterruptError - The bind completed successfully, but the user must be informed.

ERROR	DESCRIPTION
AADSTS7000112	UnauthorizedClientApplicationDisabled - The application is disabled.

Next steps

- Have a question or can't find what you're looking for? Create a GitHub issue or see [Support and help options for developers](#) to learn about other ways you can get help and support.

What's new for authentication?

8/28/2019 • 7 minutes to read • [Edit Online](#)

Get notified about updates to this page. Just add [this URL](#) to your RSS feed reader.

The authentication system alters and adds features on an ongoing basis to improve security and standards compliance. To stay up-to-date with the most recent developments, this article provides you with information about the following details:

- Latest features
- Known issues
- Protocol changes
- Deprecated functionality

TIP

This page is updated regularly, so visit often. Unless otherwise noted, these changes are only put in place for newly registered applications.

Upcoming changes

September 2019: Additional enforcement of POST semantics according to URL parsing rules - duplicate parameters will trigger an error and **BOM** ignored.

August 2019

POST form semantics will be enforced more strictly - spaces and quotes will be ignored

Effective date: September 2, 2019

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: Anywhere POST is used ([client credentials](#), [authorization code redemption](#), [ROPC](#), [OBO](#), and [refresh token redemption](#))

Starting the week of 9/2, authentication requests that use the POST method will be validated using stricter HTTP standards. Specifically, spaces and double-quotes (") will no longer be removed from request form values. These changes are not expected to break any existing clients, and will ensure that requests sent to Azure AD are reliably handled every time. In the future (see above) we plan to additionally reject duplicate parameters and ignore the BOM within requests.

Example:

Today, `?e= "f"&g=h` is parsed identically as `?e=f&g=h` - so `e == f`. With this change, it would now be parsed so that `e == "f"` - this is unlikely to be a valid argument, and the request would now fail.

July 2019

App-only tokens for single-tenant applications are only issued if the client app exists in the resource tenant

Effective date: July 26, 2019

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: Client Credentials (app-only tokens)

A security change went live July 26th that changes the way app-only tokens (via the client credentials grant) are issued. Previously, applications were allowed to get tokens to call any other app, regardless of presence in the tenant or roles consented to for that application. This behavior has been updated so that for resources (sometimes called Web APIs) set to be single-tenant (the default), the client application must exist within the resource tenant. Note that existing consent between the client and the API is still not required, and apps should still be doing their own authorization checks to ensure that a `roles` claim is present and contains the expected value for the API.

The error message for this scenario currently states:

```
The service principal named <appName> was not found in the tenant named <tenant_name>. This can happen if the application has not been installed by the administrator of the tenant.
```

To remedy this issue, use the Admin Consent experience to create the client application service principal in your tenant, or create it manually. This requirement ensures that the tenant has given the application permission to operate within the tenant.

Example request

```
https://login.microsoftonline.com/contoso.com/oauth2/authorize?  
resource=https://gateway.contoso.com/api&response_type=token&client_id=14c88eee-b3e2-4bb0-9233-f5e3053b3a28&...
```

In this example, the resource tenant (authority) is contoso.com, the resource app is a single-tenant app called `gateway.contoso.com/api` for the Contoso tenant, and the client app is `14c88eee-b3e2-4bb0-9233-f5e3053b3a28`. If the client app has a service principal within Contoso.com, this request can continue. If it doesn't, however, then the request will fail with the error above.

If the Contoso gateway app were a multi-tenant application, however, then the request would continue regardless of the client app having a service principal within Contoso.com.

Redirect URIs can now contain query string parameters

Effective date: July 22, 2019

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: All flows

Per [RFC 6749](#), Azure AD applications can now register and use redirect (reply) URIs with static query parameters (such as <https://contoso.com/oauth2?idp=microsoft>) for OAuth 2.0 requests. Dynamic redirect URIs are still forbidden as they represent a security risk, and this cannot be used to retain state information across an authentication request - for that, use the `state` parameter.

The static query parameter is subject to string matching for redirect URIs like any other part of the redirect URI - if no string is registered that matches the URI-decoded `redirect_uri`, then the request will be rejected. If the URI is found in the app registration, then the entire string will be used to redirect the user, including the static query parameter.

Note that at this time (End of July 2019), the app registration UX in Azure portal still block query parameters. However, you can edit the application manifest manually to add query parameters and test this in your app.

March 2019

Looping clients will be interrupted

Effective date: March 25, 2019

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: All flows

Client applications can sometimes misbehave, issuing hundreds of the same login request over a short period of time. These requests may or may not be successful, but they all contribute to poor user experience and heightened workloads for the IDP, increasing latency for all users and reducing availability of the IDP. These applications are operating outside the bounds of normal usage, and should be updated to behave correctly.

Clients that issue duplicate requests multiple times will be sent an `invalid_grant` error:

AADSTS50196: The server terminated an operation because it encountered a loop while processing a request.

Most clients will not need to change behavior to avoid this error. Only misconfigured clients (those without token caching or those exhibiting prompt loops already) will be impacted by this error. Clients are tracked on a per-instance basis locally (via cookie) on the following factors:

- User hint, if any
- Scopes or resource being requested
- Client ID
- Redirect URI
- Response type and mode

Apps making multiple requests (15+) in a short period of time (5 minutes) will receive an `invalid_grant` error explaining that they are looping. The tokens being requested have sufficiently long-lived lifetimes (10 minutes minimum, 60 minutes by default), so repeated requests over this time period are unnecessary.

All apps should handle `invalid_grant` by showing an interactive prompt, rather than silently requesting a token. In order to avoid this error, clients should ensure they are correctly caching the tokens they receive.

October 2018

Authorization codes can no longer be reused

Effective date: November 15, 2018

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: [Code flow](#)

Starting on November 15, 2018, Azure AD will stop accepting previously used authentication codes for apps. This security change helps to bring Azure AD in line with the OAuth specification and will be enforced on both the v1 and v2 endpoints.

If your app reuses authorization codes to get tokens for multiple resources, we recommend that you use the code to get a refresh token, and then use that refresh token to acquire additional tokens for other resources.

Authorization codes can only be used once, but refresh tokens can be used multiple times across multiple resources. Any new app that attempts to reuse an authentication code during the OAuth code flow will get an `invalid_grant` error.

For more information about refresh tokens, see [Refreshing the access tokens](#). If using ADAL or MSAL, this is handled for you by the library - replace the second instance of 'AcquireTokenByAuthorizationCodeAsync' with 'AcquireTokenSilentAsync'.

May 2018

ID tokens cannot be used for the OBO flow

Date: May 1, 2018

Endpoints impacted: Both v1.0 and v2.0

Protocols impacted: Implicit flow and [OBO flow](#)

After May 1, 2018, id_tokens cannot be used as the assertion in an OBO flow for new applications. Access tokens should be used instead to secure APIs, even between a client and middle tier of the same application. Apps registered before May 1, 2018 will continue to work and be able to exchange id_tokens for an access token; however, this pattern is not considered a best practice.

To work around this change, you can do the following:

1. Create a Web API for your application, with one or more scopes. This explicit entry point will allow finer grained control and security.
2. In your app's manifest, in the [Azure portal](#) or the [app registration portal](#), ensure that the app is allowed to issue access tokens via the implicit flow. This is controlled through the `oauth2AllowImplicitFlow` key.
3. When your client application requests an id_token via `response_type=id_token`, also request an access token (`response_type=token`) for the Web API created above. Thus, when using the v2.0 endpoint the `scope` parameter should look similar to `api://GUID/SCOPE`. On the v1.0 endpoint, the `resource` parameter should be the app URI of the web API.
4. Pass this access token to the middle tier in place of the id_token.

Microsoft identity platform developer glossary

8/6/2019 • 14 minutes to read • [Edit Online](#)

This article contains definitions for some of the core developer concepts and terminology, which are helpful when learning about application development using Microsoft identity platform.

access token

A type of [security token](#) issued by an [authorization server](#), and used by a [client application](#) in order to access a [protected resource server](#). Typically in the form of a [JSON Web Token \(JWT\)](#), the token embodies the authorization granted to the client by the [resource owner](#), for a requested level of access. The token contains all applicable [claims](#) about the subject, enabling the client application to use it as a form of credential when accessing a given resource. This also eliminates the need for the resource owner to expose credentials to the client.

Access tokens are sometimes referred to as "User+App" or "App-Only", depending on the credentials being represented. For example, when a client application uses the:

- ["Authorization code" authorization grant](#), the end user authenticates first as the resource owner, delegating authorization to the client to access the resource. The client authenticates afterward when obtaining the access token. The token can sometimes be referred to more specifically as a "User+App" token, as it represents both the user that authorized the client application, and the application.
- ["Client credentials" authorization grant](#), the client provides the sole authentication, functioning without the resource-owner's authentication/authorization, so the token can sometimes be referred to as an "App-Only" token.

See [Microsoft identity platform Token Reference](#) for more details.

application ID (client ID)

The unique identifier Azure AD issues to an application registration that identifies a specific application and the associated configurations. This application ID ([client ID](#)) is used when performing authentication requests and is provided to the authentication libraries in development time. The application ID (client ID) is not a secret.

application manifest

A feature provided by the [Azure portal](#), which produces a JSON representation of the application's identity configuration, used as a mechanism for updating its associated [Application](#) and [ServicePrincipal](#) entities. See [Understanding the Azure Active Directory application manifest](#) for more details.

application object

When you register/update an application in the [Azure portal](#), the portal creates/updates both an application object and a corresponding [service principal object](#) for that tenant. The application object *defines* the application's identity configuration globally (across all tenants where it has access), providing a template from which its corresponding service principal object(s) are *derived* for use locally at run-time (in a specific tenant).

For more information, see [Application and Service Principal Objects](#).

application registration

In order to allow an application to integrate with and delegate Identity and Access Management functions to

Azure AD, it must be registered with an Azure AD [tenant](#). When you register your application with Azure AD, you are providing an identity configuration for your application, allowing it to integrate with Azure AD and use features such as:

- Robust management of Single Sign-On using Azure AD Identity Management and [OpenID Connect](#) protocol implementation
- Brokered access to [protected resources](#) by [client applications](#), via OAuth 2.0 [authorization server](#)
- [Consent framework](#) for managing client access to protected resources, based on resource owner authorization.

See [Integrating applications with Azure Active Directory](#) for more details.

authentication

The act of challenging a party for legitimate credentials, providing the basis for creation of a security principal to be used for identity and access control. During an [OAuth2 authorization grant](#) for example, the party authenticating is filling the role of either [resource owner](#) or [client application](#), depending on the grant used.

authorization

The act of granting an authenticated security principal permission to do something. There are two primary use cases in the Azure AD programming model:

- During an [OAuth2 authorization grant](#) flow: when the [resource owner](#) grants authorization to the [client application](#), allowing the client to access the resource owner's resources.
- During resource access by the client: as implemented by the [resource server](#), using the [claim](#) values present in the [access token](#) to make access control decisions based upon them.

authorization code

A short lived "token" provided to a [client application](#) by the [authorization endpoint](#), as part of the "authorization code" flow, one of the four OAuth2 [authorization grants](#). The code is returned to the client application in response to authentication of a [resource owner](#), indicating the resource owner has delegated authorization to access the requested resources. As part of the flow, the code is later redeemed for an [access token](#).

authorization endpoint

One of the endpoints implemented by the [authorization server](#), used to interact with the [resource owner](#) in order to provide an [authorization grant](#) during an OAuth2 authorization grant flow. Depending on the authorization grant flow used, the actual grant provided can vary, including an [authorization code](#) or [security token](#).

See the OAuth2 specification's [authorization grant types](#) and [authorization endpoint](#) sections, and the [OpenIDConnect specification](#) for more details.

authorization grant

A credential representing the [resource owner's authorization](#) to access its protected resources, granted to a [client application](#). A client application can use one of the [four grant types defined by the OAuth2 Authorization Framework](#) to obtain a grant, depending on client type/requirements: "authorization code grant", "client credentials grant", "implicit grant", and "resource owner password credentials grant". The credential returned to the client is either an [access token](#), or an [authorization code](#) (exchanged later for an access token), depending on the type of authorization grant used.

authorization server

As defined by the [OAuth2 Authorization Framework](#), the server responsible for issuing access tokens to the [client](#) after successfully authenticating the [resource owner](#) and obtaining its authorization. A [client application](#) interacts with the authorization server at runtime via its [authorization](#) and [token](#) endpoints, in accordance with the OAuth2 defined [authorization grants](#).

In the case of Microsoft identity platform application integration, Microsoft identity platform implements the authorization server role for Azure AD applications and Microsoft service APIs, for example [Microsoft Graph APIs](#).

claim

A [security token](#) contains claims, which provide assertions about one entity (such as a [client application](#) or [resource owner](#)) to another entity (such as the [resource server](#)). Claims are name/value pairs that relay facts about the token subject (for example, the security principal that was authenticated by the [authorization server](#)). The claims present in a given token are dependent upon several variables, including the type of token, the type of credential used to authenticate the subject, the application configuration, etc.

See [Microsoft identity platform token reference](#) for more details.

client application

As defined by the [OAuth2 Authorization Framework](#), an application that makes protected resource requests on behalf of the [resource owner](#). The term "client" does not imply any particular hardware implementation characteristics (for instance, whether the application executes on a server, a desktop, or other devices).

A client application requests [authorization](#) from a resource owner to participate in an [OAuth2 authorization grant](#) flow, and may access APIs/data on the resource owner's behalf. The OAuth2 Authorization Framework [defines two types of clients](#), "confidential" and "public", based on the client's ability to maintain the confidentiality of its credentials. Applications can implement a [web client \(confidential\)](#) which runs on a web server, a [native client \(public\)](#) installed on a device, or a [user-agent-based client \(public\)](#) which runs in a device's browser.

consent

The process of a [resource owner](#) granting authorization to a [client application](#), to access protected resources under specific [permissions](#), on behalf of the resource owner. Depending on the permissions requested by the client, an administrator or user will be asked for consent to allow access to their organization/individual data respectively. Note, in a [multi-tenant](#) scenario, the application's [service principal](#) is also recorded in the tenant of the consenting user.

See [consent framework](#) for more information.

ID token

An [OpenID Connect security token](#) provided by an [authorization server's authorization endpoint](#), which contains [claims](#) pertaining to the authentication of an end user [resource owner](#). Like an access token, ID tokens are also represented as a digitally signed [JSON Web Token \(JWT\)](#). Unlike an access token though, an ID token's claims are not used for purposes related to resource access and specifically access control.

See [Microsoft identity platform token reference](#) for more details.

Microsoft identity platform

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, libraries, application registration and configuration, 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. See [About Microsoft identity platform](#) for more details.

multi-tenant application

A class of application that enables sign in and [consent](#) by users provisioned in any Azure AD [tenant](#), including tenants other than the one where the client is registered. [Native client](#) applications are multi-tenant by default, whereas [web client](#) and [web resource/API](#) applications have the ability to select between single or multi-tenant. By contrast, a web application registered as single-tenant, would only allow sign-ins from user accounts provisioned in the same tenant as the one where the application is registered.

See [How to sign in any Azure AD user using the multi-tenant application pattern](#) for more details.

native client

A type of [client application](#) that is installed natively on a device. Since all code is executed on a device, it is considered a "public" client due to its inability to store credentials privately/confidentially. See [OAuth2 client types and profiles](#) for more details.

permissions

A [client application](#) gains access to a [resource server](#) by declaring permission requests. Two types are available:

- "Delegated" permissions, which specify [scope-based](#) access using delegated authorization from the signed-in [resource owner](#), are presented to the resource at run-time as "[scp](#)" [claims](#) in the client's [access token](#).
- "Application" permissions, which specify [role-based](#) access using the client application's credentials/identity, are presented to the resource at run-time as "[roles](#)" [claims](#) in the client's access token.

They also surface during the [consent](#) process, giving the administrator or resource owner the opportunity to grant/deny the client access to resources in their tenant.

Permission requests are configured on the [API permissions](#) page for an application in the [Azure portal](#), by selecting the desired "Delegated Permissions" and "Application Permissions" (the latter requires membership in the Global Admin role). Because a [public client](#) can't securely maintain credentials, it can only request delegated permissions, while a [confidential client](#) has the ability to request both delegated and application permissions. The client's [application object](#) stores the declared permissions in its [requiredResourceAccess](#) property.

resource owner

As defined by the [OAuth2 Authorization Framework](#), an entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end user. For example, when a [client application](#) wants to access a user's mailbox through the [Microsoft Graph API](#), it requires permission from the resource owner of the mailbox.

resource server

As defined by the [OAuth2 Authorization Framework](#), a server that hosts protected resources, capable of accepting and responding to protected resource requests by [client applications](#) that present an [access token](#). Also known as a protected resource server, or resource application.

A resource server exposes APIs and enforces access to its protected resources through [scopes](#) and [roles](#), using the OAuth 2.0 Authorization Framework. Examples include the Azure AD Graph API which provides access to Azure AD tenant data, and the Office 365 APIs that provide access to data such as mail and calendar. Both of these are

also accessible via the [Microsoft Graph API](#).

Just like a client application, resource application's identity configuration is established via [registration](#) in an Azure AD tenant, providing both the application and service principal object. Some Microsoft-provided APIs, such as the Azure AD Graph API, have pre-registered service principals made available in all tenants during provisioning.

roles

Like [scopes](#), roles provide a way for a [resource server](#) to govern access to its protected resources. There are two types: a "user" role implements role-based access control for users/groups that require access to the resource, while an "application" role implements the same for [client applications](#) that require access.

Roles are resource-defined strings (for example "Expense approver", "Read-only", "Directory.ReadWrite.All"), managed in the [Azure portal](#) via the resource's [application manifest](#), and stored in the resource's [appRoles property](#). The Azure portal is also used to assign users to "user" roles, and configure client [application permissions](#) to access an "application" role.

For a detailed discussion of the application roles exposed by Azure AD's Graph API, see [Graph API Permission Scopes](#). For a step-by-step implementation example, see [Manage access using RBAC and the Azure portal](#).

scopes

Like [roles](#), scopes provide a way for a [resource server](#) to govern access to its protected resources. Scopes are used to implement [scope-based](#) access control, for a [client application](#) that has been given delegated access to the resource by its owner.

Scopes are resource-defined strings (for example "Mail.Read", "Directory.ReadWrite.All"), managed in the [Azure portal](#) via the resource's [application manifest](#), and stored in the resource's [oauth2Permissions property](#). The Azure portal is also used to configure client application [delegated permissions](#) to access a scope.

A best practice naming convention, is to use a "resource.operation.constraint" format. For a detailed discussion of the scopes exposed by Azure AD's Graph API, see [Graph API Permission Scopes](#). For scopes exposed by Office 365 services, see [Office 365 API permissions reference](#).

security token

A signed document containing claims, such as an OAuth2 token or SAML 2.0 assertion. For an OAuth2 [authorization grant](#), an [access token](#) (OAuth2) and an [ID Token](#) are types of security tokens, both of which are implemented as a [JSON Web Token \(JWT\)](#).

service principal object

When you register/update an application in the [Azure portal](#), the portal creates/updates both an [application object](#) and a corresponding service principal object for that tenant. The application object *defines* the application's identity configuration globally (across all tenants where the associated application has been granted access), and is the template from which its corresponding service principal object(s) are *derived* for use locally at run-time (in a specific tenant).

For more information, see [Application and Service Principal Objects](#).

sign-in

The process of a [client application](#) initiating end-user authentication and capturing related state, for the purpose of acquiring a [security token](#) and scoping the application session to that state. State can include artifacts such as user profile information, and information derived from token claims.

The sign-in function of an application is typically used to implement single-sign-on (SSO). It may also be preceded by a "sign-up" function, as the entry point for an end user to gain access to an application (upon first sign-in). The sign-up function is used to gather and persist additional state specific to the user, and may require [user consent](#).

sign-out

The process of unauthenticating an end user, detaching the user state associated with the [client application](#) session during [sign-in](#)

tenant

An instance of an Azure AD directory is referred to as an Azure AD tenant. It provides several features, including:

- a registry service for integrated applications
- authentication of user accounts and registered applications
- REST endpoints required to support various protocols including OAuth2 and SAML, including the [authorization endpoint](#), [token endpoint](#) and the "common" endpoint used by [multi-tenant applications](#).

Azure AD tenants are created/associated with Azure and Office 365 subscriptions during sign-up, providing Identity & Access Management features for the subscription. Azure subscription administrators can also create additional Azure AD tenants via the Azure portal. See [How to get an Azure Active Directory tenant](#) for details on the various ways you can get access to a tenant. See [How Azure subscriptions are associated with Azure Active Directory](#) for details on the relationship between subscriptions and an Azure AD tenant.

token endpoint

One of the endpoints implemented by the [authorization server](#) to support OAuth2 [authorization grants](#). Depending on the grant, it can be used to acquire an [access token](#) (and related "refresh" token) to a [client](#), or [ID token](#) when used with the [OpenID Connect](#) protocol.

User-agent-based client

A type of [client application](#) that downloads code from a web server and executes within a user-agent (for instance, a web browser), such as a single-page application (SPA). Since all code is executed on a device, it is considered a "public" client due to its inability to store credentials privately/confidentially. For more information, see [OAuth2 client types and profiles](#).

user principal

Similar to the way a service principal object is used to represent an application instance, a user principal object is another type of security principal, which represents a user. The Azure AD Graph [User entity](#) defines the schema for a user object, including user-related properties such as first and last name, user principal name, directory role membership, etc. This provides the user identity configuration for Azure AD to establish a user principal at run-time. The user principal is used to represent an authenticated user for Single Sign-On, recording [consent](#) delegation, making access control decisions, etc.

web client

A type of [client application](#) that executes all code on a web server, and able to function as a "confidential" client by securely storing its credentials on the server. For more information, see [OAuth2 client types and profiles](#).

Next steps

The [Microsoft identity platform Developer's Guide](#) is the landing page to use for all Microsoft identity platform development-related topics, including an overview of [application integration](#) and the basics of [Microsoft identity platform authentication and supported authentication scenarios](#). You can also find code samples & tutorials on how to get up and running quickly on [GitHub](#).

Use the following comments section to provide feedback and help to refine and shape this content, including requests for new definitions or updating existing ones!

Support and help options for developers

10/16/2019 • 2 minutes to read • [Edit Online](#)

If you're just starting to integrate with Azure Active Directory (Azure AD), Microsoft identities, or Microsoft Graph API, or when you're implementing a new feature to your application, there are times when you need to obtain help from the community or understand the support options that you have as a developer. This article helps you to understand these options, including:

- How to search whether your question hasn't been answered by the community, or if an existing documentation for the feature you're trying to implement already exists
- In some cases, you just want to use our support tools to help you debug a specific problem
- If you can't find the answer that you need, you may want to ask a question on *Stack Overflow*
- If you find an issue with one of our authentication libraries, raise a *GitHub* issue
- Finally, if you need to talk to someone, you might want to open a support request

Search

If you have a development-related question, you may be able to find the answer in the documentation, [GitHub samples](#), or answers to [Stack Overflow](#) questions.

Scoped search

For faster results, scope your search to Stack Overflow, the documentation, and the code samples by using the following query in your favorite search engine:

```
{Your Search Terms} (site:stackoverflow.com OR site:docs.microsoft.com OR site:github.com/azure-samples OR site:cloudidentity.com OR site:developer.microsoft.com/graph)
```

Where *{Your Search Terms}* correspond to your search keywords.

Use the development support tools

TOOL	DESCRIPTION
jwt.ms	Paste an ID or access token to decode the claims names and values.
Microsoft Graph Explorer	Tool that lets you make requests and see responses against the Microsoft Graph API.

Post a question to Stack Overflow

Stack Overflow is the preferred channel for development-related questions. Here, members of the developer community and Microsoft team members are directly involved in helping you to solve your problems.

If you can't find an answer to your question through search, submit a new question to Stack Overflow. Use one of the following tags when asking questions to help the community identify and answer your question more quickly:

COMPONENT/AREA	TAGS
ADAL library	[adal]
MSAL library	[msal]
OWIN middleware	[azure-active-directory]
Azure B2B	[azure-ad-b2b]
Azure B2C	[azure-ad-b2c]
Microsoft Graph API	[microsoft-graph]
Any other area related to authentication or authorization topics	[azure-active-directory]

The following posts from Stack Overflow contain tips on how to ask questions and how to add source code. Follow these guidelines to increase the chances for community members to assess and respond to your question quickly:

- [How do I ask a good question](#)
- [How to create a minimal, complete, and verifiable example](#)

Create a GitHub issue

If you find a bug or problem related to our libraries, raise an issue in our GitHub repositories. Because our libraries are open source, you can also submit a pull request.

For a list of libraries and their GitHub repositories, see the following:

- [ADAL](#) libraries and GitHub repositories
- [MSAL.NET](#) [MSAL.js](#), [MSAL.Android](#), and [MSAL.obj_c](#) libraries and GitHub repositories

Open a support request

If you need to talk to someone, you can open a support request. If you are an Azure customer, there are several support options available. To compare plans, see [this page](#). Developer support is also available for Azure customers. For information on how to purchase Developer support plans, see [this page](#).

- If you already have an Azure Support Plan, [open a support request here](#)
- If you are not an Azure customer, you can also open a support request with Microsoft via [our commercial support](#).

You can also try a [virtual agent](#) to obtain support or ask questions.