

Contents

[Microsoft identity platform documentation](#)

[Overview](#)

[What is the Microsoft identity platform?](#)

[What's new in docs?](#)

[Code samples](#)

[Concepts](#)

[Basics of the identity platform](#)

[Authentication vs. authorization](#)

[OAuth 2.0 and OpenID Connect](#)

[App types and authentication flows](#)

[Security tokens](#)

[Microsoft Authentication Library \(MSAL\)](#)

[Microsoft Graph](#)

[Glossary of terms](#)

[Permissions and access](#)

[Authorization options: ACLs, RBAC, ABAC](#)

[RBAC for app developers](#)

[Scopes, permissions, and consent](#)

[Application consent experiences](#)

[Consent framework](#)

[Conditional access \(CA\) dev guide](#)

[Conditional access \(CA\) auth context](#)

[App registrations and workload identities](#)

[Application model](#)

[Workload identities](#)

[Applications and service principals](#)

[How and why apps are added to Azure AD](#)

[Single-tenant and multi-tenant apps](#)

[Security best practices](#)

- [Zero Trust for app developers](#)
- [Least privileged access for applications](#)
- [App registration security](#)
- [Secure access control using groups in Azure AD](#)
- [Identity platform best practices](#)
- [How-to](#)
 - [Build](#)
 - [Microsoft auth libraries by app type](#)
 - [Prepare for development](#)
 - [Get an Azure AD tenant](#)
 - [Register app or web API](#)
 - [Expose scopes in web API registration](#)
 - [Grant scoped permission to web API](#)
 - [Create a self-signed certificate to authenticate your app](#)
 - [Sign in users](#)
 - [Sign-in audience](#)
 - [App sign-in flow](#)
 - [Support passwordless authentication](#)
 - [Protect and access APIs](#)
 - [Restrict your app to a set of users](#)
 - [Add app roles in your application](#)
 - [Implement RBAC in your application](#)
 - [Support FIDO2 authentication](#)
 - [Customize tokens and claims](#)
 - [Configure optional claims](#)
 - [Claims mapping policy type](#)
 - [Use directory schema extension attributes in claims](#)
 - [Customize claims using Azure portal](#)
 - [Customize claims using PowerShell](#)
 - [Configure token lifetimes](#)
 - [Configure role claim](#)
 - [Handle browser cookie restrictions](#)

[Handle ITP in Safari](#)

[Handle SameSite cookie changes in Chrome](#)

Connect

[Workload identity federation](#)

[Trust an external identity provider \(federation\)](#)

[Configure an app to trust a GitHub repo](#)

[Access identity platform-protected resources from GCP](#)

[Exchange AD FS SAML for Microsoft Graph access token](#)

Deploy

[Automatic user provisioning \(SCIM\)](#)

[Common provisioning scenarios](#)

[Automate configuration using MS Graph](#)

[Develop and integrate a SCIM endpoint](#)

Secure

[Build for resilience](#)

[Development Overview](#)

[Client applications](#)

[Daemon applications](#)

[OpenID Connect metadata refresh](#)

[Use Continuous Access Evaluation APIs](#)

[Claims challenges and requests](#)

Test

[Build a test environment](#)

[Run automated integration tests as a user](#)

[Throttling and service limits](#)

[Create a service principal using the Azure portal](#)

[Create a service principal using Azure PowerShell](#)

Publish

[Prepare app for multi-tenant access](#)

[Enable multi-tenant log-ins](#)

[Modify accounts supported by an app](#)

[Become a verified publisher](#)

- [Publisher verification](#)
 - [Configure the publisher domain](#)
 - [Mark your app as publisher verified](#)
 - [Troubleshoot publisher verification](#)
- [Satisfy marketplace requirements](#)
 - [Follow the branding guidelines](#)
 - [Configure Terms of Service and Privacy Statement](#)
- [Publish app to a marketplace](#)
 - [Publish to App Source](#)
 - [Publish to Azure AD App Gallery](#)
 - [Publish to Microsoft 365 App Stores](#)
- [Migrate](#)
 - [Migrate apps from ADAL to MSAL](#)
 - [List all apps using ADAL](#)
- [Deprecate](#)
 - [Remove an app registration](#)
 - [Restore or remove a deleted app registration](#)
- [Single-page app \(SPA\)](#)
 - [SPA authentication documentation](#)
 - [Quickstart](#)
 - [Tutorials](#)
 - [Angular](#)
 - [Blazor WebAssembly](#)
 - [JavaScript](#)
 - [React](#)
 - [Samples](#)
 - [How-to](#)
 - [SPA that signs in users and calls web API](#)
 - [Overview](#)
 - [App registration](#)
 - [Code configuration](#)
 - [Sign in and sign out](#)

[Acquire token for an API](#)

[Call a web API](#)

[Move to production](#)

[Reference](#)

[MSAL.js](#)

[Library initialization](#)

[Init MSAL.js 2.x and 1.x clients](#)

[Running and debugging in IE](#)

[Integrate with Azure AD B2C](#)

[Authentication and single sign-on \(SSO\)](#)

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

[Pass custom state in authentication requests](#)

[Prompt behavior](#)

[Avoid page reloads](#)

[Migration](#)

[Migrate to MSAL.js](#)

[Migrate SPA from implicit to auth code flow](#)

[Logging and error handling](#)

[Logging in MSAL.js](#)

[Handle errors and exceptions in MSAL.js](#)

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

[Web app](#)

[Web app authentication documentation](#)

[Quickstart](#)

[Tutorials](#)

[ASP.NET](#)

[Blazor Server](#)

[Node.js](#)

[Samples](#)

[How-to](#)

[Web app that signs in users](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

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

[Move to production](#)

[Web app that calls an API](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Global sign-out](#)

[Acquire token for the app](#)

[Call a web API](#)

[Move to production](#)

[Web API](#)

[Web API authentication documentation](#)

[Quickstart](#)

[Samples](#)

[How-to](#)

[Protected web API](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Verification of scopes or app roles](#)

[Move to production](#)

[Web API that calls an API](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Acquire a token](#)

[Call a web API](#)

[Move to production](#)

[Reference](#)

[Scopes for web API accepting v1.0 tokens](#)

Desktop

[Desktop app authentication documentation](#)

[Quickstart](#)

[Tutorials](#)

[Universal Windows Platform \(UWP\)](#)

[Windows Presentation Foundation \(WPF\)](#)

[Electron](#)

[Samples](#)

[How-to](#)

[Desktop app that calls an API](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Acquire token](#)

[Overview](#)

[Interactively](#)

[Interactive with WAM](#)

[Integrated Windows authentication](#)

[Username and password](#)

[Device code flow](#)

[Call a web API](#)

[Move to production](#)

Mobile

[Mobile app authentication documentation](#)

[Quickstart](#)

[Tutorials](#)

[Android](#)

[Android - Shared-device mode](#)

[iOS and macOS](#)

[Samples](#)

[Concepts](#)

[Mobile single sign-on \(SSO\)](#)

Shared devices

How-to

[Mobile app that calls an API](#)

[Overview](#)

[App registration](#)

[Code configuration](#)

[Code configuration: Android and iOS](#)

[Android](#)

[System browser on Android](#)

[Xamarin Android](#)

[Shared device mode for Android devices](#)

[iOS](#)

[Microsoft Enterprise SSO plug-in for Apple devices](#)

[Xamarin iOS](#)

[Shared device mode for iOS devices](#)

[Acquire token](#)

[Acquire token with broker](#)

[Calling a web API](#)

[Move to production](#)

[Migrate broker-enabled Xamarin Android app MSAL.NET](#)

[Migrate broker-enabled Xamarin iOS app MSAL.NET](#)

Reference

[MSAL Android](#)

[Library initialization](#)

[Android MSAL configuration file](#)

[Shared device mode](#)

[Integrate with Azure AD B2C](#)

[Authentication and single sign-on \(SSO\)](#)

[SSO with MSAL Android](#)

[Accounts and tenant profiles](#)

[Single- and multi-account public client apps](#)

[Logging and error handling](#)

- [Logging in MSAL for Android](#)
- [Handle errors and exceptions in MSAL for Android](#)
- [Migrate from ADAL to MSAL Android](#)
- [Service, daemon, script](#)
- [Service/daemon authentication documentation](#)
- [Quickstart](#)
- [Tutorials](#)
 - [ASP.NET](#)
 - [Node.js](#)
- [Samples](#)
- [How-to](#)
 - [Non-interactive app that calls a web API](#)
 - [Overview](#)
 - [App registration](#)
 - [Code configuration](#)
 - [Acquire token](#)
 - [Call a web API](#)
 - [Move to production](#)
- [Reference](#)
 - [Microsoft Authentication Library \(MSAL\) reference](#)
 - [Auth flows supported by MSAL](#)
 - [Request and cache tokens](#)
 - [Public and confidential client apps](#)
 - [National clouds and MSAL](#)
 - [MSAL iOS and macOS](#)
 - [Library initialization](#)
 - [MSAL for iOS and macOS differences](#)
 - [Redirect URI configuration](#)
 - [Customize browsers and WebViews](#)
 - [Shared device mode for iOS devices](#)
 - [Integrate with Azure AD B2C](#)
 - [Authentication and single sign-on \(SSO\)](#)

[Configure keychain](#)

[SSO between MSAL apps](#)

[SSO between ADAL and MSAL apps](#)

[Request custom claims](#)

[Logging and error handling](#)

[Logging in MSAL for iOS/macOS](#)

[Handle errors and exceptions in MSAL for iOS/macOS](#)

[Troubleshoot SSL issues \(MSAL iOS/macOS\)](#)

[Migrate to MSAL.iOS / macOS](#)

[MSAL Java](#)

[Custom token cache serialization](#)

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

[Migrate to MSAL for Java](#)

[AD FS support in MSAL for Java](#)

[Handle errors and exceptions in MSAL for Java](#)

[Logging in MSAL for Java](#)

[MSAL.NET](#)

[Library initialization](#)

[Microsoft.Identity.Web \(MSAL wrapper\)](#)

[Application configuration](#)

[Application initialization](#)

[Confidential client instantiation](#)

[Confidential client credentials \(cert, secret, assertion\)](#)

[Public client instantiation](#)

[Instantiation with custom HttpClient](#)

[Web browser interaction \(system and embedded\)](#)

[Universal Windows Platform](#)

[Authentication and single sign-on \(SSO\)](#)

[Get a token from the cache](#)

[Clear the token cache](#)

[Serialize the token cache](#)

[Get consent for several resources](#)

- [Connections and integrations](#)
 - [Integrate with AD FS](#)
 - [Integrate with Azure AD B2C](#)
- [Logging and error handling](#)
 - [Handle errors and exceptions in MSAL.NET](#)
 - [Logging in MSAL.NET](#)
- [Migration to MSAL.NET](#)
 - [Migrate from ADAL.NET to MSAL.NET](#)
 - [Migrate confidential client apps to MSAL.NET](#)
 - [Migrate public client apps to MSAL.NET](#)
 - [Differences between ADAL.NET and MSAL.NET](#)
- [MSAL Node](#)
 - [Migrate to MSAL Node](#)
 - [Cache tokens to disk](#)
- [MSAL Python](#)
 - [Custom token cache serialization](#)
 - [Migrate to MSAL Python](#)
 - [AD FS support in MSAL Python](#)
 - [Handle errors and exceptions in MSAL for Python](#)
 - [Logging in MSAL for Python](#)
- [Protocol reference \(OAuth, OIDC, SAML\)](#)
 - [OAuth 2.0 and OpenID Connect \(OIDC\)](#)
 - [Token grant flows](#)
 - [OAuth 2.0 auth code grant](#)
 - [OAuth 2.0 client credentials grant](#)
 - [OAuth 2.0 device code flow](#)
 - [OAuth 2.0 on-behalf-of flow](#)
 - [OAuth 2.0 implicit grant flow](#)
 - [OAuth 2.0 resource owner password credentials grant](#)
 - [OpenID Connect](#)
 - [Security tokens](#)
 - [Access tokens](#)

- [ID tokens](#)
- [Refresh tokens](#)
- [Token lifetime](#)
- [OAuth 2.0 application types](#)
- [Certificate credentials](#)
- [Signing key rollover](#)
- [UserInfo endpoint \(OIDC\)](#)
- [SAML 2.0](#)
 - [How Azure AD uses the SAML protocol](#)
 - [SAML single sign-on \(SSO\)](#)
 - [Sign out \(SAML\)](#)
 - [SAML tokens](#)
 - [Token exchange scenarios](#)
- [App registration reference](#)
 - [Application manifest](#)
 - [Redirect URI/reply URL restrictions and limitations](#)
 - [Validation differences by supported account types](#)
 - [Microsoft auth libraries by app type](#)
 - [AADSTS error code reference](#)
 - [Endpoint reference: Admin consent URI](#)
 - [Endpoint reference: National and public cloud URIs](#)
 - [Microsoft Graph API reference \(identity operations\)](#)
- [Resources](#)
 - [Help and support options](#)
 - [Updates and breaking changes](#)
 - [Video learning](#)
 - [Blog: M365 Developer - Microsoft identity platform](#)
 - [Blog: Azure AD - Identity](#)
 - [Azure roadmap - security and identity](#)

What is the Microsoft identity platform?

4/12/2022 • 2 minutes to read • [Edit Online](#)

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

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

- **OAuth 2.0 and OpenID Connect standard-compliant authentication service** enabling developers to authenticate several identity types, including:
 - Work or school accounts, provisioned through Azure AD
 - Personal Microsoft account, like Skype, Xbox, and Outlook.com
 - Social or local accounts, by using 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 in the Azure portal, along with the other Azure management capabilities.
- **Application configuration API and PowerShell**: Programmatic configuration of your applications through the Microsoft Graph API and PowerShell so you can automate your DevOps tasks.
- **Developer content**: Technical documentation including quickstarts, tutorials, how-to guides, and code samples.

For developers, the Microsoft identity platform offers integration of modern innovations in the identity and security space like 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 the 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).

For a video overview of the platform and a demo of the authentication experience, see [What is the Microsoft identity platform for developers?](#).

Getting started

Choose the [application scenario](#) you'd like to build. Each of these scenario paths starts with an overview and links to a quickstart to help you get up and running:

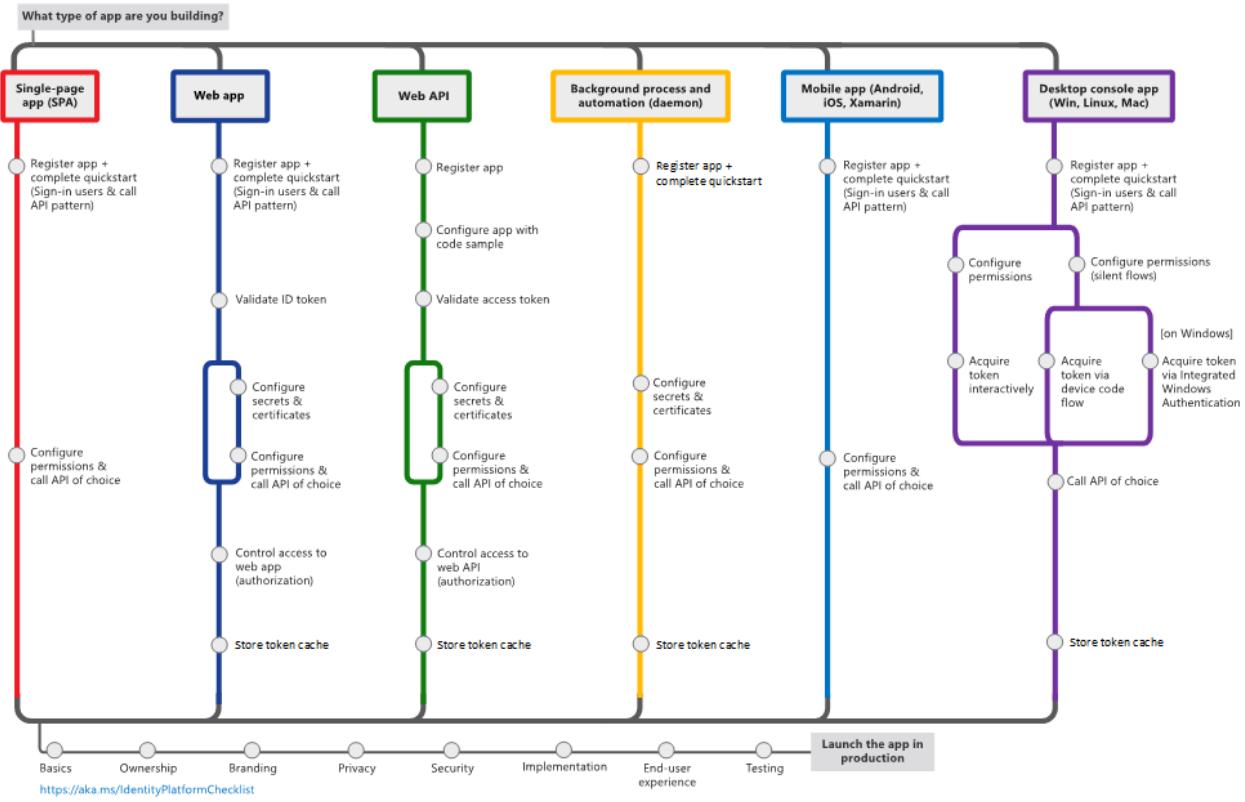
- [Single-page app \(SPA\)](#)
- [Web app that signs in users](#)
- [Web app that calls web APIs](#)
- [Protected web API](#)
- [Web API that calls web APIs](#)
- [Desktop app](#)
- [Daemon app](#)
- [Mobile app](#)

As you work with the Microsoft identity platform to integrate authentication and authorization in your apps, you

can refer to this image that outlines the most common app scenarios and their identity components. Select the image to view it full-size.

Microsoft identity platform

<http://aka.ms/identityPlatform>



Learn authentication concepts

Learn how core authentication and Azure AD concepts apply to the Microsoft identity platform in this recommended set of articles:

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

More identity and access management options

[Azure AD B2C](#) - Build customer-facing applications your users can sign in to using their social accounts like Facebook or Google, or by using an email address and password.

[Azure AD B2B](#) - Invite external users into your Azure AD tenant as "guest" users, and assign permissions for authorization while they use their existing credentials for authentication.

[Azure Active Directory for developers \(v1.0\)](#) - Shown here for developers with existing apps that use the older v1.0 endpoint. **Do not** use v1.0 for new projects.

Next steps

If you have an Azure account you already have access to an Azure Active Directory tenant, but most Microsoft identity platform developers need their own Azure AD tenant for use while developing applications, a "dev tenant."

Learn how to create your own tenant for use while building your applications:

[Quickstart: Set up an Azure AD tenant](#)

Microsoft identity platform docs: What's new

4/12/2022 • 2 minutes to read • [Edit Online](#)

Welcome to what's new in the Microsoft identity platform documentation. This article lists new docs that have been added and those that have had significant updates in the last three months.

March 2022

New articles

- [Secure access control using groups in Azure AD](#)

Updated articles

- [Authentication flow support in MSAL](#)
- [Claims mapping policy type](#)
- [Configure an app to trust an external identity provider \(preview\)](#)
- [OAuth 2.0 and OpenID Connect in the Microsoft identity platform](#)
- [Signing key rollover in the Microsoft identity platform](#)
- [Troubleshoot publisher verification](#)

February 2022

Updated articles

- [Desktop app that calls web APIs: Acquire a token using WAM](#)

January 2022

New articles

- [Access Azure AD protected resources from an app in Google Cloud \(preview\)](#)

Updated articles

- [Confidential client assertions](#)
- [Claims mapping policy type](#)
- [Configure an app to trust a GitHub repo \(preview\)](#)
- [Configure an app to trust an external identity provider \(preview\)](#)
- [Exchange a SAML token issued by AD FS for a Microsoft Graph access token](#)
- [Logging in MSAL.js](#)
- [Permissions and consent in the Microsoft identity platform](#)
- [Quickstart: Acquire a token and call the Microsoft Graph API by using a console app's identity](#)
- [Quickstart: Acquire a token and call Microsoft Graph API from a desktop application](#)
- [Quickstart: Add sign-in with Microsoft to a web app](#)
- [Quickstart: Protect a web API with the Microsoft identity platform](#)
- [Quickstart: Sign in users and call the Microsoft Graph API from a mobile application](#)

Microsoft identity platform code samples

4/12/2022 • 9 minutes to read • [Edit Online](#)

These code samples are built and maintained by Microsoft to demonstrate usage of our authentication libraries with the Microsoft identity platform. Common authentication and authorization scenarios are implemented in several [application types](#), development languages, and frameworks.

- Sign in users to web applications and provide authorized access to protected web APIs.
- Protect a web API by requiring an access token to perform API operations.

Each code sample includes a *README.md* file describing how to build the project (if applicable) and run the sample application. Comments in the code help you understand how these libraries are used in the application to perform authentication and authorization by using the identity platform.

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Angular	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Call .NET Core web API• Call .NET Core web API (B2C)• Call Microsoft Graph via OBO• Call .NET Core web API using PoP• Use App Roles for access control• Use Security Groups for access control• Deploy to Azure Storage and App Service	MSAL Angular	<ul style="list-style-type: none">• Authorization code with PKCE• On-behalf-of (OBO)• Proof of Possession (PoP)
Blazor WebAssembly	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Deploy to Azure App Service	MSAL.js	Implicit Flow

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
JavaScript	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API via OBO and CA • Deploy to Azure Storage and App Service 	MSAL.js	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA)
React	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API using PoP • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service • Deploy to Azure Static Web Apps 	MSAL React	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA) • Proof of Possession (PoP)

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Customize token cache • Call Graph (multi-tenant) • Call Azure REST APIs • Protect web API • Protect web API (B2C) • Protect multi-tenant web API • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service 	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	<ul style="list-style-type: none"> • OpenID connect • Authorization code • On-Behalf-Of

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Blazor	Blazor Server Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call web API • Call web API (B2C) 	MSAL.NET	Authorization code Grant Flow
ASP.NET Core	Advanced Token Cache Scenarios	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	On-Behalf-Of (OBO)
ASP.NET Core	Use the Conditional Access auth context to perform step-up authentication	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	Authorization code
ASP.NET Core	Active Directory FS to Azure AD migration	MSAL.NET	<ul style="list-style-type: none"> • SAML • OpenID connect
ASP.NET	<ul style="list-style-type: none"> • Microsoft Graph Training Sample • Sign in users and call Microsoft Graph • Sign in users and call Microsoft Graph with admin restricted scope • Quickstart: Sign in users 	MSAL.NET	<ul style="list-style-type: none"> • OpenID connect • Authorization code
Java Spring	Azure AD Spring Boot Starter Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Groups for access control • Deploy to Azure App Service 	<ul style="list-style-type: none"> • MSAL Java • Azure AD Boot Starter 	Authorization code
Java Servlets	Spring-less Servlet Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure App Service 	MSAL Java	Authorization code
Java	Sign in users and call Microsoft Graph	MSAL Java	Authorization code
Java Spring	Sign in users and call Microsoft Graph via OBO • Web API	MSAL Java	<ul style="list-style-type: none"> • Authorization code • On-Behalf-Of (OBO)

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js Express	Express web app series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service • Use App Roles for access control • Use Security Groups for access control • Web app that sign in users 	MSAL Node	Authorization code
Python Flask	Flask Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Sign in users and call Microsoft Graph • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Python Django	Django Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Ruby	Graph Training <ul style="list-style-type: none"> • Sign in users and call Microsoft Graph 	OmniAuth OAuth2	Authorization code

Web API

The following samples show how to protect a web API with the Microsoft identity platform, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET	Call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
ASP.NET Core	Sign in users and call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
Java	Sign in users	MSAL Java	On-Behalf-Of (OBO)
Node.js	<ul style="list-style-type: none"> • Protect a Node.js web API • Protect a Node.js Web API with Azure AD B2C 	MSAL Node	Authorization bearer

Desktop

The following samples show public client desktop applications that access the Microsoft Graph API, or your own

web API in the name of the user. Apart from the *Desktop (Console) with Web Authentication Manager (WAM)* sample, all these client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> • Call Microsoft Graph • Call Microsoft Graph with token cache • Call Microsoft Graph with custom web UI HTML • Call Microsoft Graph with custom web browser • Sign in users with device code flow 	MSAL.NET	<ul style="list-style-type: none"> • Authorization code with PKCE • Device code
.NET	<ul style="list-style-type: none"> • Call Microsoft Graph with daemon console • Call web API with daemon console 	MSAL.NET	Authorization code with PKCE
.NET	Invoke protected API with integrated Windows authentication	MSAL.NET	Integrated Windows authentication
Java	Call Microsoft Graph	MSAL Java	Integrated Windows authentication
Node.js	Sign in users	MSAL Node	Authorization code with PKCE
PowerShell	Call Microsoft Graph by signing in users using username/password	MSAL.NET	Resource owner password credentials
Python	Sign in users	MSAL Python	Resource owner password credentials
Universal Window Platform (UWP)	Call Microsoft Graph	MSAL.NET	Web account manager
Windows Presentation Foundation (WPF)	Sign in users and call Microsoft Graph	MSAL.NET	Authorization code with PKCE
XAML	<ul style="list-style-type: none"> • Sign in users and call ASP.NET core web API • Sign in users and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Mobile

The following samples show public client mobile applications that access the Microsoft Graph API, or your own web API in the name of the user. These client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
iOS	<ul style="list-style-type: none"> Call Microsoft Graph native Call Microsoft Graph with Azure AD nxauth 	MSAL iOS	Authorization code with PKCE
Java	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Kotlin	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Xamarin	<ul style="list-style-type: none"> Sign in users and call Microsoft Graph Sign in users with broker and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Service / daemon

The following samples show an application that accesses the Microsoft Graph API with its own identity (with no user).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> Call Microsoft Graph Call web API Call own web API Using managed identity and Azure key vault 	MSAL.NET	Client credentials grant
ASP.NET	Multi-tenant with Microsoft identity platform endpoint	MSAL.NET	Client credentials grant
Java	Call Microsoft Graph	MSAL Java	Client credentials grant
Node.js	Sign in users and call web API	MSAL Node	Client credentials grant
Python	<ul style="list-style-type: none"> Call Microsoft Graph with secret Call Microsoft Graph with certificate 	MSAL Python	Client credentials grant

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, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET	.NET Azure function web API secured by Azure AD	MSAL.NET	Authorization code

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Node.js Azure function web API secured by Azure AD	MSAL Node	Authorization bearer
Node.js	Call Microsoft Graph API on behalf of a user	MSAL Node	On-Behalf-Of (OBO)
Python	Python Azure function web API secured by Azure AD	MSAL Python	Authorization code

Headless

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 the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET core	Invoke protected API from text-only device	MSAL.NET	Device code
Java	Sign in users and invoke protected API	MSAL Java	Device code
Python	Call Microsoft Graph	MSAL Python	Device code

Microsoft Teams applications

The following sample illustrates Microsoft Teams Tab application that signs in users. Additionally it demonstrates how to call Microsoft Graph API with the user's identity using the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Teams Tab app: single sign-on (SSO) and call Microsoft Graph	MSAL Node	On-Behalf-Of (OBO)

Multi-tenant SaaS

The following samples show how to configure your application to accept sign-ins from any Azure Active Directory (Azure AD) tenant. Configuring your application to be *multi-tenant* means that you can offer a **Software as a Service** (SaaS) application to many organizations, allowing their users to be able to sign-in to your application after providing consent.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core MVC web application calls Microsoft Graph API	MSAL.NET	OpenID connect
ASP.NET Core	ASP.NET Core MVC web application calls ASP.NET Core Web API	MSAL.NET	Authorization code

Next steps

If you'd like to delve deeper into more sample code, see:

- [Sign in users and call the Microsoft Graph API from an Angular](#)
- [Sign in users in a Nodejs and Express web app](#)
- [Call the Microsoft Graph API from a Universal Windows Platform](#)

Authentication vs. authorization

4/12/2022 • 2 minutes to read • [Edit Online](#)

This article defines authentication and authorization. It also briefly covers how you can use the Microsoft identity platform to authenticate and authorize users in your web apps, web APIs, or apps that call protected web APIs. If you see a term you aren't familiar with, try our [glossary](#) or our [Microsoft identity platform videos](#), which cover basic concepts.

Authentication

Authentication is the process of proving that you are who you say you are. It's sometimes shortened to *AuthN*. The Microsoft identity platform uses the [OpenID Connect](#) protocol for handling authentication.

Authorization

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*. The Microsoft identity platform uses the [OAuth 2.0](#) protocol for handling authorization.

Authentication and authorization using the Microsoft identity platform

Creating apps that each maintain their own username and password information incurs a high administrative burden when adding or removing users across multiple apps. Instead, your 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](#), which is sometimes called two-factor authentication or 2FA.
- 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 called *single sign-on (SSO)*.

The Microsoft identity platform simplifies authorization and authentication for application developers by providing identity as a service. It supports industry-standard protocols and 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](#), access Microsoft APIs, or access other APIs that developers have built.

This video explains the Microsoft identity platform and the basics of modern authentication:

Here's a comparison of the protocols that the Microsoft identity platform uses:

- **OAuth versus OpenID Connect:** The platform uses OAuth for authorization and OpenID Connect (OIDC) for authentication. OpenID Connect is built on top of OAuth 2.0, so the terminology and flow are similar between the two. You can even both authenticate a user (through OpenID Connect) and get authorization to access a protected resource that the user owns (through OAuth 2.0) in one request. For more information, see [OAuth 2.0 and OpenID Connect protocols](#) and [OpenID Connect protocol](#).
- **OAuth versus SAML:** The platform uses OAuth 2.0 for authorization and SAML for authentication. For

more information on how to use these protocols together to both authenticate a user and get authorization to access a protected resource, see [Microsoft identity platform and OAuth 2.0 SAML bearer assertion flow](#).

- **OpenID Connect versus SAML:** The platform uses both OpenID Connect and SAML to authenticate a user and enable single sign-on. SAML authentication is commonly used with identity providers such as Active Directory Federation Services (AD FS) federated to Azure AD, so it's often used in enterprise applications. OpenID Connect is commonly used for apps that are purely in the cloud, such as mobile apps, websites, and web APIs.

Next steps

For other topics that cover authentication and authorization basics:

- To learn how access tokens, refresh tokens, and ID tokens are used in authorization and authentication, see [Security tokens](#).
- To learn about the process of registering your application so it can integrate with the Microsoft identity platform, see [Application model](#).

OAuth 2.0 and OpenID Connect (OIDC) in the Microsoft identity platform

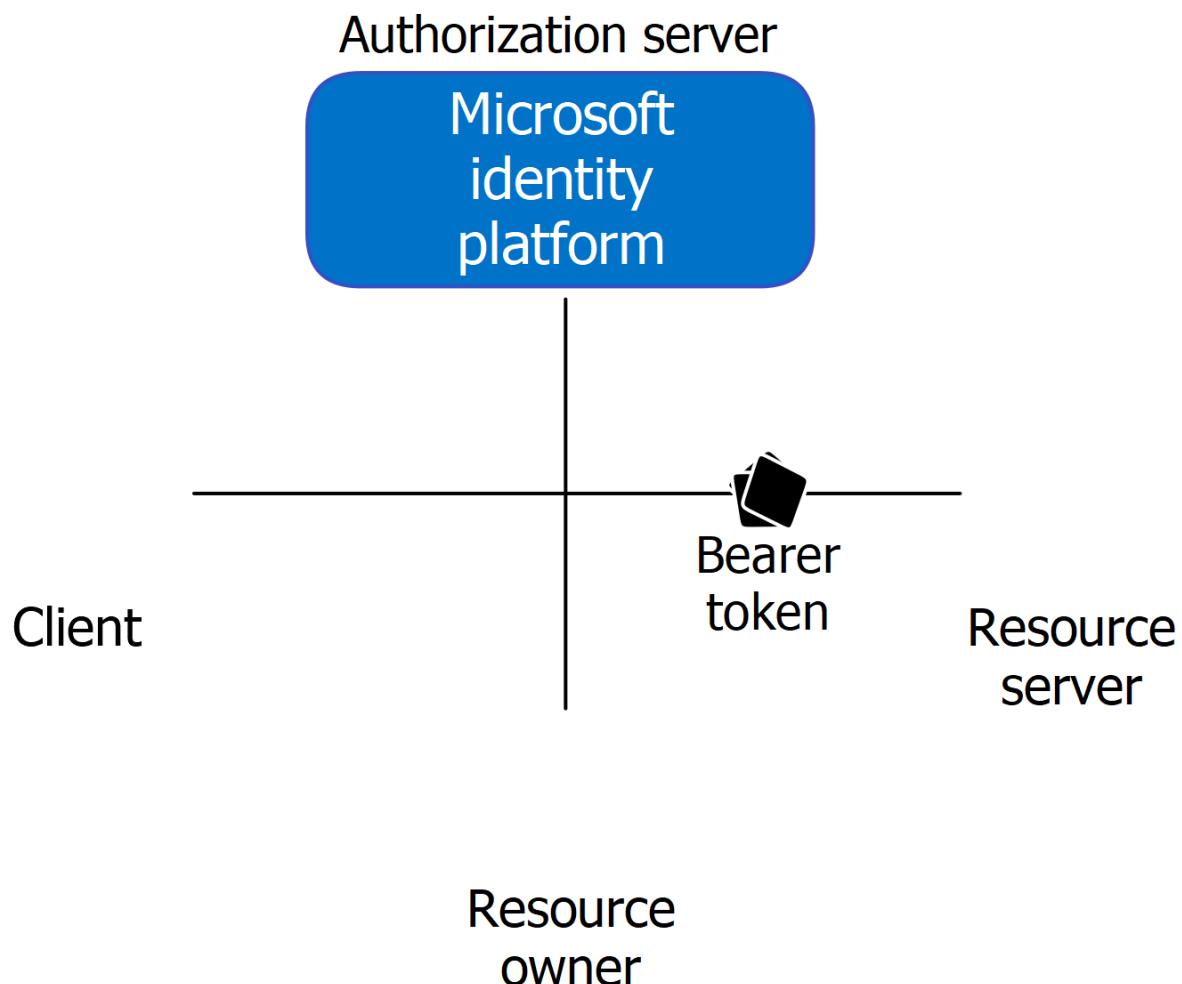
4/12/2022 • 5 minutes to read • [Edit Online](#)

You don't need to learn OAuth or OpenID Connect (OIDC) at the protocol level to use the Microsoft identity platform. You will, however, encounter these and other protocol terms and concepts as you use the identity platform to add auth functionality to your apps.

As you work with the Azure portal, our documentation, and our authentication libraries, knowing a few basics like these can make your integration and debugging tasks easier.

Roles in OAuth 2.0

Four parties are typically involved in an OAuth 2.0 and OpenID Connect authentication and authorization exchange. Such exchanges are often called *authentication flows* or *auth flows*.



- **Authorization server** - The Microsoft identity platform itself is the authorization server. Also called an *identity provider* or *IdP*, it securely handles the end-user's information, their access, and the trust relationships between the parties in the auth flow. The authorization server issues the security tokens your apps and APIs use for granting, denying, or revoking access to resources (authorization) after the user has signed in (authenticated).
- **Client** - The client in an OAuth exchange is the application requesting access to a protected resource. The

client could be a web app running on a server, a single-page web app running in a user's web browser, or a web API that calls another web API. You'll often see the client referred to as *client application*, *application*, or *app*.

- **Resource owner** - The resource owner in an auth flow is typically the application user, or *end-user* in OAuth terminology. The end-user "owns" the protected resource--their data--your app accesses on their behalf. The resource owner can grant or deny your app (the client) access to the resources they own. For example, your app might call an external system's API to get a user's email address from their profile on that system. Their profile data is a resource the end-user owns on the external system, and the end-user can consent to or deny your app's request to access their data.
- **Resource server** - The resource server hosts or provides access to a resource owner's data. Most often, the resource server is a web API fronting a data store. The resource server relies on the authorization server to perform authentication and uses information in bearer tokens issued by the authorization server to grant or deny access to resources.

Tokens

The parties in an authentication flow use **bearer tokens** to assure identification (authentication) and to grant or deny access to protected resources (authorization). Bearer tokens in the Microsoft identity platform are formatted as [JSON Web Tokens](#) (JWT).

Three types of bearer tokens are used by the Microsoft identity platform as *security tokens*:

- **Access tokens** - Access tokens are issued by the authorization server to the client application. The client passes access tokens to the resource server. Access tokens contain the permissions the client has been granted by the authorization server.
- **ID tokens** - ID tokens are issued by the authorization server to the client application. Clients use ID tokens when signing in users and to get basic information about them.
- **Refresh tokens** - The client uses a refresh token, or *RT*, to request new access and ID tokens from the authorization server. Your code should treat refresh tokens and their string content as opaque because they're intended for use only by authorization server.

App registration

Your client app needs a way to trust the security tokens issued to it by the Microsoft identity platform. The first step in establishing that trust is by [registering your app](#) with the identity platform in Azure Active Directory (Azure AD).

When you register your app in Azure AD, the Microsoft identity platform automatically assigns it some values, while others you configure based on the application's type.

Two the most commonly referenced app registration settings are:

- **Application (client) ID** - Also called *application ID* and *client ID*, this value is assigned to your app by the Microsoft identity platform. The client ID uniquely identifies your app in the identity platform and is included in the security tokens the platform issues.
- **Redirect URI** - The authorization server uses a redirect URI to direct the resource owner's *user-agent* (web browser, mobile app) to another destination after completing their interaction. For example, after the end-user authenticates with the authorization server. Not all client types use redirect URIs.

Your app's registration also holds information about the authentication and authorization *endpoints* you'll use in your code to get ID and access tokens.

Endpoints

The Microsoft identity platform offers authentication and authorization services using standards-compliant implementations of OAuth 2.0 and OpenID Connect (OIDC) 1.0. Standards-compliant authorization servers like the Microsoft identity platform provide a set of HTTP endpoints for use by the parties in an auth flow to execute the flow.

The endpoint URLs for your app are generated for you when you register or configure your app in Azure AD. The endpoints you use in your app's code depend on the application's type and the identities (account types) it should support.

Two commonly used endpoints are the [authorization endpoint](#) and [token endpoint](#). Here are examples of the `authorize` and `token` endpoints:

```
# Authorization endpoint - used by client to obtain authorization from the resource owner.  
https://login.microsoftonline.com/<issuer>/oauth2/v2.0/authorize  
# Token endpoint - used by client to exchange an authorization grant or refresh token for an access token.  
https://login.microsoftonline.com/<issuer>/oauth2/v2.0/token  
  
# NOTE: These are examples. Endpoint URI format may vary based on application type,  
#       sign-in audience, and Azure cloud instance (global or national cloud).
```

To find the endpoints for an application you've registered, in the [Azure portal](#) navigate to:

Azure Active Directory > App registrations > <YOUR-APPLICATION> > Endpoints

Next steps

Next, learn about the OAuth 2.0 authentication flows used by each application type and the libraries you can use in your apps to perform them:

- [Authentication flows and application scenarios](#)
- [Microsoft Authentication Library \(MSAL\)](#)

We strongly advise against crafting your own library or raw HTTP calls to execute authentication flows. A [Microsoft authentication library](#) is safer and much easier. However, if your scenario prevents you from using our libraries or you'd just like to learn more about the identity platform's implementation, we have protocol reference:

- [Authorization code grant flow](#) - Single-page apps (SPA), mobile apps, native (desktop) applications
- [Client credentials flow](#) - Server-side processes, scripts, daemons
- [On-behalf-of \(OBO\) flow](#) - Web APIs that call another web API on a user's behalf
- [OpenID Connect](#) - User sign-in, sign-out, and single sign-on (SSO)

Authentication flows and application scenarios

4/12/2022 • 10 minutes to read • [Edit Online](#)

The Microsoft identity platform 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](#). By using the [authentication libraries for the Microsoft identity platform](#), applications authenticate identities and acquire tokens to access protected APIs.

This article describes authentication flows and the application scenarios that they're used in.

Application categories

Tokens can be acquired from several types of applications, including:

- Web apps
- Mobile apps
- Desktop apps
- Web APIs

Tokens can also be acquired by apps running on devices that don't have a browser or are running on the Internet of Things (IoT).

The following sections describe the categories of applications.

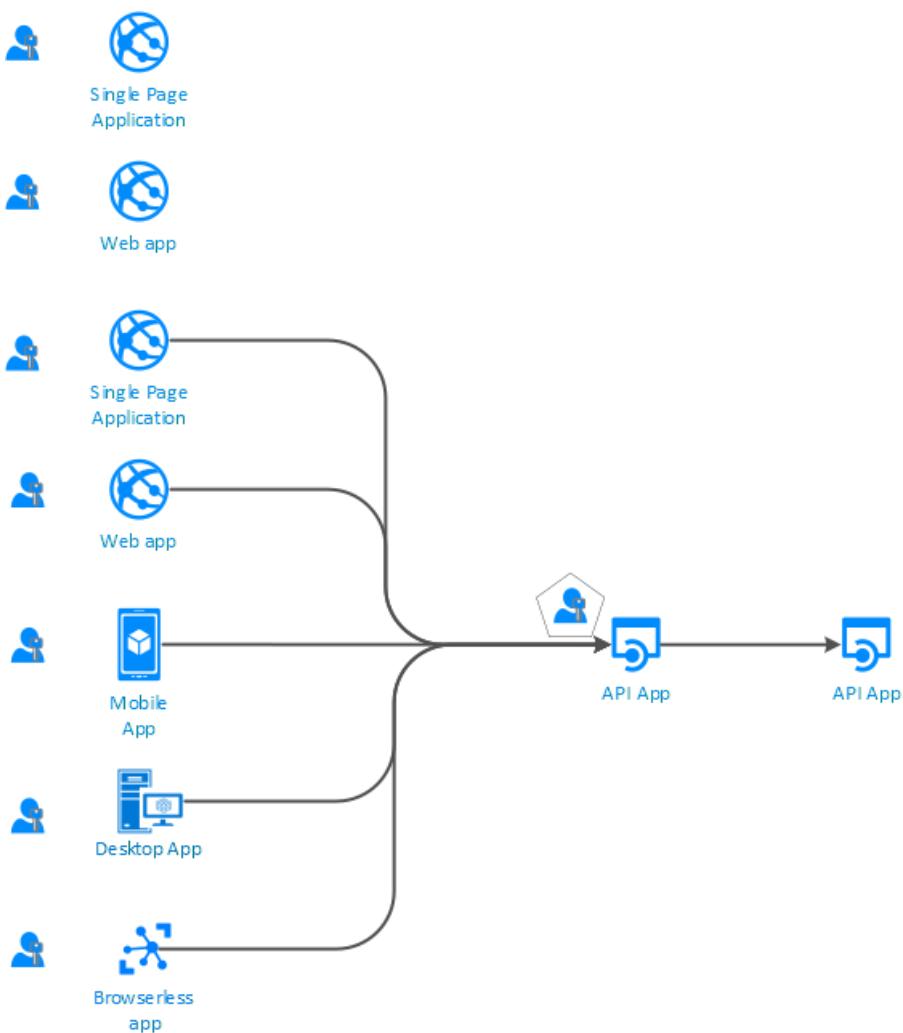
Protected resources vs. client applications

Authentication scenarios involve two activities:

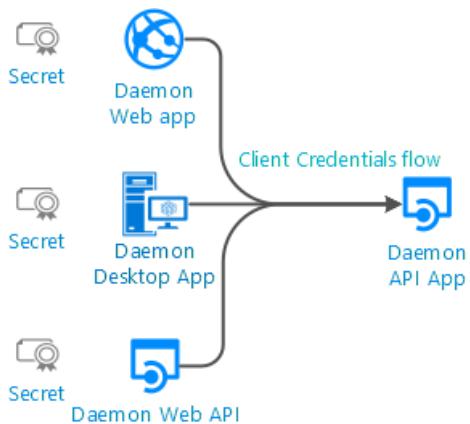
- **Acquiring security tokens for a protected web API:** We recommend that you use the [Microsoft Authentication Library \(MSAL\)](#), developed and supported by Microsoft.
- **Protecting a web API or a web app:** One challenge of protecting these resources is validating the security token. On some platforms, 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 apps. In these scenarios, applications acquire tokens on behalf of themselves with no user.



Single-page, public client, and confidential client applications

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

- **Single-page applications**: Also known as SPAs, these are web apps in which tokens are acquired by a JavaScript or TypeScript app running in the browser. Many modern apps have a single-page application at the front end that's 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**: Apps in this category, like the following types, always sign in users:
 - Desktop apps that call web APIs on behalf of signed-in users

- Mobile apps
- Apps running on devices that don't have a browser, like those running on IoT
- **Confidential client applications:** Apps in this category include:
 - Web apps that call a web API
 - Web APIs that call a web API
 - Daemon apps, even when implemented as a console service like a Linux daemon or a Windows service

Sign-in audience

The available authentication flows differ depending on the sign-in audience. Some flows are available only for work or school accounts. Others are available both for work or school accounts and for personal Microsoft accounts.

For more information, see [Supported account types](#).

Application scenarios

The Microsoft identity platform supports authentication for these app architectures:

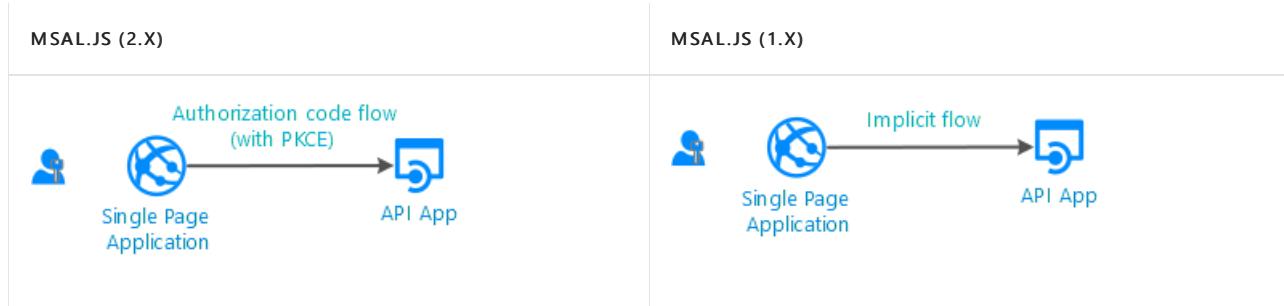
- Single-page apps
- Web apps
- Web APIs
- Mobile apps
- Native apps
- Daemon apps
- Server-side apps

Applications use the different authentication flows to sign in users and get tokens to call protected APIs.

Single-page application

Many modern web apps are built as client-side single-page applications. These applications use JavaScript or a framework like Angular, Vue, and React. These applications run in a web browser.

Single-page applications differ from traditional server-side web apps in terms of authentication characteristics. By using the Microsoft identity platform, single-page applications can sign in users and get tokens to access back-end services or web APIs. The Microsoft identity platform offers two grant types for JavaScript applications:



Web app that signs in a user



To help protect a web app that signs in a user:

- If you develop in .NET, you use ASP.NET or ASP.NET Core with the ASP.NET OpenID Connect middleware.

Protecting a resource involves validating the security token, which is done by the [IdentityModel extensions for .NET](#) and not MSAL libraries.

- If you develop in Node.js, you use [MSAL Node](#) or [Passport.js](#).

For more information, see [Web app that signs in users](#).

Web app that signs in a user and calls a web API on behalf of the user



To call a web API from a web app on behalf of a user, 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 [Web app that calls web APIs](#).

Desktop app that calls 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 MSAL. 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 that you use it, the [username/password flow](#) is available in public client applications. This flow is still needed in some scenarios like DevOps.

Using the username/password flow constrains your applications. For instance, applications can't sign in a user who needs to use multifactor authentication or the Conditional Access tool in Azure AD. 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 persist, you can customize the [token cache serialization](#). By implementing [dual token cache serialization](#), you can use backward-compatible and forward-compatible token caches. These tokens support 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](#).

Mobile app that calls a web API on behalf of an interactive user

Similar to a desktop app, a mobile app calls the interactive token-acquisition methods of MSAL 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. MSAL can now interact with brokers. For more information about brokers, see [Leveraging brokers on Android and iOS](#).

For more information, see [Mobile app that calls web APIs](#).

NOTE

A 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 is recognized by Intune as a managed app. For more information, see [Microsoft Intune App SDK overview](#).

The [Intune App SDK](#) is separate from MSAL libraries and interacts with Azure AD on its own.

Protected web API

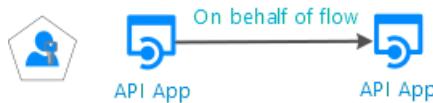
You can use the Microsoft identity platform endpoint to secure web services like your app's RESTful API. A protected web API is called through an access token. The token helps secure the API's data and authenticate incoming requests. The caller of a web API appends an access token in the authorization header of an HTTP request.

If you want to protect your ASP.NET or ASP.NET Core web API, 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](#).

Web API that calls another web API on behalf of a user

For your 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. Such calls are sometimes referred to as *service-to-service* calls. Web APIs that call other web APIs need to provide custom cache serialization.

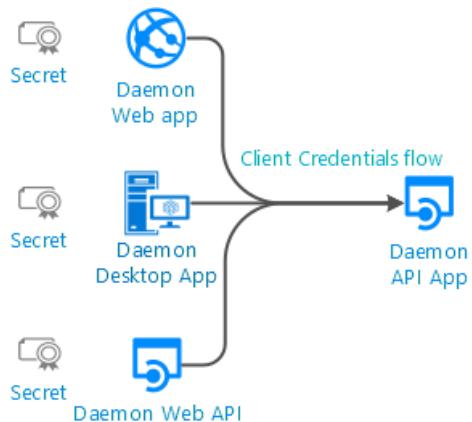


For more information, see [Web API that calls web APIs](#).

Daemon app that calls 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. 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 [client credential](#) acquisition methods in MSAL. These methods require a client secret that you add to the app registration in Azure AD. The app then shares the secret with the called daemon. Examples of such secrets include application passwords, certificate assertion, and client assertion.



For more information, see [Daemon application that calls web APIs](#).

Scenarios and supported authentication flows

You use authentication flows to implement the application scenarios that are requesting tokens. There isn't a one-to-one mapping between application scenarios and authentication flows.

Scenarios that involve acquiring tokens also map to OAuth 2.0 authentication flows. For more information, see [OAuth 2.0 and OpenID Connect protocols on the Microsoft identity platform](#).

SCENARIO	DETAILED SCENARIO WALK-THROUGH	OAUTH 2.0 FLOW AND GRANT	AUDIENCE
  Single Page Application	Single-page app	Authorization code with PKCE	Work or school accounts, personal accounts, and Azure Active Directory B2C (Azure AD B2C)
  Single Page Application	Single-page app	Implicit	Work or school accounts, personal accounts, and Azure Active Directory B2C (Azure AD B2C)
  Web app	Web app that signs in users	Authorization code	Work or school accounts, personal accounts, and Azure AD B2C
  Web app	Web app that calls web APIs	Authorization code	Work or school accounts, personal accounts, and Azure AD B2C
  Desktop App	Desktop app that calls web APIs	- Authorization code flow (with PKCE) - Integrated Windows Authentication - Username/Password	Work or school accounts, personal accounts, and Azure AD B2C
		Integrated Windows authentication	Work or school accounts

SCENARIO	DETAILED SCENARIO WALK-THROUGH	OAUTH 2.0 FLOW AND GRANT	AUDIENCE
		Resource owner password	Work or school accounts and Azure AD B2C
		Device code	Work or school accounts, personal accounts, but not Azure AD B2C
	Mobile app that calls web APIs	Interactive by using authorization code with PKCE	Work or school accounts, personal accounts, and Azure AD B2C
		Resource owner password	Work or school accounts and Azure AD B2C
	Daemon app that calls web APIs	Client credentials	App-only permissions that have no user and are used only in Azure AD organizations
	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:

- .NET Core
- .NET Framework
- Java
- JavaScript
- macOS
- Native Android
- Native iOS
- Node.js
- Python
- Windows 10/UWP
- Xamarin.iOS
- Xamarin.Android

You can also use various languages to build your applications.

NOTE

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
<p>Single-page app Single Page Application</p>	 MSAL.js	 MSAL.js	 MSAL.js	 MSAL.js	 MSAL.js
<p>Single-page app Single Page Application</p>	 MSAL.js	 MSAL.js	 MSAL.js	 MSAL.js	 MSAL.js
<p>Web app that signs in users Web app</p>	 ASP.NET Core MSAL Node	 ASP.NET Core MSAL Node	 ASP.NET Core MSAL Node		
<p>Web app that calls web APIs Web app</p>	 ASP.NET Core + MSAL.NET MSAL Java Flask + MSAL Python MSAL Node	 ASP.NET Core + MSAL.NET MSAL Java Flask + MSAL Python MSAL Node	 ASP.NET Core + MSAL.NET MSAL Java Flask + MSAL Python MSAL Node		
<p>Desktop app that calls web APIs</p> <p>- Authorization code flow (with PKCE) - Integrated Windows Authentication - Username/Password</p> <p>Device Code Flow Browserless App</p>	 MSAL.NET MSAL Java MSAL Python MSAL Node	 MSAL.NET MSAL Java MSAL Python MSAL Node	 MSAL.NET MSAL Java MSAL Python MSAL Node	<i>iOS</i> MSAL.objc	
<p>Mobile app that calls web APIs</p> <p>Authorization code flow (with PKCE)</p>	 MSAL.NET MSAL.NET			 MSAL.ObjC	 MSAL.Android

SCENARIO	WINDOWS	LINUX	MAC	IOS	ANDROID
<p>Daemon app</p> <pre> graph LR subgraph Daemon_App [Daemon app] DCF[Client Credentials flow] --> DW[Daemon Web app] DCF --> DDA[Daemon Desktop App] DW --> DAA[Daemon API App] DDA --> DAA end DAA </pre>	 MSAL.NET MSAL Java MSAL Python MSAL Node	 MSAL.NET MSAL Java MSAL Python MSAL Node	 MSAL.NET MSAL Java MSAL Python MSAL Node		
<p>Web API that calls web APIs</p> <pre> graph LR OBF[On behalf of flow] --> AA[API App] AA --> AA </pre>	 ASP.NET Core + MSAL.NET MSAL Java MSAL Python MSAL Node	 ASP.NET Core + MSAL.NET MSAL Java MSAL Python MSAL Node	 ASP.NET Core + MSAL.NET MSAL Java MSAL Python MSAL Node		

For more information, see [Microsoft identity platform authentication libraries](#).

Next steps

- Learn more about [authentication basics](#) and [access tokens](#) in the Microsoft identity platform.
- Learn more about [securing access to IoT apps](#).

Security tokens

4/12/2022 • 4 minutes to read • [Edit Online](#)

A centralized identity provider is especially useful for apps that have users located around the globe who don't necessarily sign in from the enterprise's network. The Microsoft identity platform authenticates users and provides security tokens, such as [access tokens](#), [refresh tokens](#), and [ID tokens](#). Security tokens allow a [client application](#) to access protected resources on a [resource server](#).

Access token: An access token is a security token that's issued by an [authorization server](#) as part of an [OAuth 2.0](#) flow. It contains information about the user and the resource for which the token is intended. The information can be used to access web APIs and other protected resources. Access tokens are validated by resources to grant access to a client app. To learn more about how the Microsoft identity platform issues access tokens, see [Access tokens](#).

Refresh token: Because access tokens are valid for only a short period of time, authorization servers will sometimes issue a refresh token at the same time the access token is issued. The client application can then exchange this refresh token for a new access token when needed. To learn more about how the Microsoft identity platform uses refresh tokens to revoke permissions, see [Refresh tokens](#).

ID token: ID tokens are sent to the client application as part of an [OpenID Connect](#) flow. They can be sent alongside or instead of an access token. ID tokens are used by the client to authenticate the user. To learn more about how the Microsoft identity platform issues ID tokens, see [ID tokens](#).

NOTE

This article discusses security tokens used by the OAuth2 and OpenID Connect protocols. Many enterprise applications use SAML to authenticate users. For information on SAML assertions, see [Azure Active Directory SAML token reference](#).

Validate security 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 authorization server with a private key. The authorization server publishes the corresponding public key. To validate a token, the app verifies the signature by using the authorization server public key to validate that the signature was created using the private key.

Tokens are valid for only a limited amount of time. Usually, the authorization server provides a pair of tokens, such as:

- An access token, which accesses the application or protected resource.
- A refresh token, which is 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 authorization server. 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 authorization server receives the refresh token, it won't issue another valid access token if the user is no longer authorized.

JSON Web Tokens and claims

The Microsoft identity platform implements security tokens as JSON Web Tokens (JWTs) that contain *claims*. Since JWTs are used as security tokens, this form of authentication is sometimes called *JWT authentication*.

A [claim](#) provides assertions about one entity, such as a client application or [resource owner](#), to another entity, such as a resource server. A claim might also be referred to as a JWT claim or a JSON Web Token claim.

Claims are name or value pairs that relay facts about the token subject. For example, a claim might contain facts about the security principal that was authenticated by the authorization server. The claims present in a specific token depend on many things, such as the type of token, the type of credential used to authenticate the subject, and the application configuration.

Applications can use claims for various tasks, such as to:

- Validate the token.
- Identify the token subject's [tenant](#).
- Display user information.
- Determine the subject's authorization.

A claim consists of key-value pairs that provide information such as the:

- Security Token Server that generated the token.
- Date when the token was generated.
- Subject (such as the user--except for daemons).
- Audience, which is the app for which the token was generated.
- App (the client) that asked for the token. In the case of web apps, this app might be the same as the audience.

To learn more about how the Microsoft identity platform implements tokens and claim information, see [Access tokens](#) and [ID tokens](#).

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 various tokens (ID tokens, refresh tokens, access tokens) and authorization codes. They require different tokens to make them work. This table 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 because they're 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's too long. As a result, these tokens don't have `groups` or `wids` claims.

Next steps

For more information about authentication and authorization in the Microsoft identity platform, see the following articles:

- To learn about the basic concepts of authentication and authorization, see [Authentication vs. authorization](#).
- To learn about registering your application for integration, see [Application model](#).
- To learn about the sign-in flow of web, desktop, and mobile apps, see [App sign-in flow](#).

Overview of the Microsoft Authentication Library (MSAL)

4/12/2022 • 2 minutes to read • [Edit Online](#)

The Microsoft Authentication Library (MSAL) enables developers to acquire [tokens](#) from the Microsoft identity platform in order to authenticate users and access secured web APIs. It can be used to provide secure access to Microsoft Graph, other Microsoft APIs, third-party web APIs, or your own web API. MSAL supports many different application architectures and platforms including .NET, JavaScript, Java, Python, Android, and iOS.

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

- No need to directly use the OAuth libraries or code against the protocol in your application.
- Acquires tokens on behalf of a user or on behalf of an application (when applicable to the platform).
- Maintains a token cache and refreshes tokens for you when they are close to expire. You don't need to handle token expiration on your own.
- Helps you specify which audience you want your application to sign in (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
MSAL for Android	Android
MSAL Angular	Single-page apps with Angular and Angularjs frameworks

LIBRARY	SUPPORTED PLATFORMS AND FRAMEWORKS
MSAL for iOS and macOS	iOS and macOS
MSAL Go (Preview)	Windows, macOS, Linux
MSAL Java	Windows, macOS, Linux
MSAL.js	JavaScript/TypeScript frameworks such as Vue.js, Ember.js, or Durandal.js
MSAL.NET	.NET Framework, .NET Core, Xamarin Android, Xamarin iOS, Universal Windows Platform
MSAL Node	Web apps with Express, desktop apps with Electron, Cross-platform console apps
MSAL Python	Windows, macOS, Linux
MSAL React	Single-page apps with React and React-based libraries (Next.js, Gatsby.js)

Migrate apps that use ADAL to MSAL

Active Directory Authentication Library (ADAL) integrates with the Azure AD for developers (v1.0) endpoint, where MSAL integrates with the Microsoft identity platform. 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 information about how to migrate to MSAL, see [Migrate applications to the Microsoft Authentication Library \(MSAL\)](#).

Microsoft Graph API

4/12/2022 • 2 minutes to read • [Edit Online](#)

The Microsoft Graph API is a RESTful web API that enables you to access Microsoft Cloud service resources. After you register your app and get authentication tokens for a user or service, you can make requests to the Microsoft Graph API. For more information, see [Overview of Microsoft Graph](#).

Microsoft Graph exposes REST APIs and client libraries to access data on the following Microsoft cloud services:

- Microsoft 365 services: Delve, Excel, Microsoft Bookings, Microsoft Teams, OneDrive, OneNote, Outlook/Exchange, Planner, and SharePoint
- Enterprise Mobility and Security services: Advanced Threat Analytics, Advanced Threat Protection, Azure Active Directory, Identity Manager, and Intune
- Windows 10 services: activities, devices, notifications
- Dynamics 365 Business Central

Versions

The following versions of the Microsoft Graph API are currently available:

- **Beta version:** The beta version includes APIs that are currently in preview and are accessible in the `https://graph.microsoft.com/beta` endpoint. To start using the beta APIs, see [Microsoft Graph beta endpoint reference](#)
- **v1.0 version:** The v1.0 version includes APIs that are generally available and ready for production use. The v1.0 version is accessible in the `https://graph.microsoft.com/v1.0` endpoint. To start using the v1.0 APIs, see [Microsoft Graph REST API v1.0 reference](#)

For more information about Microsoft Graph API versions, see [Versioning, support, and breaking change policies for Microsoft Graph](#).

Get started

To read from or write to a resource such as a user or an email message, you construct a request that looks like the following pattern:

```
{HTTP method} https://graph.microsoft.com/{version}/{resource}?{query-parameters}
```

For more information about the elements of the constructed request, see [Use the Microsoft Graph API](#)

Quickstart samples are available to show you how to access the power of the Microsoft Graph API. The samples that are available access two services with one authentication: Microsoft account and Outlook. Each quickstart accesses information from Microsoft account users' profiles and displays events from their calendar. The quickstarts involve four steps:

- Select your platform
- Get your app ID (client ID)
- Build the sample
- Sign in, and view events on your calendar

When you complete the quickstart, you have an app that's ready to run. For more information, see the [Microsoft Graph quickstart FAQ](#). To get started with the samples, see [Microsoft Graph QuickStart](#).

Tools

Microsoft Graph Explorer is a web-based tool that you can use to build and test requests to the Microsoft Graph API. Access Microsoft Graph Explorer at <https://developer.microsoft.com/graph/graph-explorer>.

Postman is another tool you can use for making requests to the Microsoft Graph API. You can download Postman at <https://www.getpostman.com>. To interact with Microsoft Graph in Postman, use the [Microsoft Graph Postman collection](#).

Next steps

For more information about Microsoft Graph, including usage information and tutorials, see:

- [Use the Microsoft Graph API](#)
- [Microsoft Graph tutorials](#)

Microsoft identity platform developer glossary

4/12/2022 • 17 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 only valid for a short period of time and cannot be revoked. An authorization server may also issue a [refresh token](#) when the access token is issued. Refresh tokens are typically provided only to confidential client applications.

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 the [access tokens reference](#) for more details.

Actor

Another term for the [client application](#) - this is the party acting on behalf of the subject, or [resource owner](#).

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 the Microsoft identity platform application integration, the 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 the [Microsoft identity platform token reference](#) for more details.

Client application

Also known as the "[actor](#)". As defined by the [OAuth2 Authorization Framework](#), an application that makes protected resource requests on behalf of the [resource owner](#). They receive permissions from the resource owner in the form of scopes. 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 the [ID token reference](#) for more details.

Managed identities

Eliminate the need for developers to manage credentials. Managed identities provide an identity for applications to use when connecting to resources that support Azure AD authentication. Applications may use the managed identity to obtain Azure AD tokens. For example, an application may use a managed identity to access resources like Azure Key Vault where developers can store credentials in a secure manner or to access storage accounts. For more information, see [managed identities overview](#).

Microsoft identity platform

The 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.

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](#). These indicate the permission granted to the [actor](#) by the [subject](#).
- "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. These indicate permissions granted to the [subject](#) by the tenant.

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.

Refresh token

A type of [security token](#) issued by an [authorization server](#), and used by a [client application](#) in order to request a new [access token](#) before the access token expires. Typically in the form of a [JSON Web Token \(JWT\)](#).

Unlike access tokens, refresh tokens can be revoked. If a client application attempts to request a new access token using a refresh token that has been revoked, the authorization server will deny the request, and the client application will no longer have permission to access the [resource server](#) on behalf of the [resource owner](#).

See the [refresh tokens](#) for more details.

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. The "resource owner" is also sometimes called the [subject](#).

Every [security token](#) represents a resource owner. The resource owner is what the subject [claim](#), object ID claim, and personal data in the token represent. Resource owners are the party that grants delegated permissions to a client application, in the form of scopes. Resource owners are also the recipients of [roles](#) that indicate expanded permissions within a tenant or on an application.

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 [Microsoft Graph API](#), which provides access to Azure AD tenant data, and the Microsoft 365 APIs that provide access to data such as mail and calendar.

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 Microsoft Graph API, have pre-registered service principals made available in all tenants during provisioning.

Roles

Like [scopes](#), app roles provide a way for a [resource server](#) to govern access to its protected resources. Unlike scopes, roles represent privileges that the [subject](#) has been granted beyond the baseline - this is why reading your own email is a scope, while being an email administrator that can read everyone's email is a role.

App roles can support two assignment types: "user" assignment implements role-based access control for users/groups that require access to the resource, while "application" assignment implements the same for [client applications](#) that require access. An app role can be defined as user-assignable, app-assignable, or both.

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" assignable roles, and configure client [application permissions](#) to request "application" assignable roles.

For a detailed discussion of the application roles exposed by the Microsoft Graph API, see [Graph API Permission Scopes](#). For a step-by-step implementation example, see [Add or remove Azure role assignments using 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 Microsoft Graph API, see [Graph API Permission Scopes](#). For scopes exposed by Microsoft 365 services, see [Microsoft 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), [refresh token](#), and an [ID Token](#) are types of security tokens, all 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](#)

Subject

Also known as the [resource owner](#).

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 Microsoft 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 [Associate or add an Azure subscription to your Azure Active Directory tenant](#) for details on the relationship between subscriptions and an Azure AD tenant, and for instructions on how to associate or add a subscription to 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 Microsoft Graph [User resource type](#) 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](#).

Workload identity

An identity used by a software workload (such as an application, service, script, or container) to authenticate and access other services and resources. In Azure AD, workload identities are apps, service principals, and managed identities. For more information, see [workload identity overview](#).

Workload identity federation

Allows you to securely access Azure AD protected resources from external apps and services without needing to

manage secrets (for supported scenarios). For more information, see [workload identity federation](#).)

Next steps

The [Microsoft identity platform Developer's Guide](#) is the landing page to use for all the Microsoft identity platform development-related topics, including an overview of [application integration](#) and the basics of the [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!

Authorization basics

4/12/2022 • 4 minutes to read • [Edit Online](#)

Authorization (sometimes abbreviated as *AuthZ*) is used to set permissions that are used to evaluate access to resources or functionality. In contrast, **authentication** (sometimes abbreviated as *AuthM*) is focused on proving that an entity like a user or service is indeed who they claim to be.

Authorization can include specifying what functionality (or resources) an entity is allowed to access or what data that entity can access and what they can do with that data. This is often referred to as *access control*.

NOTE

Authentication and authorization are concepts that are not limited to only users. Services or daemon applications are often built to make requests for resources as themselves rather than on behalf of a specific user. When discussing these topics, the term "entity" is used to refer to either a user or an application.

Authorization approaches

There are several common approaches to handle authorization. [Role-based access control](#) is currently the most common approach using Microsoft identity platform.

Authentication as authorization

Possibly the simplest form of authorization is to grant or deny access based on whether the entity making a request has been authenticated. If the requestor can prove they're who they claim to be, they can access the protected resources or functionality.

Access control lists

Authorization via access control lists (ACLs) involves maintaining explicit lists of specific entities who do or don't have access to a resource or functionality. ACLs offer finer control over authentication-as-authorization but become difficult to manage as the number of entities increases.

Role-based access control

Role-based access control (RBAC) is possibly the most common approach to enforcing authorization in applications. When using RBAC, roles are defined to describe the kinds of activities an entity may perform. An application developer grants access to roles rather than to individual entities. An administrator can then assign roles to different entities to control which ones have access to what resources and functionality.

In advanced RBAC implementations, roles may be mapped to collections of permissions, where a permission describes a granular action or activity that can be performed. Roles are then configured as combinations of permissions. You compute the entities' overall permission set for an application by intersecting the permissions granted to the various roles the entity is assigned. A good example of this approach is the RBAC implementation that governs access to resources in Azure subscriptions.

NOTE

[Application RBAC](#) differs from [Azure RBAC](#) and [Azure AD RBAC](#). Azure custom roles and built-in roles are both part of Azure RBAC, which helps you manage Azure resources. Azure AD RBAC allows you to manage Azure AD resources.

Attribute-based access control

Attribute-based access control (ABAC) is a more fine-grained access control mechanism. In this approach, rules

are applied to attributes of the entity, the resources being accessed, and the current environment to determine whether access to some resources or functionality is permitted. An example might be only allowing users who are managers to access files identified with a metadata tag of "managers during working hours only" during the hours of 9AM - 5PM on working days. In this case, access is determined by examining the user's attribute (status as manager), the resource's attribute (metadata tag on a file), and also an environment attribute (the current time).

One advantage of ABAC is that more granular and dynamic access control can be achieved through rule and condition evaluations without the need to create large numbers of very specific roles and RBAC assignments.

One method for achieving ABAC with Azure Active Directory is using [dynamic groups](#). Dynamic groups allow administrators to dynamically assign users to groups based on specific user attributes with desired values. For example, an Authors group could be created where all users with the job title Author are dynamically assigned to the Authors group. Dynamic groups can be used in combination with RBAC for authorization where you map roles to groups and dynamically assign users to groups.

[Azure ABAC](#) is an example of an ABAC solution that is available today. Azure ABAC builds on Azure RBAC by adding role assignment conditions based on attributes in the context of specific actions.

Implementing authorization

Authorization logic is often implemented within the applications or solutions where access control is required. In many cases, application development platforms offer middleware or other API solutions that simplify the implementation of authorization. Examples include use of the [AuthorizeAttribute](#) in ASP.NET or [Route Guards](#) in Angular.

For authorization approaches that rely on information about the authenticated entity, an application will evaluate information exchanged during authentication. For example, by using the information that was provided within a [security token](#)). For information not contained in a security token, an application might make extra calls to external resources.

It's not strictly necessary for developers to embed authorization logic entirely within their applications. Instead, dedicated authorization services can be used to centralize authorization implementation and management.

Next steps

- To learn about custom role-based access control implementation in applications, see [Role-based access control for application developers](#).
- To learn about the process of registering your application so it can integrate with the Microsoft identity platform, see [Application model](#).
- For an example of configuring simple authentication-based authorization, see [Configure your App Service or Azure Functions app to use Azure AD login](#).

Role-based access control for application developers

4/12/2022 • 6 minutes to read • [Edit Online](#)

Role-based access control (RBAC) allows certain users or groups to have specific permissions regarding which resources they have access to, what they can do with those resources, and who manages which resources. Application role-based access control differs from [Azure role-based access control](#) and [Azure AD role-based access control](#). Azure custom roles and built-in roles are both part of Azure RBAC, which helps you manage Azure resources. Azure AD RBAC allows you to manage Azure AD resources. This article explains application-specific role-based access control.

What are roles?

Role-based access control (RBAC) is a popular mechanism to enforce authorization in applications. When using RBAC, an application developer defines roles rather than authorizing individual users or groups. An administrator can then assign roles to different users and groups to control who has access to what content and functionality.

RBAC helps you, as an app developer, manage resources and what users can do with those resources. RBAC also allows an app developer to control what areas of an app users have access to. While admins can control which users have access to an app using the *User assignment required* property, developers need to account for specific users within the app and what users can do within the app.

As an app developer, you need to first create a role definition within the app's registration section in the Azure AD admin center. The role definition includes a value that is returned for users who are assigned to that role. A developer can then use this value to implement application logic to determine what those users can or can't do in an application.

Options for adding RBAC to apps

There are several considerations that must be managed when including role-based access control authorization in an application. These include:

- Defining the roles that are required by an application's authorization needs.
- Applying, storing, and retrieving the pertinent roles for authenticated users.
- Affecting the desired application behavior based on the roles assigned to the current user.

Once you define the roles, the Microsoft identity platform supports several different solutions that can be used to apply, store, and retrieve role information for authenticated users. These solutions include app roles, Azure AD groups, and the use of custom datastores for user role information.

Developers have the flexibility to provide their own implementation for how role assignments are to be interpreted as application permissions. This can involve leveraging middleware or other functionality provided by their applications' platform or related libraries. Apps will typically receive user role information as claims and will decide user permissions based on those claims.

App roles

Azure AD supports declaring app roles for an application registration. When a user signs into an application, Azure AD will include a [roles claim](#) for each role that the user has been granted for that application. Applications that receive tokens that contain these claims can then use this information to determine what permissions the

user may exercise based on the roles they're assigned.

Groups

Developers can also use [Azure AD groups](#) to implement RBAC in their applications, where the users' memberships in specific groups are interpreted as their role memberships. When using Azure AD groups, Azure AD will include a [groups claim](#) that will include the identifiers of all of the groups to which the user is assigned within the current Azure AD tenant. Applications that receive tokens that contain these claims can then use this information to determine what permissions the user may exercise based on the roles they're assigned.

IMPORTANT

When working with groups, developers need to be aware of the concept of an [overage claim](#). By default, if a user is a member of more than the overage limit (150 for SAML tokens, 200 for JWT tokens, 6 if using the implicit flow), Azure AD will not emit a groups claim in the token. Instead, it will include an "overage claim" in the token that indicates the token's consumer will need to query the Graph API to retrieve the user's group memberships. For more information about working with overage claims, see [Claims in access tokens](#). It is possible to only emit groups that are assigned to an application, though [group-based assignment](#) does require Azure Active Directory Premium P1 or P2 edition.

Custom data store

App roles and groups both store information about user assignments in the Azure AD directory. Another option for managing user role information that is available to developers is to maintain the information outside of the directory in a custom data store. For example, in a SQL Database, Azure Table storage or Azure Cosmos DB Table API.

Using custom storage allows developers extra customization and control over how to assign roles to users and how to represent them. However, the extra flexibility also introduces more responsibility. For example, there's no mechanism currently available to include this information in tokens returned from Azure AD. If developers maintain role information in a custom data store, they'll need to have the apps retrieve the roles. This is typically done using extensibility points defined in the middleware available to the platform that is being used to develop the application. Furthermore, developers are responsible for properly securing the custom data store.

Using [Azure AD B2C Custom policies](#) it is possible to interact with custom data stores and to include custom claims within a token.

Choosing an approach

In general, app roles are the recommended solution. App roles provide the simplest programming model and are purpose made for RBAC implementations. However, specific application requirements may indicate that a different approach would be better solution.

Developers can use app roles to control whether a user can sign into an app, or an app can obtain an access token for a web API. App roles are preferred over Azure AD groups by developers when they want to describe and control the parameters of authorization in their app themselves. For example, an app using groups for authorization will break in the next tenant as both the group ID and name could be different. An app using app roles remains safe. In fact, assigning groups to app roles is popular with SaaS apps for the same reasons.

Although either app roles or groups can be used for authorization, key differences between them can influence which is the best solution for a given scenario.

APP ROLES	AZURE AD GROUPS	CUSTOM DATA STORE
-----------	-----------------	-------------------

	APP ROLES	AZURE AD GROUPS	CUSTOM DATA STORE
Programming model	Simplest. They are specific to an application and are defined in the app registration. They move with the application.	More complex. Group IDs vary between tenants and overage claims may need to be considered. Groups aren't specific to an app, but to an Azure AD tenant.	Most complex. Developers must implement means by which role information is both stored and retrieved.
Role values are static between Azure AD tenants	Yes	No	Depends on the implementation.
Role values can be used in multiple applications	No. Unless role configuration is duplicated in each app registration.	Yes	Yes
Information stored within directory	Yes	Yes	No
Information is delivered via tokens	Yes (roles claim)	Yes (In the case of an overage, <i>groups claims</i> may need to be retrieved at runtime)	No. Retrieved at runtime via custom code.
Lifetime	Lives in app registration in directory. Removed when the app registration is removed.	Lives in directory. Remain intact even if the app registration is removed.	Lives in custom data store. Not tied to app registration.

Next steps

- [How to add app roles to your application and receive them in the token.](#)
- [Register an application with the Microsoft identity platform.](#)
- [Azure Identity Management and access control security best practices.](#)

Permissions and consent in the Microsoft identity platform

4/12/2022 • 26 minutes to read • [Edit Online](#)

Applications that integrate with the 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. 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.

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*.

Here are some examples of Microsoft web-hosted resources:

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

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

Because of these types of permission definitions, 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.

When a resource's functionality is chunked into small permission sets, third-party apps can be built to request only the permissions that they need to perform their function. Users and administrators can know what data the app can access. And they can be more confident that the app isn't behaving with malicious intent. Developers should always abide by the principle of least privilege, asking for only the permissions they need for their applications to function.

In OAuth 2.0, these types of permission sets are called *scopes*. They're also often referred to as *permissions*. In the Microsoft identity platform, a permission is represented as a string value. An app requests the permissions it needs by specifying the permission in the `scope` query parameter. Identity platform supports several well-defined [OpenID Connect scopes](#) as well as resource-based permissions (each permission is indicated by appending the permission value to the resource's identifier or application ID URI). For example, the permission string `https://graph.microsoft.com/Calendars.Read` is used to request permission to read users' calendars in Microsoft Graph.

An app most commonly requests these permissions by specifying the scopes in requests to the Microsoft

identity platform authorize endpoint. However, some high-privilege permissions can be granted only through administrator consent. They can be requested or granted by using the [administrator consent endpoint](#). Keep reading to learn more.

In requests to the authorization, token or consent endpoints for the Microsoft Identity platform, if the resource identifier is omitted in the scope parameter, the resource is assumed to be Microsoft Graph. For example,

`scope=User.Read` is equivalent to `https://graph.microsoft.com/User.Read`.

Permission types

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

Some delegated permissions can be consented to by nonadministrators. But some high-privileged permissions require [administrator consent](#). To learn which administrator roles can consent to delegated permissions, see [Administrator role permissions in Azure Active Directory \(Azure AD\)](#).

- **Application permissions** are used by apps that run without a signed-in user present, for example, apps that run as background services or daemons. Only [an administrator can consent to application permissions](#).

Effective permissions are the permissions that your app has when it makes requests to the target resource. It's important to understand the difference between the delegated permissions and application permissions that your app is granted, and the effective permissions your app is granted when it makes calls to the target resource.

- For delegated permissions, the *effective permissions* of your app are the least-privileged intersection of the delegated permissions the app has been granted (by 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 can 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 can update the profile of every user in the organization. However, if the signed-in user doesn't have an administrator role, your app can update only the profile of the signed-in user. It can't update the profiles of other users in the organization because the user that it has permission to act on behalf of doesn't 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.

OpenID Connect scopes

The Microsoft identity platform implementation of OpenID Connect has a few well-defined scopes that are also hosted on Microsoft Graph: `openid`, `email`, `profile`, and `offline_access`. The `address` and `phone` OpenID Connect scopes aren't supported.

If you request the OpenID Connect scopes and a token, you'll get a token to call the [UserInfo endpoint](#).

openid

If an app signs in by using [OpenID Connect](#), it must request the `openid` scope. The `openid` scope appears on the work account consent page as the **Sign you in** permission. On the personal Microsoft account consent page, it appears as the **View your profile and connect to apps and services using your Microsoft account** permission.

By using this permission, an app can receive a unique identifier for the user in the form of the `sub` claim. The permission 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. The app can use these tokens for authentication.

email

The `email` scope can be used with the `openid` scope and any other scopes. 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 your app uses the `email` scope, the app needs to be able to handle a case in which no `email` claim exists in the token.

profile

The `profile` scope can be used with the `openid` scope and any other scope. It gives the app access to a large 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.

NOTE

This permission currently appears on all consent pages, even for flows that don't provide a refresh token (such as the [implicit flow](#)). This setup addresses scenarios where a client can begin within the implicit flow and then move to the code flow where a refresh token is expected.

On the Microsoft identity platform (requests made to the v2.0 endpoint), your app must explicitly request the `offline_access` scope, to receive refresh tokens. So 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. It 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.

For more information about how to get and use refresh tokens, see the [Microsoft identity platform protocol reference](#).

Consent types

Applications in Microsoft identity platform rely on consent in order to gain access to necessary resources or APIs. There are a number of kinds of consent that your app may need to know about in order to be successful. If

you are defining permissions, you will also need to understand how your users will gain access to your app or API.

Static user consent

In the static user consent scenario, you must specify all the permissions it needs in the app's configuration in the Azure portal. If the user (or administrator, as appropriate) has not granted consent for this app, then Microsoft identity platform will prompt the user to provide consent at this time.

Static permissions also enables administrators to [consent on behalf of all users](#) in the organization.

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

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

Incremental and dynamic user consent

With the Microsoft identity platform endpoint, you can ignore the static permissions defined in the app registration information in the Azure portal and request permissions incrementally instead. You can ask for a bare minimum set of permissions upfront and request more over time as the customer uses additional app features. To do so, you can specify the scopes your app needs at any time by including the new scopes in the `scope` parameter when [requesting an access token](#) - without the need to pre-define them in the application registration information. If the user hasn't yet consented to new scopes added to the request, they'll be prompted to consent only to the new permissions. Incremental, or dynamic consent, only applies to delegated permissions and not to application permissions.

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.

IMPORTANT

Dynamic consent can be convenient, but presents a big challenge for permissions that require admin consent. The admin consent experience in the **App registrations** and **Enterprise applications** blades in the portal doesn't know about those dynamic permissions at consent time. We recommend that a developer list all the admin privileged permissions that are needed by the app in the portal. This enables tenant admins to consent on behalf of all their users in the portal, once. Users won't need to go through the consent experience for those permissions on sign in. The alternative is to use dynamic consent for those permissions. To grant admin consent, an individual admin signs in to the app, triggers a consent prompt for the appropriate permissions, and selects **consent for my entire org** in the consent dialogue.

Admin consent

[Admin consent](#) is required when your app needs access to certain high-privilege permissions. Admin consent ensures that administrators have some additional controls before authorizing apps or users to access highly privileged data from the organization.

[Admin consent done on behalf of an organization](#) is highly recommended if your app has an enterprise audience. Admin consent done on behalf of an organization requires the static permissions to be registered for the app in the portal. Set those permissions for apps in the app registration portal if you need an admin to give consent on behalf of the entire organization. The admin can consent to those permissions on behalf of all users in the org, once. The users will not need to go through the consent experience for those permissions when

signing in to the app. This is easier for users and reduces the cycles required by the organization admin to set up the application.

Requesting individual user consent

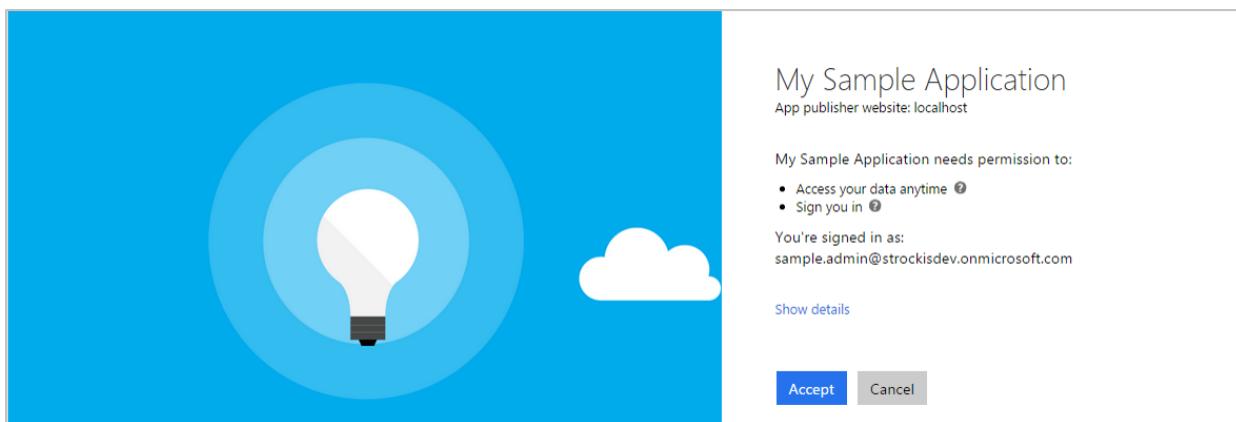
In an [OpenID Connect or OAuth 2.0](#) authorization request, an app can request the permissions it needs by using the `scope` query parameter. For example, when a user signs in to an app, the app sends a request like the following example. (Line breaks are added for legibility.)

```
GET https://login.microsoftonline.com/common/oauth2/v2.0/authorize?  
client_id=6731de76-14a6-49ae-97bc-6eba6914391e  
&response_type=code  
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F  
&response_mode=query  
&scope=  
https%3A%2F%2Fgraph.microsoft.com%2Fcalendars.read%20  
https%3A%2F%2Fgraph.microsoft.com%2Fmail.send  
&state=12345
```

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

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

At this time, the `offline_access` ("Maintain access to data you have given it access to") permission and `User.Read` ("Sign you in and read your profile") permission are automatically included in the initial consent to an application. These permissions are generally required for proper app functionality. The `offline_access` permission gives the app access to refresh tokens that are critical for native apps and web apps. The `User.Read` permission gives access to the `sub` claim. It allows 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. The user doesn't have to consent again when they later sign in to the application.

Requesting consent for an entire tenant

When an organization purchases a license or subscription for an application, the organization often 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 don't see a consent page for the application.

Admin consent done on behalf of an organization requires the static permissions registered for the app. Set those permissions for apps in the app registration portal if you need an admin to give consent on behalf of the entire organization.

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 Microsoft resources can be set to *admin-restricted*. Here are some examples of these kinds of permissions:

- 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`

NOTE

In requests to the authorization, token or consent endpoints for the Microsoft Identity platform, if the resource identifier is omitted in the scope parameter, the resource is assumed to be Microsoft Graph. For example, `scope=User.Read` is equivalent to <https://graph.microsoft.com/User.Read>.

Although a consumer user might grant an application access to this kind of data, organizational users can't grant 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 scopes for admin-restricted permissions, an organization's administrator must consent to those scopes on behalf of the organization's users. To avoid displaying prompts to users that request consent for permissions they can't grant, your app can use the admin consent endpoint. The admin consent endpoint is covered in the next section.

If the application requests high-privilege delegated permissions and an administrator grants these permissions through the admin consent endpoint, consent is granted for all users in the tenant.

If the application requests application permissions and an administrator grants these permissions through 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 used only by daemon services and other noninteractive applications that run in the background.

Using the admin consent endpoint

After you use the admin consent endpoint to grant admin consent, you're finished. Users don't need to take any further action. After admin consent is granted, users can get an access token through a typical auth flow. The resulting access token has the consented permissions.

When a Global Administrator uses your application and is directed to the authorize endpoint, the Microsoft identity platform detects the user's role. It asks if the Global Administrator wants to consent on behalf of the entire tenant for the permissions you requested. You could instead use a dedicated admin consent endpoint to proactively request an administrator to grant permission on behalf of the entire tenant. This endpoint is also necessary for requesting application permissions. Application permissions can't be requested by 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 operation is high privilege. Use the operation only if necessary for your scenario.

To see a code sample that implements the steps, see the [admin-restricted scopes sample](#) in GitHub.

Request the permissions in the app registration portal

In the app registration portal, applications can list the permissions they require, including both delegated permissions and application permissions. This setup allows the use of the `.default` scope and the Azure portal's **Grant admin consent** option.

In general, the permissions should be statically defined for a given application. They should be a superset of the permissions that the app will request dynamically or incrementally.

NOTE

Application permissions can be requested only through the use of `.default`. So if your app needs application permissions, make sure they're listed in the app registration portal.

To configure the list of statically requested permissions for an application:

1. Go to your application in the [Azure portal - App registrations](#) quickstart experience.
2. Select an application, or [create an app](#) if you haven't already.
3. On the application's **Overview** page, under **Manage**, select **API Permissions > Add a permission**.
4. Select **Microsoft Graph** from the list of available APIs. Then add the permissions that your app requires.
5. Select **Add Permissions**.

Recommended: Sign the user in to 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.
- 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 Microsoft account or school Microsoft account.

When you sign the user in to your app, you can identify the organization to which the admin belongs before you ask them to approve the necessary permissions. Although this step isn't strictly necessary, it can help you create a more intuitive experience for your organizational users.

To sign the user in, follow the [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. It can be provided in a GUID or friendly name format. Or it can be generically referenced with organizations, as seen in the example. Don't use "common," because personal accounts can't provide admin consent except in the context of a tenant. To ensure the best compatibility with personal accounts that manage tenants, use the tenant ID when possible.
<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. Scopes can be either static (using <code>.default</code>) or dynamic. This set can include the OpenID Connect scopes (<code>openid</code> , <code>profile</code> , <code>email</code>). If you need application permissions, you must use <code>.default</code> to request the statically configured list of permissions.

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 requested in the `scope` parameter. If you 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
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> .

Error response

If the admin doesn't 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
error	An error code string that can be used to classify types of errors that occur. It can also be used to react to errors.
error_description	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 the 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 do 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

The `.default` scope is used to refer generically to a resource service (API) in a request, without identifying specific permissions. If consent is necessary, using `.default` signals that consent should be prompted for all required permissions listed in the application registration (for all APIs in the list).

The scope parameter value is constructed by using the identifier URI for the resource and `.default`, separated by a forward slash (`/`). For example, if the resource's identifier URI is `https://contoso.com`, the scope to request is `https://contoso.com/.default`. For cases where you must include a second slash to correctly request the token, see the [section about trailing slashes](#).

Using `scope={resource-identifier}/.default` is functionally the same as `resource={resource-identifier}` on the v1.0 endpoint (where `{resource-identifier}` is the identifier URI for the API, for example `https://graph.microsoft.com` for Microsoft Graph).

The `.default` scope can be used in any OAuth 2.0 flow and to initiate [admin consent](#). Its use is required in the [On-Behalf-Of flow](#) and [client credentials flow](#).

Clients can't combine static (`.default`) consent and dynamic consent in a single request. So

`scope=https://graph.microsoft.com/.default Mail.Read` results in an error because it combines scope types.

.default when the user has already given consent

The `.default` scope is functionally identical to the behavior of the `resource`-centric v1.0 endpoint. It carries the consent behavior of the v1.0 endpoint as well. That is, `.default` triggers a consent prompt only if consent has not been granted for any delegated permission between the client and the resource, on behalf of the signed-in user.

If consent does exist, the returned token contains all scopes granted for that resource for the signed-in user. However, if no permission has been granted for the requested resource (or if the `prompt=consent` parameter has been provided), a consent prompt is shown for all required permissions configured on the client application registration, for all APIs in the list.

For example, if the scope `https://graph.microsoft.com/.default` is requested, your application is requesting an access token for the Microsoft Graph API. If at least one delegated permission has been granted for Microsoft Graph on behalf of the signed-in user, the sign-in will continue and all Microsoft Graph delegated permissions which have been granted for that user will be included in the access token. If no permissions have been granted for the requested resource (Microsoft Graph, in this example), then a consent prompt will be presented for all required permissions configured on the application, for all APIs in the list.

Example 1: The user, or tenant admin, has granted permissions

In this example, the user or a tenant administrator has granted the `Mail.Read` and `User.Read` Microsoft Graph permissions to the client.

If the client requests `scope=https://graph.microsoft.com/.default`, no consent prompt is shown, regardless of the contents of the client application's registered permissions for Microsoft Graph. The returned token contains the scopes `Mail.Read` and `User.Read`.

Example 2: The user hasn't granted permissions between the client and the resource

In this example, the user hasn't granted consent between the client and Microsoft Graph, nor has an administrator. The client has registered for the permissions `User.Read` and `Contacts.Read`. It has also registered for 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 sees a consent page for the Microsoft Graph `User.Read` and `Contacts.Read` scopes, and for the Azure Key Vault `user_impersonation` scope. The returned token contains only the `User.Read` and `Contacts.Read` scopes, and it can be used only against Microsoft Graph.

Example 3: The user has consented, and the client requests more scopes

In this example, the user has already consented to `Mail.Read` for the client. The client has registered for the `Contacts.Read` scope.

The client first performs a sign-in with `scope=https://graph.microsoft.com/.default`. Based on the `scopes` parameter of the response, the application's code detects that only `Mail.Read` has been granted. The client then initiates a second sign-in using `scope=https://graph.microsoft.com/.default`, and this time forces consent using `prompt=consent`. If the user is allowed to consent for all the permissions that the application registered, they will be shown the consent prompt. (If not, they will be shown an error message or the [admin consent request](#) form.) Both `Contacts.Read` and `Mail.Read` will be in the consent prompt. If consent is granted and the sign-in continues, the token returned is for Microsoft Graph, and contains `Mail.Read` and `Contacts.Read`.

Using the `.default` scope with the client

In some cases, a client can request 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 code example produces a consent page for all registered permissions if the preceding descriptions of consent and `.default` apply to the scenario. Then the code returns an `id_token`, rather than an access token.

This behavior accommodates some legacy clients that are moving from Azure AD Authentication Library (ADAL) to the Microsoft Authentication Library (MSAL). This setup *shouldn't* be used by new clients that target the Microsoft identity platform.

Client credentials grant flow and `.default`

Another use of `.default` is to request app roles (also known as application permissions) in a non-interactive application like a daemon app that uses the [client credentials](#) grant flow to call a web API.

To define app roles (application permissions) for a web API, see [Add app roles in your application](#).

Client credentials requests in your client service *must* include `scope={resource}/.default`. Here, `{resource}` is the web API that your app intends to call, and wishes to obtain an access token for. Issuing a client credentials request by using individual application permissions (roles) is *not* supported. All the app roles (application permissions) that have been granted for that web API are included in the returned access token.

To grant access to the app roles you define, including granting admin consent for the application, see [Configure a client application to access a web API](#).

Trailing slash and `.default`

Some resource URIs have a trailing forward slash, for example, `https://contoso.com/` as opposed to `https://contoso.com`. The trailing slash can cause problems with token validation. Problems occur primarily when a token is requested for Azure Resource Manager (`https://management.azure.com/`). In this case, a trailing slash on the resource URI means the slash must be present when the token is requested. So when you request a token for `https://management.azure.com/` and use `.default`, you must request `https://management.azure.com//.default` (notice the double slash!). In general, if you verify that the token is being issued, and if the token is being rejected by the API that should accept it, consider adding a second forward slash and trying again.

Troubleshooting permissions and consent

For troubleshooting steps, see [Unexpected error when performing consent to an application](#).

Next steps

- [ID tokens in the Microsoft identity platform](#)
- [Access tokens in the Microsoft identity platform](#)

Understanding Azure AD application consent experiences

4/12/2022 • 5 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 testuser@fourthcoffeetest.onmicrosoft.com

② Permissions requested



This application is not published by Microsoft or your organization. ⑦

This app would like to:

- ✓ Maintain access to data you have given it access to
- ✗ Sign you in and read your profile

⑨ [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.
This is a permission requested to access your data in Fourth Coffee.] ⑧

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>. ⑩ ⑪

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. [\[Hide details\]](#)

Does this app look suspicious? [Report it here](#) ⑫

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	Publisher verified	The blue "verified" badge means that the app publisher has verified their identity using a Microsoft Partner Network account and has completed the verification process.
7	Publisher information	Displays whether the application is published by Microsoft or your organization.
8	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.
9	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.
10	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.
11	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.

#	COMPONENT	PURPOSE
12	Report it here	This link is used to report a suspicious app if you don't trust the app, if you believe the app is impersonating another app, if you believe the app will misuse your data, or for some other reason.

App requires a permission within the user's scope of authority

A common consent scenario is that the user accesses an app which requires a permission set that is within the user's scope of authority. The user is directed to the user consent flow.

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.



testadmin@fourthcoffeetest.onmicrosoft.com

Permissions requested



Best Practices Demo
microsoftidentity.dev

This application is not published by Microsoft or your organization.

This app would like to:

- ✓ Maintain access to data you have given it access to
- ✓ Sign you in and read your profile
- Consent on behalf of your organization

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](#)

Does this app look suspicious? [Report it here](#)

[Cancel](#)

[Accept](#)

Users will see the traditional consent prompt.



testuser@fourthcoffeetest.onmicrosoft.com

Permissions requested



Best Practices Demo

microsoftidentity.dev

This application is not published by Microsoft or your organization.

This app would like to:

- ✓ Maintain access to data you have given it access to
- ✓ 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](#)

Does this app look suspicious? [Report it here](#)

Cancel

Accept

App requires a permission outside of the user's scope of authority

Another common consent scenario is that the user accesses an app which requires at least one permission that is outside the user's scope of authority.

Admins will see an additional control on the traditional consent prompt that will allow them consent on behalf of the entire tenant.



testadmin@fourthcoffeetest.onmicrosoft.com

Permissions requested



Best Practices Demo
microsoftidentity.dev

This application is not published by Microsoft or your organization.

This app would like to:

- Maintain access to data you have given it access to
- Sign you in and read your profile
- Consent on behalf of your organization

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](#)

Does this app look suspicious? [Report it here](#)

[Cancel](#)

[Accept](#)

Non-admin users will be blocked from granting consent to the application, and they will be told to ask their admin for access to the app.



testuser@fourthcoffeetest.onmicrosoft.com

Need admin approval



Best Practices Demo
microsoftidentity.dev

Best Practices Demo 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](#)

User is directed to the admin consent flow

Another common scenario is when the user navigates to or is directed to the admin consent flow.

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.



testadmin@fourthcoffeetest.onmicrosoft.com

Permissions requested



Best Practices Demo

microsoftidentity.dev



This application is not published by Microsoft or your organization.

This app would like to:

- Read all groups
- Maintain access to data you have given it access to
- View your basic profile
- Consent on behalf of your organization

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](#)

Does this app look suspicious? [Report it here](#)

[Cancel](#)

[Accept](#)

Non-admin users will be blocked from granting consent to the application, and they will be told to ask their admin for access to the app.



testuser@fourthcoffeetest.onmicrosoft.com

Need admin approval



Best Practices Demo

microsoftidentity.dev



Best Practices Demo 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](#)

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](#).

Microsoft identity platform consent framework

4/12/2022 • 2 minutes to read • [Edit Online](#)

Multi-tenant applications allow sign-ins by user accounts from Azure AD tenants other than the tenant in which the app was initially registered. The Microsoft identity platform consent framework enables a tenant administrator or user in these other tenants to consent to (or deny) an application's request for permission to access their resources.

For example, perhaps a web application requires read-only access to a user's calendar in Microsoft 365. It's the identity platform's consent framework that enables the prompt asking the user to consent to the app's request for permission to read their calendar. If the user consents, the application is able to call the Microsoft Graph API on their behalf and get their calendar data.

Consent experience - an example

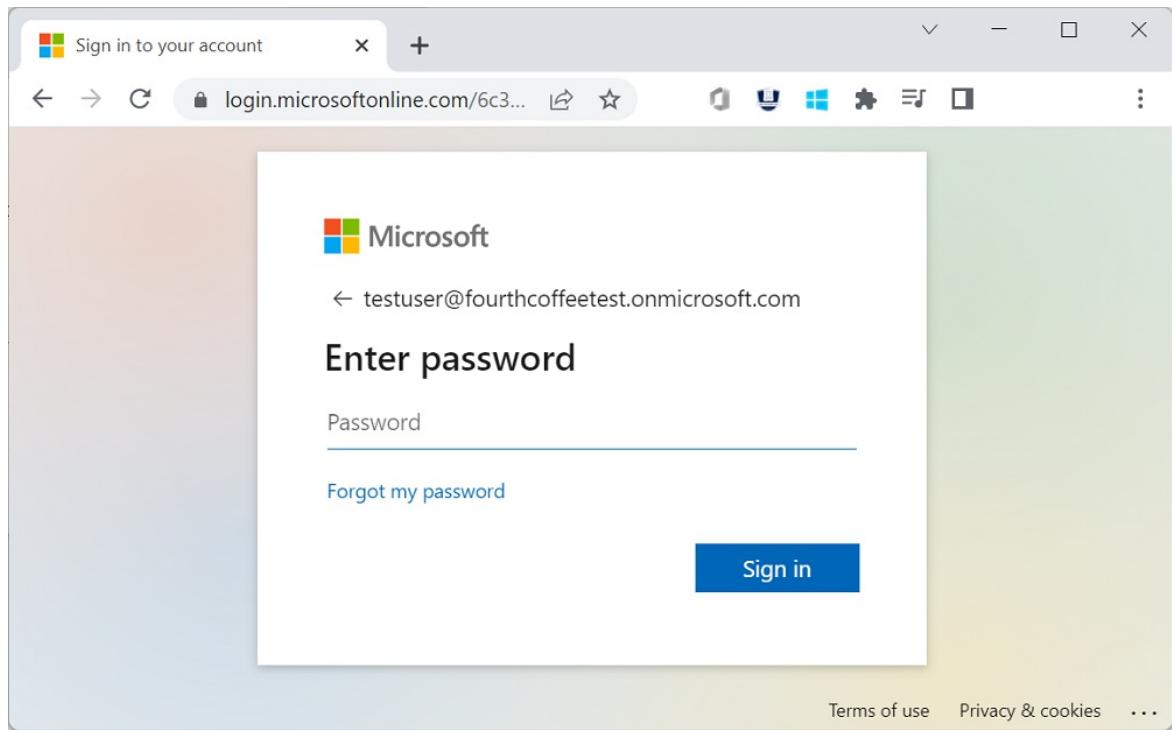
The following steps show you how the consent experience works for both the application developer and the user.

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:

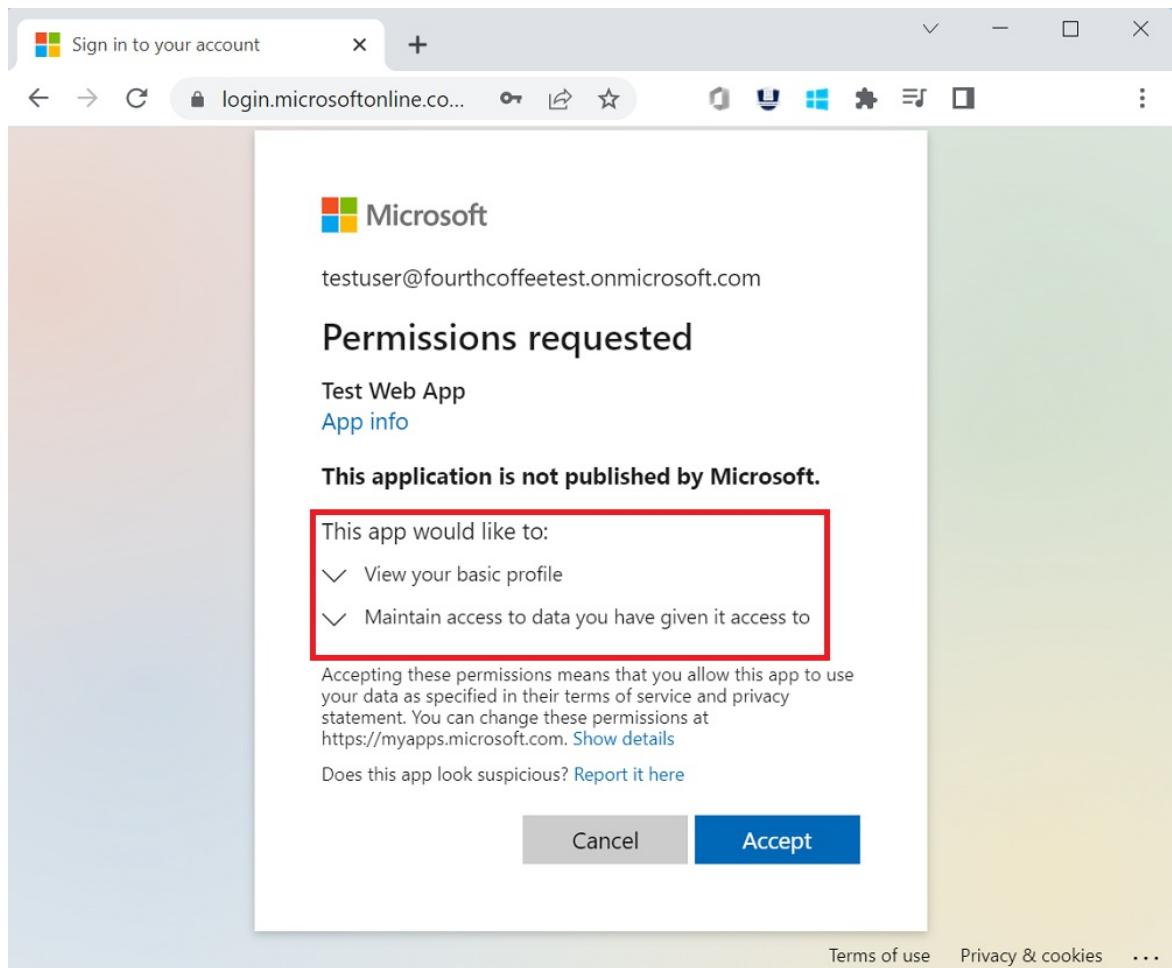
The screenshot shows the Azure portal interface for managing API permissions. The left sidebar has a 'Manage' section with options like Branding & properties, Authentication, Certificates & secrets, Token configuration, API permissions (which is selected), Expose an API, App roles, Owners, and Roles and administrators. The main content area is titled 'Test Web App | API permissions'. It shows a table of configured permissions for 'Microsoft Graph (2)'. The table has columns: API / Permissions name, Type, Description, Admin consent req..., and Status. Two rows are listed: 'Group.Read.All' (Delegated, Read all groups, Yes, Not granted for Fourth...) and 'User.Read' (Delegated, Sign in and read user profile, No). A note at the bottom says 'To view and manage permissions and user consent, try Enterprise applications.'

API / Permissions name	Type	Description	Admin consent req...	Status
Group.Read.All	Delegated	Read all groups	Yes	Not granted for Fourth...
User.Read	Delegated	Sign in and read user profile	No	

2. 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.
3. 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.



- 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 [OAuth 2.0 authorization code flow](#).

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.

The screenshot shows the Azure portal interface for managing API permissions. On the left, there's a sidebar with options like Overview, Quickstart, Integration assistant, Manage (with sub-options: Branding & properties, Authentication, Certificates & secrets, Token configuration, API permissions, Expose an API, App roles, Owners, Roles and administrators), and Home > Test Web App. The 'API permissions' option is highlighted with a red box. The main content area is titled 'Test Web App | API permissions'. It has a search bar, refresh and feedback buttons, and a message about 'Admin consent required'. Below this is a 'Configured permissions' section with a table. The table has columns: API / Permissions name, Type, Description, Grant admin consent req..., and Status. It lists two permissions: 'Group.Read.All' (Delegated, Read all groups, Yes, Not granted for Fourth...) and 'User.Read' (Delegated, Sign in and read user profile, No). At the bottom of the table, there's a note: 'To view and manage permissions and user consent, try Enterprise applications.' A button labeled '+ Add a permission' with a checkmark and the text 'Grant admin consent for Fourth Coffee' is also highlighted with a red box.

API / Permissions name	Type	Description	Grant admin consent req...	Status
Group.Read.All	Delegated	Read all groups	Yes	Not granted for Fourth...
User.Read	Delegated	Sign in and read user profile	No	

IMPORTANT

Granting explicit consent using the **Grant permissions** button is currently required for single-page applications (SPA) that use MSAL.js. Otherwise, the application fails when the access token is requested.

Next steps

See [how to convert an app to multi-tenant](#)

Developer guidance for Azure Active Directory Conditional Access

4/12/2022 • 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 the article [What is Conditional Access](#).

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.

NOTE

Using this feature requires an Azure AD Premium P1 license. To find the right license for your requirements, see [Comparing generally available features of the Free, Basic, and Premium editions](#). Customers with [Microsoft 365 Business licenses](#) also have access to Conditional Access features.

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 MSAL.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. For your app to continue functioning when a new policy is applied, implement 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="ChannelMessages.Read.All Mail.Read"
```

An app can expect their users to fulfill all policies set on Teams 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](#). Customers with [Microsoft 365 Business licenses](#) also have access to Conditional Access features.

Considerations for specific scenarios

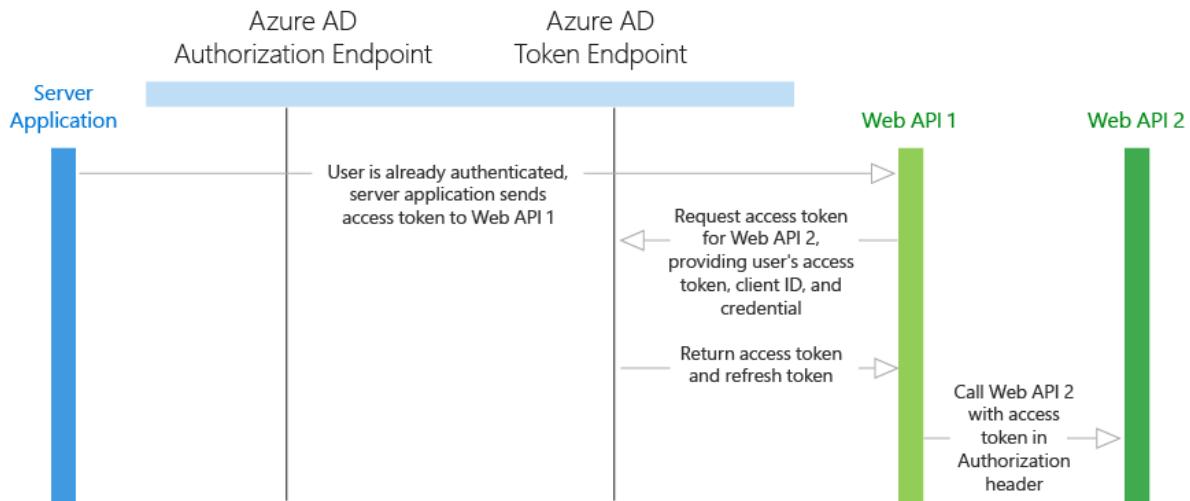
The following information only applies in these Conditional Access scenarios:

- Apps performing the on-behalf-of flow
- Apps accessing multiple services/resources
- Single-page apps using MSAL.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>"]}}}
  
```

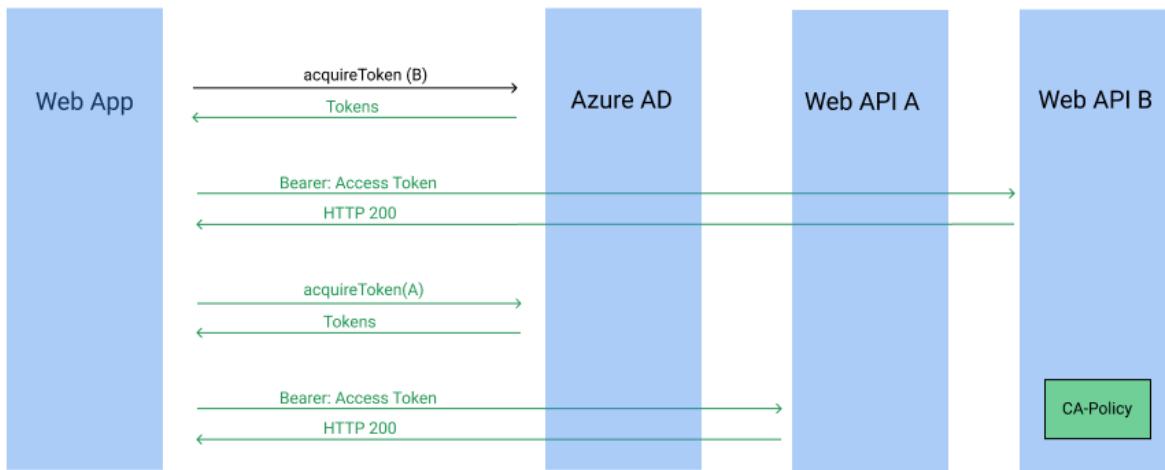
In Web API 1, we catch the error `error=interaction_required`, and send back the `claims` challenge to the desktop app. At that point, the desktop app can make a new `acquireToken()` call and append the `claims` challenge as an extra query string parameter. This new request requires the user to do multi-factor 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.

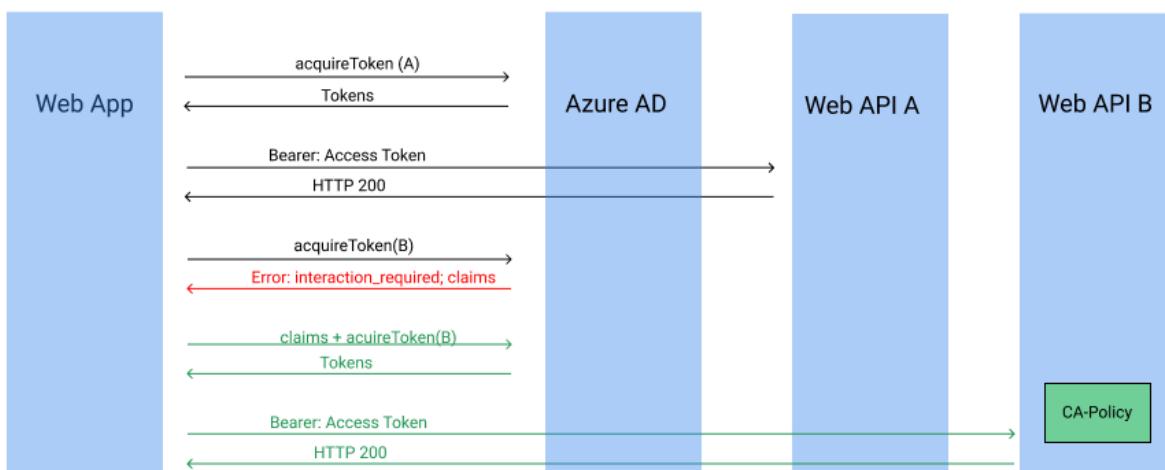


Alternatively, if the app initially requests a token for web service A, the end user does not invoke the Conditional Access policy. This allows the app developer to control the end-user experience and not force the Conditional Access policy to be invoked in all cases. The tricky case is if the app subsequently 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 MSAL 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.

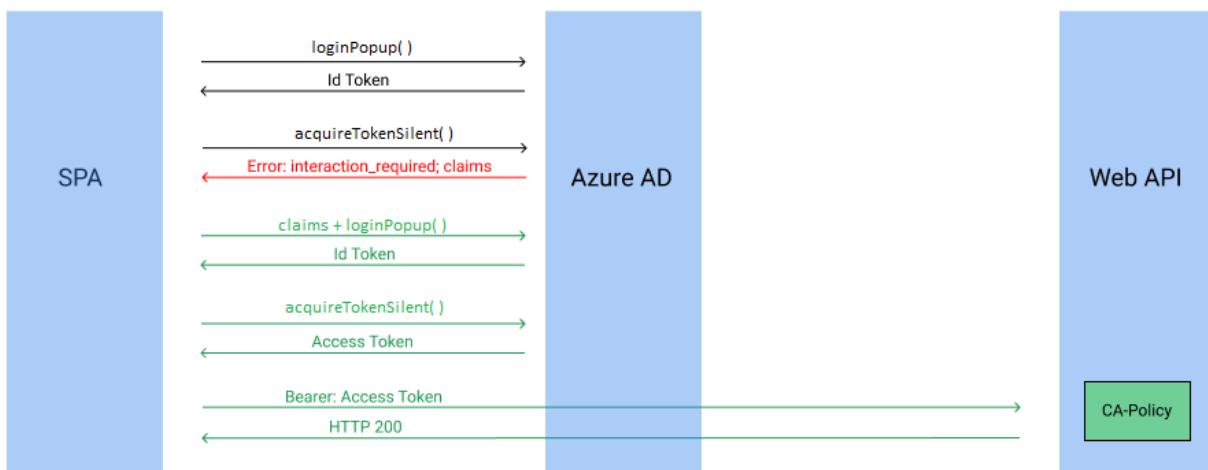
Scenario: Single-page app (SPA) using MSAL.js

In this scenario, we walk through the case when we have a single-page app (SPA) calling a Conditional Access protected web API using MSAL.js. This is a simple architecture but has some nuances that need to be taken into account when developing around Conditional Access.

In MSAL.js, there are a few functions that obtain tokens: `acquireTokenSilent()`, `acquireTokenPopup()`, and `acquireTokenRedirect()`.

- `acquireTokenSilent()` can 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 `acquireTokenSilent()`. If the token 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 `loginPopup()` 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 `acquireTokenSilent()` 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 [JavaScript SPA calling Node.js web API using on-behalf-of flow](#) code sample. This code sample uses the Conditional Access policy and web API you registered earlier with a JavaScript 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.

See also

- To learn more about the capabilities, see [Conditional Access in Azure Active Directory](#).
- For more Azure AD code samples, see [samples](#).
- For more info on the MSAL SDK's and access the reference documentation, see the [Microsoft Authentication Library overview](#).
- To learn more about multi-tenant scenarios, see [How to sign in users using the multi-tenant pattern](#).
- Learn more about [Conditional access and securing access to IoT apps](#).

Developers' guide to Conditional Access authentication context

4/12/2022 • 9 minutes to read • [Edit Online](#)

[Conditional Access](#) is the Zero Trust control plane that allows you to target policies for access to all your apps – old or new, private, or public, on-premises, or multi-cloud. With [Conditional Access authentication context](#), you can apply different policies within those apps.

Conditional Access authentication context (auth context) allows you to apply granular policies to sensitive data and actions instead of just at the app level. You can refine your Zero Trust policies for least privileged access while minimizing user friction and keeping users more productive and your resources more secure. Today, it can be used by applications using [OpenID Connect](#) for authentication developed by your company to protect sensitive resources, like high-value transactions or viewing employee personal data.

Use the Azure AD Conditional access engine's new auth context feature to trigger a demand for step-up authentication from within your application and services. Developers now have the power to demand enhanced stronger authentication, selectively, like MFA from their end users from within their applications. This feature helps developers build smoother user experiences for most parts of their application, while access to more secure operations and data remains behind stronger authentication controls.

Problem statement

The IT administrators and regulators often struggle between balancing prompting their users with additional factors of authentication too frequently and achieving adequate security and policy adherence for applications and services where parts of them contain sensitive data and operations. It can be a choice between a strong policy that impacts users' productivity when they access most data and actions or a policy that is not strong enough for sensitive resources.

So, what if apps were able to mix both, where they can function with a relatively lesser security and less frequent prompts for most users and operations and yet conditionally stepping up the security requirement when the users accessed more sensitive parts?

Common scenarios

For example, while users may sign in to SharePoint using multi-factor authentication, accessing site collection in SharePoint containing sensitive documents can require a compliant device and only be accessible from trusted IP ranges.

Steps

The following are the prerequisites and the steps if you want to use Conditional Access authentication context.

Prerequisites

First, your app should be integrated with the Microsoft Identity Platform using the use [OpenID Connect/ OAuth 2.0](#) protocols for authentication and authorization. We recommend you use [Microsoft identity platform authentication libraries](#) to integrate and secure your application with Azure Active Directory. [Microsoft identity platform documentation](#) is a good place to start learning how to integrate your apps with the Microsoft Identity Platform. Conditional Access Auth Context feature support is built on top of protocol extensions provided by the industry standard [OpenID Connect](#) protocol. Developers use a [Conditional Access Auth Context reference value](#) with the [Claims Request](#) parameter to give apps a way to trigger and satisfy policy.

Second, [Conditional Access](#) requires Azure AD Premium P1 licensing. More information about licensing can be found on the [Azure AD pricing page](#).

Third, today it is only available to applications that sign-in users. Applications that authenticate as themselves are not supported. Use the [Authentication flows and application scenarios guide](#) to learn about the supported authentication app types and flows in the Microsoft Identity Platform.

Integration steps

Once your application is integrated using the supported authentication protocols and registered in an Azure AD tenant that has the Conditional Access feature available for use, you can kick start the process to integrating this feature in your applications that sign-in users.

NOTE

A detailed walkthrough of this feature is also available as a recorded session at [Use Conditional Access Auth Context in your app for step-up authentication](#).

First, declare and make the authentication contexts available in your tenant. For more information, see [Configure authentication contexts](#).

Values C1-C25 are available for use as **Auth Context IDs** in a tenant. Examples of auth context may be:

- **C1** – Require strong authentication
- **C2** – Require compliant devices
- **C3** – Require trusted locations

Create or modify your Conditional Access policies to use the Conditional Access Auth Contexts. Examples policies could be:

- All users signing-into this web application should have successfully completed 2FA for auth context ID C1.
- All users signing into this web application should have successfully completed 2FA and also access the web app from a certain IP address range for auth context ID C3.

NOTE

The Conditional Access auth context values are declared and maintained separately from applications. It is not advisable for applications to take hard dependency on auth context ids. The Conditional Access policies are usually crafted by IT administrators as they have a better understanding of the resources available to apply policies on. For example, for an Azure AD tenant, IT admins would have the knowledge of how many of the tenant's users are equipped to use 2FA for MFA and thus can ensure that Conditional Access policies that require 2FA are scoped to these equipped users. Similarly, if the application is used in multiple tenants, the auth context ids in use could be different and, in some cases, not available at all.

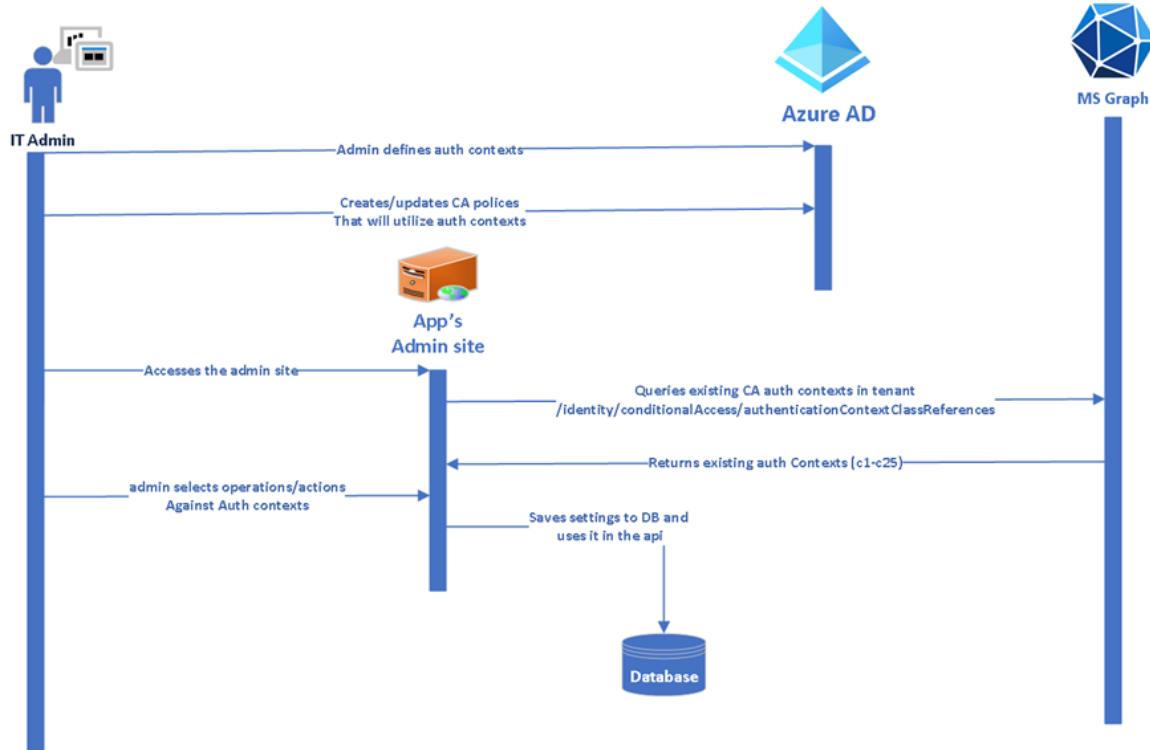
Second: The developers of an application planning to use Conditional Access auth context are advised to first provide the application admins or IT admins a means to map potential sensitive actions to auth context IDs. The steps roughly being:

1. Identity actions in the code that can be made available to map against auth context IDs.
2. Build a screen in the admin portal of the app (or an equivalent functionality) that IT admins can use to map sensitive actions against an available auth context ID.
3. See the code sample, [Use the Conditional Access Auth Context to perform step-up authentication](#) for an example on how it is done.

These steps are the changes that you need to carry in your code base. The steps broadly comprise of

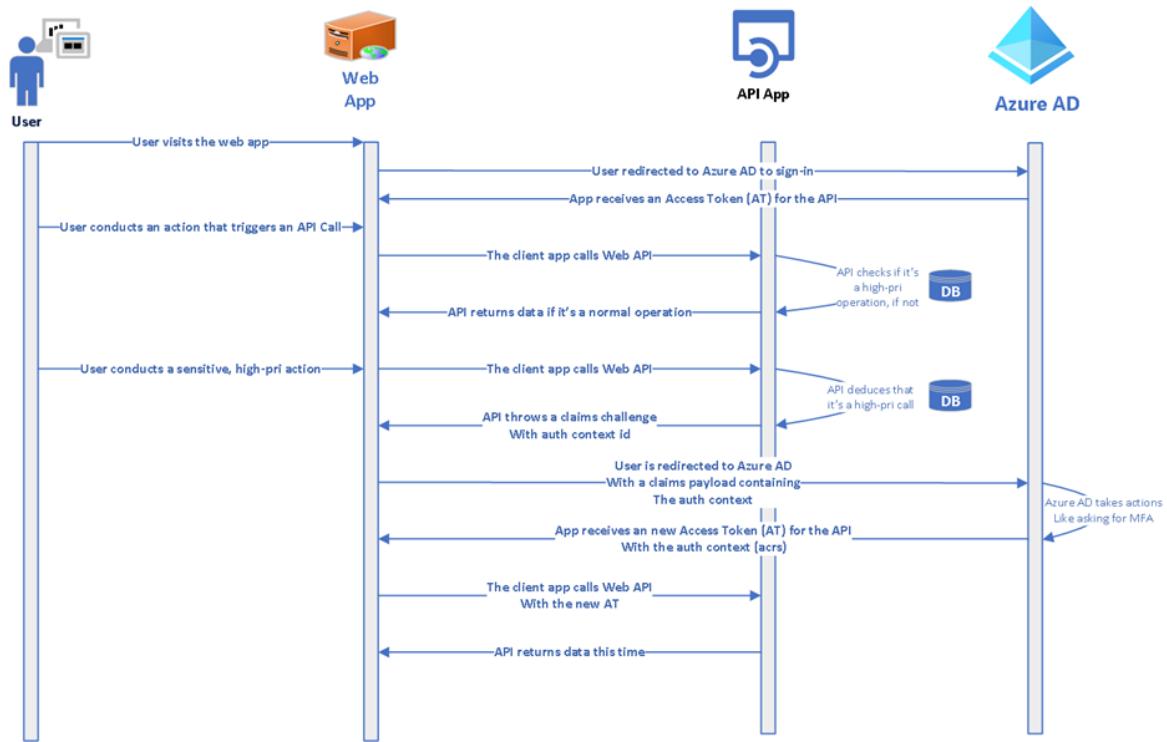
- Query MS Graph to [list all the available Auth Contexts](#).
- Allow IT admins to select sensitive/ high-privileged operations and assign them against the available Auth Contexts using CA policies.
- Save this mapping information in your database/local store.

Set up auth context using an admin site



Third: Your application, and for this example, we'd assume it's a web API, then needs to evaluate calls against the saved mapping and accordingly raise claim challenges for its client apps. To prepare for this action, the following steps are to be taken:

1. In a sensitive and protected by auth context operation, evaluate the values in the **acrs** claim against the Auth Context ID mappings saved earlier and raise a [Claims Challenge](#) as provided in the code snippet below.
2. The following diagram shows the interaction between the user, client app, and the web API.



The code snippet that follows is from the code sample, [Use the Conditional Access auth context to perform step-up authentication](#). The first method, `CheckForRequiredAuthContext()` in the API

- Checks if the application's action being called requires step-up authentication. It does so by checking its database for a saved mapping for this method
- If this action indeed requires an elevated auth context, it checks the `acrs` claim for an existing, matching Auth Context ID.
- If a matching Auth Context ID is not found, it raises a [claims challenge](#).

```

public void CheckForRequiredAuthContext(string method)
{
    string authType = _commonDBContext.AuthContext.FirstOrDefault(x => x.Operation == method
        && x.TenantId == _configuration["AzureAD:TenantId"]?.AuthContextId);

    if (!string.IsNullOrEmpty(authType))
    {
        HttpContext context = this.HttpContext;
        string authenticationContextClassReferencesClaim = "acrs";

        if (context == null || context.User == null || context.User.Claims == null
            || !context.User.Claims.Any())
        {
            throw new ArgumentNullException("No Usercontext is available to pick claims from");
        }

        Claim acrsClaim =
context.User.FindAll(authenticationContextClassReferencesClaim).FirstOrDefault(x
            => x.Value == authType);

        if (acrsClaim == null || acrsClaim.Value != authType)
        {
            if (IsClientCapableofClaimsChallenge(context))
            {
                string clientId = _configuration.GetSection("AzureAd").GetSection("ClientId").Value;
                var base64str = Convert.ToBase64String(Encoding.UTF8.GetBytes("{\"access_token\":"
                    + "\"acrs\":[\"essential\":true,\"value\":\"" + authType + "\"]}}"));

                context.Response.Headers.Append("WWW-Authenticate", $"Bearer realm=\"\",
authorization_uri=\"https://login.microsoftonline.com/common/oauth2/authorize\", client_id=\""
                    + clientId + "\", error=\"insufficient_claims\", claims=\"\" + base64str + "\","
                    + "cc_type=\"authcontext\"");
                context.Response.StatusCode = (int) HttpStatusCode.Unauthorized;
                string message = string.Format(CultureInfo.InvariantCulture, "The presented access
tokens had insufficient claims. Please request for claims requested in the WWW-Authentication header
and try again.");
                context.Response.WriteAsync(message);
                context.Response.CompleteAsync();
                throw new UnauthorizedAccessException(message);
            }
            else
            {
                throw new UnauthorizedAccessException("The caller does not meet the authentication
bar to carry our this operation. The service cannot allow this operation");
            }
        }
    }
}

```

NOTE

The format of the claims challenge is described in the article, [Claims Challenge in the Microsoft Identity Platform](#).

3. In the client application, Intercept the claims challenge and redirect the user back to Azure AD for further policy evaluation. The code snippet that follows is from the code sample, [Use the Conditional Access auth context to perform step-up authentication](#).

```

internal static string ExtractHeaderValues(WebApiMsalUiRequiredException response)
{
    if (response.StatusCode == System.Net.HttpStatusCode.Unauthorized &&
    response.Headers.WwwAuthenticate.Any())
    {
        AuthenticationHeaderValue bearer = response.Headers.WwwAuthenticate.First(v => v.Scheme == "Bearer");
        IEnumerable<string> parameters = bearer.Parameter.Split(',').Select(v => v.Trim()).ToList();
        var errorValue = GetParameterValue(parameters, "error");

        try
        {
            // read the header and checks if it contains error with insufficient_claims value.
            if (null != errorValue && "insufficient_claims" == errorValue)
            {
                var claimChallengeParameter = GetParameterValue(parameters, "claims");
                if (null != claimChallengeParameter)
                {
                    var claimChallenge = ConvertBase64String(claimChallengeParameter);

                    return claimChallenge;
                }
            }
        }
        catch (Exception ex)
        {
            throw ex;
        }
    }
    return null;
}

```

Handle exception in the call to Web API, if a claims challenge is presented, the redirect the user back to Azure AD for further processing.

```

try
{
    // Call the API
    await _todoListService.AddAsync(todo);
}
catch (WebApiMsalUiRequiredException hex)
{
    // Challenges the user if exception is thrown from Web API.
    try
    {
        var claimChallenge = ExtractHeaderValues(hex);
        _consentHandler.ChallengeUser(new string[] { "user.read" }, claimChallenge);

        return new EmptyResult();
    }
    catch (Exception ex)
    {
        _consentHandler.HandleException(ex);
    }

    Console.WriteLine(hex.Message);
}
return RedirectToAction("Index");

```

4. (Optional) Declare client capability. Client capabilities help resources providers (RP) like our Web API above to detect if the calling client application understands the claims challenge and can then customize its response accordingly. This capability could be useful where not all of the APIs clients are capable of handling claim challenges and some older ones still expect a different response. For more information,

see the section [Client capabilities](#).

Caveats and recommendations

Do not hard-code Auth Context values in your app. Apps should read and apply auth context [using MS Graph calls](#). This practice is critical for [multi-tenant applications](#). The Auth Context values will vary between Azure AD tenants will not available in Azure AD free edition. For more information on how an app should query, set, and use auth context in their code, see the code sample, [Use the Conditional Access auth context to perform step-up authentication](#) as how an app should query, set and use auth context in their code.

Do not use auth context where the app itself is going to be a target of Conditional Access policies. The feature works best when parts of the application require the user to meet a higher bar of authentication.

Next steps

- [Granular Conditional Access for sensitive data and actions \(Blog\)](#)
- [Zero trust with the Microsoft Identity platform](#)
- [Building Zero Trust ready apps with the Microsoft identity platform](#)
- [Use the Conditional Access auth context to perform step-up authentication for high-privilege operations in a Web app](#)
- [Use the Conditional Access auth context to perform step-up authentication for high-privilege operations in a Web API](#)
- [Conditional Access authentication context](#)
- [authenticationContextClassReference resource type - MS Graph](#)
- [Claims challenge, claims request, and client capabilities in the Microsoft Identity Platform](#)
- [Using authentication context with Microsoft Information Protection and SharePoint](#)
- [Authentication flows and application scenarios](#)
- [How to use Continuous Access Evaluation enabled APIs in your applications](#)

Application model

4/12/2022 • 4 minutes to read • [Edit Online](#)

Applications can sign in users themselves or delegate sign-in to an identity provider. This article discusses the steps that are required to register an application with the Microsoft identity platform.

Register an application

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 Active Directory (Azure AD), you're providing an identity configuration for your application that allows it to integrate with the Microsoft identity platform. Registering the app also allows you to:

- Customize the branding of your application in the sign-in dialog box. This branding is important because signing in is the first experience a user will have with your app.
- Decide if you want to allow users to sign in only if they belong to your organization. This architecture is known as a single-tenant application. Or, you can allow users to sign in by using any work or school account, which is known as 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 the Microsoft identity platform that proves the app's identity. Using a secret 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. A trusted back-end server is required to store the credentials.

After the app is registered, it's given a unique identifier that it shares with the Microsoft identity platform 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 by using a model that fulfills two main functions:

- Identify the app by the authentication protocols it supports.
- 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 facilitates 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.

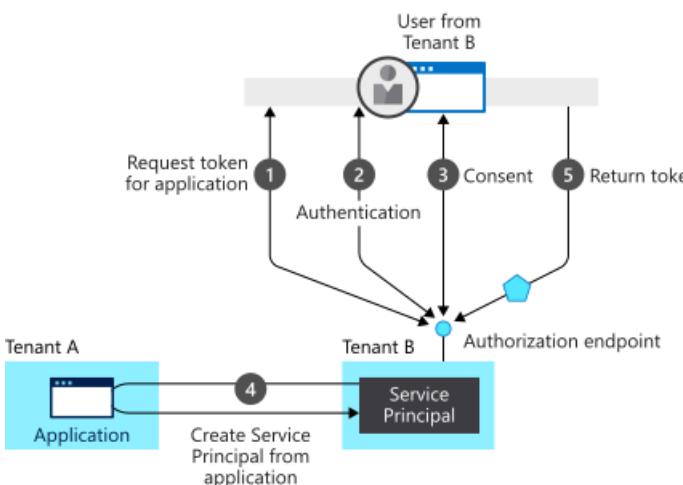
- Administrators ultimately decide what apps are allowed to do and which users can use specific apps, and how the directory resources are accessed.

Multi-tenant apps

In the Microsoft identity platform, an [application object](#) describes an application. 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:

- A user from tenant B attempts to sign in with the app. The authorization endpoint requests a token for the application.
- The user credentials are acquired and verified for authentication.
- The user is prompted to provide consent for the app to gain access to tenant B.
- The Microsoft identity platform uses the application object in tenant A as a blueprint for creating a service principal in tenant B.
- The user receives the requested token.

You can repeat this process for more 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 the Microsoft identity platform](#).

Next steps

For more information about authentication and authorization in the Microsoft identity platform, see the following articles:

- To learn about the basic concepts of authentication and authorization, see [Authentication vs. authorization](#).
- To learn how access tokens, refresh tokens, and ID tokens are used in authentication and authorization, see [Security tokens](#).
- To learn about the sign-in flow of web, desktop, and mobile apps, see [App sign-in flow](#).

For more information about the application model, see the following articles:

- For more information on application objects and service principals in the Microsoft identity platform, see [How and why applications are added to Azure AD](#).
- For more information on single-tenant apps and multi-tenant apps, see [Tenancy in Azure Active Directory](#).
- For more information on how Azure AD also provides Azure Active Directory B2C so that organizations can sign in users, typically customers, by using social identities like a Google account, see [Azure Active Directory B2C documentation](#).

What are workload identities?

4/12/2022 • 2 minutes to read • [Edit Online](#)

A workload identity is an identity used by a software workload (such as an application, service, script, or container) to authenticate and access other services and resources. The terminology is inconsistent across the industry, but generally a workload identity is something you need for your software entity to authenticate with some system. For example, a workload identity could be a user account that your client authenticates as to access a MongoDB database. A workload identity could also be an AWS service role attached to an EC2 instance with read-only access to an Amazon S3 bucket.

In Azure Active Directory (Azure AD), workload identities are applications, service principals, and managed identities.

An [application](#) is an abstract entity, or template, defined by its application object. The application object is the *global*/representation of your application for use across all tenants. The application object describes how tokens are issued, the resources the application needs to access, and the actions that the application can take.

A [service principal](#) is the *local*/representation, or application instance, of a global application object in a specific tenant. An application object is used as a template to create a service principal object in every tenant where the application is used. The service principal object defines what the app can actually do in a specific tenant, who can access the app, and what resources the app can access.

A [managed identity](#) is a special type of service principal that eliminates the need for developers to manage credentials.

Here are some ways that workload identities in Azure AD are used:

- An app that enables a web app to access Microsoft Graph based on admin or user consent. This access could be either on behalf of the user or on behalf of the application.
- A managed identity used by a developer to provision their service with access to an Azure resource such as Azure Key Vault or Azure Storage.
- A service principal used by a developer to enable a CI/CD pipeline to deploy a web app from GitHub to Azure App Service.

Workload identities, other machine identities, and human identities

At a high level, there are two types of identities: human and machine/non-human identities. Workload identities and device identities together make up a group called machine (or non-human) identities. Workload identities represent software workloads while device identities represent devices such as desktop computers, mobile, IoT sensors, and IoT managed devices. Machine identities are distinct from human identities, which represent people such as employees (internal workers and front line workers) and external users (customers, consultants, vendors, and partners).



Supported scenarios

Here are some ways you can use workload identities:

- Review service principals and applications that are assigned to privileged directory roles in Azure AD using [access reviews for service principals](#).
- Access Azure AD protected resources without needing to manage secrets (for supported scenarios) using [workload identity federation](#).
- Apply Conditional Access policies to service principals owned by your organization using [Conditional Access for workload identities](#).

Next steps

Learn how to [secure access of workload identities](#) with adaptive policies.

Application and service principal objects in Azure Active Directory

4/12/2022 • 6 minutes to read • [Edit Online](#)

This article describes application registration, application objects, and service principals in Azure Active Directory (Azure AD): what they are, how they're used, and how they are related to each other. A multi-tenant example scenario is also presented to illustrate the relationship between an application's application object and corresponding service principal objects.

Application registration

To delegate Identity and Access Management functions to Azure AD, an application must be registered with an Azure AD tenant. When you register your application with Azure AD, you're creating an identity configuration for your application that allows it to integrate with Azure AD. When you register an app in the Azure portal, you choose whether it's a single tenant (only accessible in your tenant) or multi-tenant (accessible in other tenants) and can optionally set a redirect URI (where the access token is sent to). For step-by-step instructions on registering an app, see the [app registration quickstart](#).

When you've completed the app registration, you have a globally unique instance of the app (the application object) which lives within your home tenant or directory. You also have a globally unique ID for your app (the app or client ID). In the portal, you can then add secrets or certificates and scopes to make your app work, customize the branding of your app in the sign-in dialog, and more.

If you register an application in the portal, an application object as well as a service principal object are automatically created in your home tenant. If you register/create an application using the Microsoft Graph APIs, creating the service principal object is a separate step.

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). An application object is used as a template or blueprint to create one or more service principal objects. A service principal is created in every tenant where the application is used. Similar to a class in object-oriented programming, the application object has some static properties that are applied to all the created service principals (or application instances).

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

You can use the **App registrations** blade in the [Azure portal](#) to list and manage the application objects in your home tenant.

Display name	Application (client) ID	Created on	Certificates & secrets
AP	f3acec02-b647-446e-9...	12/12/2019	-
NA	15005dcb-a021-41a1-8...	12/12/2019	-
AU	3d070de4-37d7-4d14...	1/30/2020	✓ Current
EX	fc94652e-0259-47a1-a...	6/25/2020	-

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 requirement 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.

There are three types of service principal:

- **Application** - The type of service principal is the local representation, or application instance, of a global application object in a single tenant or directory. In this case, a service principal is a concrete instance created from the application object and inherits certain properties from that application object. A service principal is created in each tenant where the application is used and references the globally unique app object. The service principal object defines what the app can actually do in the specific tenant, who can access the app, and what resources the app can access.

When an application is given permission to access resources in a tenant (upon registration or consent), a service principal object is created. When you register an application using the Azure portal, a service principal is created automatically. You can also create service principal objects in a tenant using Azure PowerShell, Azure CLI, Microsoft Graph, and other tools.

- **Managed identity** - This type of service principal is used to represent a [managed identity](#). Managed identities eliminate the need for developers to manage credentials. Managed identities provide an identity for applications to use when connecting to resources that support Azure AD authentication. When a managed identity is enabled, a service principal representing that managed identity is created in your tenant. Service principals representing managed identities can be granted access and permissions, but cannot be updated or modified directly.
- **Legacy** - This type of service principal represents a legacy app, which is an app created before app registrations were introduced or an app created through legacy experiences. A legacy service principal can have credentials, service principal names, reply URLs, and other properties that an authorized user can edit, but does not have an associated app registration. The service principal can only be used in the tenant where it was created.

The Microsoft Graph [ServicePrincipal entity](#) defines the schema for a service principal object's properties.

You can use the **Enterprise applications** blade in the Azure portal to list and manage the service principals in a tenant. You can see the service principal's permissions, user consented permissions, which users have done that consent, sign in information, and more.

The screenshot shows the 'Enterprise applications | All applications' blade in the Azure portal. The left sidebar has sections for Overview, Manage (with 'All applications' selected), Application proxy, User settings, Security, Conditional Access, Consent and permissions, Activity, Sign-ins, Usage & insights (Preview), Audit logs, and Provisioning logs (Preview). The main area has filters for Application type (Enterprise Applications), Applications status (Any), Application visibility (Any), and buttons for 'Apply' and 'Reset'. A message bar says 'Try out the new Enterprise Apps search preview! Click to enable the preview.' Below is a table with columns: Name, Homepage URL, Object ID, and Application ID. The table lists several applications:

Name	Homepage URL	Object ID	Application ID
AppModelv2-NativeClient-1		e7fe23c0-3332-448a-a78...	f3acec02-b647-446e-9bc...
AuthTest		3c7ff4bf-349e-4200-b37...	3d070de4-37d7-4d14-9f...
example-app		265104ee-632c-44df-913...	fc94652e-0259-47a1-a57...
FourthCoffeeApp		e1f3bb11-cd76-407d-89...	b65c7678-0add-4b3c-a7...
Graph explorer	https://developer.microsoft.com/e...	a70ddeab-79a5-4936-a5f...	de8bc8b5-d9f9-48b1-a8...
NativeClient-DotNet-TodoL		be97cda5-52b4-4521-84...	15005dcba021-41a1-86...
Tutorial Sample App		44f38317-644b-4bd9-a6...	6731de76-14a6-49ae-97...

Relationship between application objects and service principals

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

An application object has:

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

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 application also has a service principal created in each tenant where a user from that tenant has consented to its use.

Consequences of modifying and deleting applications

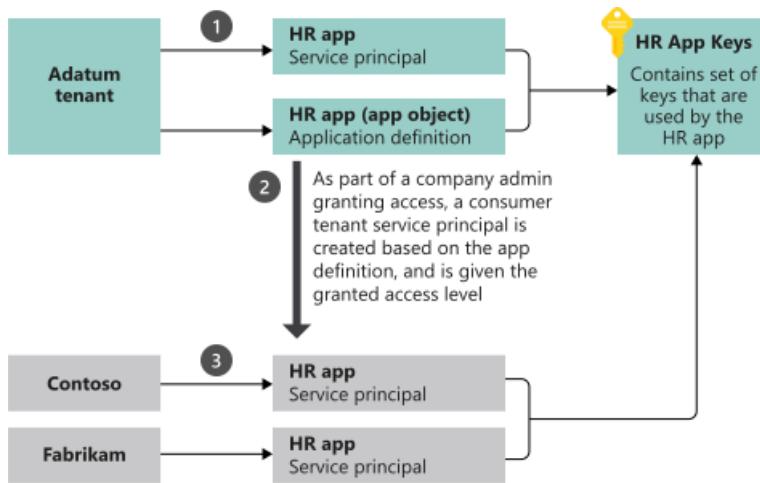
Any changes that you make to your application object are also reflected in its service principal object in the application's home tenant only (the tenant where it was registered). This means that deleting an application object will also delete its home tenant service principal object. However, restoring that application object will not restore its corresponding service principal. 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.

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

Learn how to create a service principal:

- [Using the Azure portal](#)
- [Using Azure PowerShell](#)
- [Using Azure CLI](#)
- [Using Microsoft Graph](#) and then use [Microsoft Graph Explorer](#) to query both the application and service principal objects.

How and why applications are added to Azure AD

4/12/2022 • 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 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

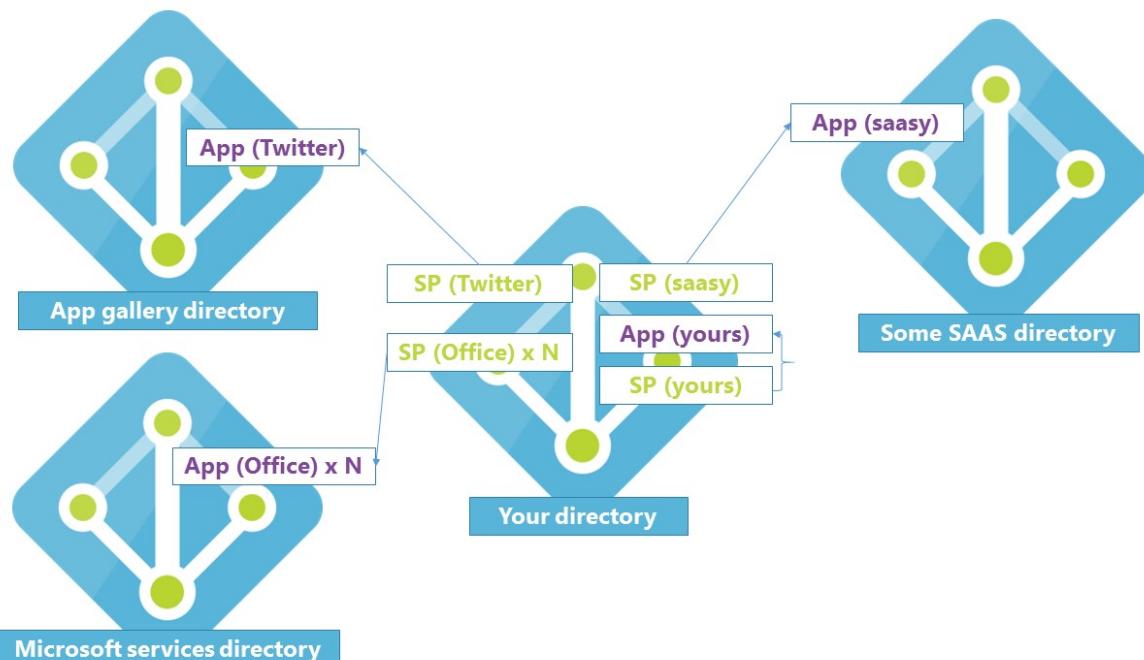
- Records of local user and admin permissions granted to the application
 - For example: permission for the application to access a particular user's email
- Records of local policies including Conditional Access policy
- Records of alternate local settings for an application
 - Claims transformation 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 [Microsoft 365](#)
 - When you subscribe to Microsoft 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 Microsoft 365.
 - Some Microsoft 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 Microsoft 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 Microsoft 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 Microsoft Graph 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 Microsoft 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
- Directory schema extension attributes - [Extend the schema of service principal and user objects](#) to store additional data in Azure AD

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 change the user consent settings in your organization, see [Configure how users consent to applications](#).
- 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 allowing users to register applications and only allows user consent for a very limited set of permissions.

Tenancy in Azure Active Directory

4/12/2022 • 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 Microsoft 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 Microsoft 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.
- Provide appropriate names and descriptions for any permissions you expose as part of your app. This helps users and admins know what they're 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

For more information about tenancy in Azure AD, see:

- [How to convert an app to be multi-tenant](#)
- [Enable multi-tenant log-ins](#)

Build Zero Trust-ready apps using Microsoft identity platform features and tools

4/12/2022 • 4 minutes to read • [Edit Online](#)

You can no longer assume a secure network perimeter around the applications you build. Nearly every app you build will, by design, be accessed from outside the network perimeter. You also can't guarantee every app you build is secure or will remain so after it's deployed.

Knowing this as the app developer, it's your responsibility to not only maximize your app's security, but also minimize the damage your app can cause if it's compromised.

Additionally, you are responsible for supporting the evolving needs of your customers and users, who will expect that your application meets their Zero Trust security requirements.

By learning the principles of the [Zero Trust model](#) and adopting their practices, you can:

- Build more secure apps
- Minimize the damage your apps could cause if there is a breach

Zero Trust principles

The Zero Trust model prescribes a culture of explicit verification rather than implicit trust. The model is anchored on three key [guiding principles](#):

- Verify explicitly
- Use least privileged access
- Assume breach

Best practices for building Zero Trust-ready apps with the Microsoft identity platform

Follow these best practices to build Zero Trust-ready apps with the [Microsoft identity platform](#) and its tools.

Verify explicitly

The Microsoft identity platform offers authentication mechanisms for verifying the identity of the person or service accessing a resource. Apply the best practices described below to ensure that you *verify explicitly* any entities that need to access data or resources.

BEST PRACTICE	BENEFITS TO APP SECURITY
Use the Microsoft Authentication Libraries (MSAL).	MSAL is a set of Microsoft's authentication libraries for developers. With MSAL, you can authenticate users and applications, and acquire tokens to access corporate resources using just a few lines of code. MSAL uses modern protocols (OpenID Connect and OAuth 2.0) that remove the need for apps to ever handle a user's credentials directly. This vastly improves the security for both users and applications as the identity provider becomes the security perimeter. Also, these protocols continuously evolve to address new paradigms, opportunities, and challenges in identity security.

BEST PRACTICE	BENEFITS TO APP SECURITY
Adopt enhanced security extensions like Continuous Access Evaluation (CAE) and Conditional Access authentication context when appropriate.	In Azure AD, some of the most used extensions include Conditional Access (CA), Conditional Access authentication context and CAE. Applications that use enhanced security features like CAE and Conditional Access authentication context must be coded to handle claims challenges. Open protocols enable you to use the claims challenges and claims requests to invoke extra client capabilities. This might be to indicate to apps that they need to re-interact with Azure AD, like if there was an anomaly or if the user no longer satisfies the conditions under which they authenticated earlier. As a developer you can code for these extensions without disturbing their primary code flows for authentication.
Use the correct authentication flow by app type for authentication. For web applications, you should always aim to use confidential client flows . For mobile applications, you should use brokers or the system browser for authentication.	The flows for web applications that can hold a secret (confidential clients) are considered more secure than public clients (for example: Desktop and Console apps). When you use the system web browser to authenticate your mobile apps, you get a secure single sign-on (SSO) experience that supports app protection policies.

Use least privileged access

Using the Microsoft identity platform, you can grant permissions (scopes) and verify that a caller has been granted proper permission before allowing access. You can enforce least privileged access in your apps by enabling fine-grained permissions that allow you to grant the smallest amount of access necessary. Follow the practices described below to ensure you adhere to the [principle of least privilege](#).

DO	DON'T
Evaluate the permissions you request to ensure that you seek the absolute least privileged set to get the job done.	Create "catch-all" permissions with access to the entire API surface.
When designing APIs, provide granular permissions to allow least-privileged access. Start with dividing the functionality and data access into sections that can be controlled via scopes and App roles .	Add your APIs to existing permissions in a way that changes the semantics of the permission.
Offer read-only permissions. " <i>Write</i> access", includes privileges for create, update, and delete operations. A client should never require write access to only read data	-----
Offer both delegated and application permissions. Skipping application permissions can create hard requirement for your clients to achieve common scenarios like automation, microservices and more.	-----
Consider "standard" and "full" access permissions if working with sensitive data. Restrict the sensitive properties so that they cannot be accessed using a "standard" access permission, for example <code>Resource.Read</code> . And then implement a "full" access permission, for example <code>Resource.ReadFull</code> , that returns all available properties including sensitive information.	-----

Assume breach

The Microsoft identity platform app registration portal is the primary entry point for applications intending to use the platform for their authentication and associated needs. When registering and configuring your apps,

follow the practices described below to minimize the damage your apps could cause if there is a security breach. For more guidance, check [Azure AD application registration security best practices](#).

Home > Microsoft

Microsoft | App registrations ... X

Azure Active Directory

Manage

- Users
- Groups
- External Identities
- Roles and administrators
- Administrative units
- Enterprise applications
- Devices
- App registrations**
- Identity Governance
- Application proxy
- Custom security attributes (Preview)

+ New registration Endpoints Troubleshooting Refresh Download Preview features | Got feedback?

Starting June 30th, 2020 we will no longer add any new features to Azure Active Directory Authentication Library (ADAL) and Azure AD Graph. We will continue to provide technical support and security updates but we will no longer provide feature updates. Applications will need to be upgraded to Microsoft Authentication Library (MSAL) and Microsoft Graph. [Learn more](#)

⚠ 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](#)

All applications Owned applications Deleted applications

Start typing a display name to filter these results Application (client) ID starts with Add filters

3 applications found

Display name ↑	Application (client) ID	Created on ↑↓	Certificates & secrets
AS another sample	08402e8d-5f7d-415e-8956-f4e7f889...	5/26/2021	-

DO	DON'T
Properly define your redirect URLs	Use the same app registration for multiple apps
Check redirect URIs used in your app registration for ownership and to avoid domain takeovers	Create your application as a multi-tenant unless you really intended to
Ensure app and service principal owners are always defined and maintained for your apps registered in your tenant	-----

Next steps

- Zero Trust [Guidance Center](#)
- Microsoft identity platform [best practices and recommendations](#).

Enhance security with the principle of least privilege

4/12/2022 • 4 minutes to read • [Edit Online](#)

The information security principle of least privilege asserts that users and applications should be granted access only to the data and operations they require to perform their jobs.

Follow the guidance here to help reduce your application's attack surface and the impact of a security breach (the *blast radius*) should one occur in your Microsoft identity platform-integrated application.

Recommendations at a glance

- Prevent **overprivileged** applications by revoking *unused* and *reducible* permissions.
- Use the identity platform's **consent** framework to require that a human consents to the app's request to access protected data.
- **Build** applications with least privilege in mind during all stages of development.
- **Audit** your deployed applications periodically to identify overprivileged apps.

What's an *overprivileged* application?

Any application that's been granted an **unused** or **reducible** permission is considered "overprivileged." Unused and reducible permissions have the potential to provide unauthorized or unintended access to data or operations not required by the app or its users to perform their jobs.

Unused permissions

An unused permission is a permission that's been granted to an application but whose API or operation exposed by that permission isn't called by the app when used as intended.

- **Example:** An application displays a list of files stored in the signed-in user's OneDrive by calling the Microsoft Graph API and leveraging the [Files.Read](#) permission. However, the app has also been granted the [Calendars.Read](#) permission, yet it provides no calendar features and doesn't call the Calendars API.
- **Security risk:** Unused permissions pose a *horizontal privilege escalation* security risk. An entity that exploits a security vulnerability in your application could use an unused permission to gain access to an API or operation not normally supported or allowed by the application when it's used as intended.
- **Mitigation:** Remove any permission that isn't used in API calls made by your application.

Reducible permissions

A reducible permission is a permission that has a lower-privileged counterpart that would still provide the application and its users the access they need to perform their required tasks.

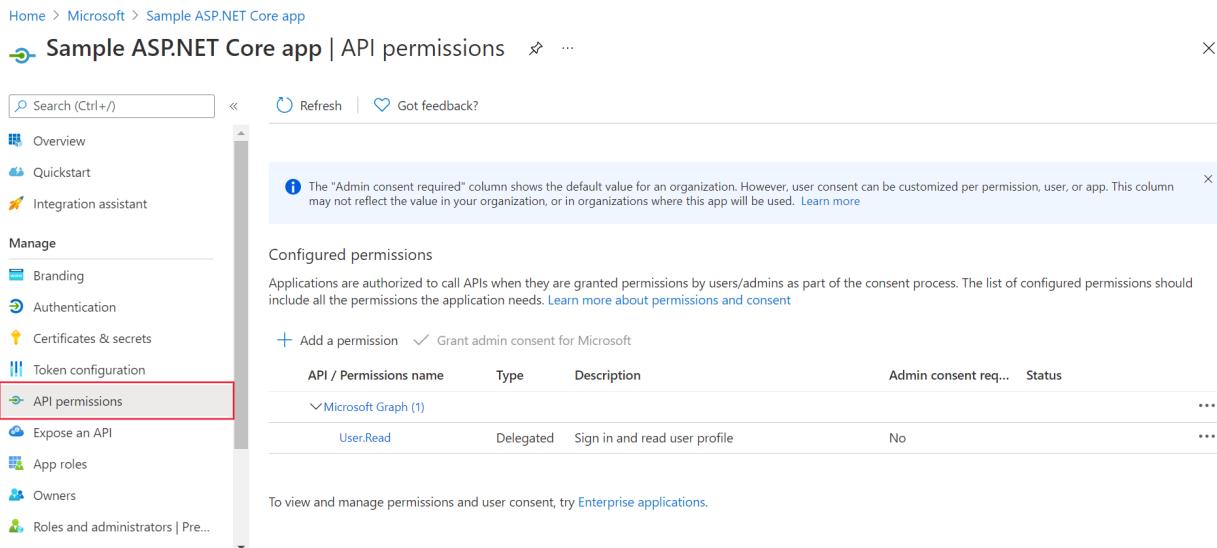
- **Example:** An application displays the signed-in user's profile information by calling the Microsoft Graph API, but doesn't support profile editing. However, the app has been granted the [User.ReadWrite.All](#) permission. The [User.ReadWrite.All](#) permission is considered reducible here because the less permissive [User.Read.All](#) permission grants sufficient read-only access to user profile data.
- **Security risk:** Reducible permissions pose a *vertical privilege escalation* security risk. An entity that exploits a security vulnerability in your application could use the reducible permission for unauthorized access to data or to perform operations not normally allowed by that entity's role.
- **Mitigation:** Replace each reducible permission in your application with its least-permissive counterpart still enabling the application's intended functionality.

Avoid security risks posed by unused and reducible permissions by granting *just enough* permission: the permission with the least-permissive access required by an application or user to perform their required tasks.

Use consent to control access to data

Most applications you build will require access to protected data, and the owner of that data needs to [consent](#) that access. Consent can be granted in several ways, including by a tenant administrator who can consent for *all* users in an Azure AD tenant, or by the application users themselves who can grant access.

Whenever an application that runs in your user's device requests access to protected data, the app should ask for the user's consent before granting access to the protected data. The end user is required to grant (or deny) consent for the requested permission before the application can progress.



The screenshot shows the 'API permissions' section of the Microsoft Azure portal for a 'Sample ASP.NET Core app'. The left sidebar has a red box around the 'API permissions' item under the 'Manage' section. The main content area shows a table of configured permissions:

API / Permissions name	Type	Description	Admin consent req...	Status
Microsoft Graph (1)				
User.Read	Delegated	Sign in and read user profile	No	...

Below the table, a note says: 'To view and manage permissions and user consent, try [Enterprise applications](#)'.

Least privilege during app development

As a developer building an application, consider the security of your app and its users' data to be *your responsibility*.

Adhere to these guidelines during application development to help avoid building an overprivileged app:

- Fully understand the permissions required for the API calls that your application needs to make.
- Understand the least privileged permission for each API call that your app needs to make using [Graph Explorer](#).
- Find the corresponding [permissions](#) from least to most privileged.
- Remove any duplicate sets of permissions in cases where your app makes API calls that have overlapping permissions.
- Apply only the least privileged set of permissions to your application by choosing the least privileged permission in the permission list.

Least privilege for deployed apps

Organizations often hesitate to modify running applications to avoid impacting their normal business operations. However, your organization should consider mitigating the risk of a security incident made possible or more severe by your app's overprivileged permissions to be worthy of a scheduled application update.

Make these standard practices in your organization to help ensure your deployed apps aren't overprivileged and don't become overprivileged over time:

- Evaluate the API calls being made from your applications.
- Use [Graph Explorer](#) and the [Microsoft Graph](#) documentation for the required and least privileged

permissions.

- Audit privileges that are granted to users or applications.
- Update your applications with the least privileged permission set.
- Conduct permissions reviews regularly to make sure all authorized permissions are still relevant.

Next steps

Protected resource access and consent

For more information about configuring access to protected resources and the user experience of providing consent to access those protected resources, see the following articles:

- [Permissions and consent in the Microsoft identity platform](#)
- [Understanding Azure AD application consent experiences](#)

Zero Trust - Consider employing the least-privilege measures described here as part of your organization's proactive [Zero Trust security strategy](#).

Azure AD application registration security best practices

4/12/2022 • 5 minutes to read • [Edit Online](#)

An Azure Active Directory (Azure AD) application registration is a critical part of your business application. Any misconfiguration or lapse in hygiene of your application can result in downtime or compromise.

It's important to understand that your application registration has a wider impact than the business application because of its surface area. Depending on the permissions added to your application, a compromised app can have an organization-wide effect. Since an application registration is essential to getting your users logged in, any downtime to it can affect your business or some critical service that your business depends upon. So, it's important to allocate time and resources to ensure your application registration stays in a healthy state always. We recommend that you conduct a periodical security and health assessment of your applications much like a Security Threat Model assessment for your code. For a broader perspective on security for organizations, check the [security development lifecycle](#) (SDL).

This article describes security best practices for the following application registration properties.

- Redirect URI
- Implicit grant flow for access token
- Credentials
- AppId URI
- Application ownership
- Checklist

Redirect URI configuration

It's important to keep Redirect URIs of your application up to date. A lapse in the ownership of one of the redirect URIs can lead to an application compromise. Ensure that all DNS records are updated and monitored periodically for changes. Along with maintaining ownership of all URIs, don't use wildcard reply URLs or insecure URI schemes such as http, or URN.

Home > saurabhmsft > MyApp

MyApp | Authentication

Search (Ctrl+ /) Save Discard Got feedback?

- Overview
- Quickstart
- Integration assistant

Manage

- Branding
- Authentication**
- Certificates & secrets
- Token configuration
- API permissions
- Expose an API
- App roles | Preview
- Owners
- Roles and administrators | Preview
- Manifest

Support + Troubleshooting

- Troubleshooting
- New support request

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.

Web

Redirect URIs

The URIs we will accept as destinations when returning authentication responses (tokens) after successfully authenticating or signing out users. Also referred to as reply URLs. [Learn more about Redirect URIs and their restrictions](#)

- https://*.myWildcardDomain.com
- https://myExpiredDomain.net
- https://myOldProductName.net

Add URI

Front-channel logout URL

This is where we send a request to have the application clear the user's session data. This is required for single sign-out to work correctly.

e.g. https://example.com/logout

Implicit grant and hybrid flows

Redirect URI summary

DO	DON'T
Maintain ownership of all URIs	Use wildcards
Keep DNS up to date	Use URN scheme
Keep the list small	-----
Trim any unnecessary URIs	-----
Update URLs from Http toHttps scheme	-----

Implicit flow token configuration

Scenarios that required **implicit flow** can now use **Auth code flow** to reduce the risk of compromise associated with implicit grant flow misuse. If you configured your application registration to get Access tokens using implicit flow, but don't actively use it, we recommend you turn off the setting to protect from misuse.

Home > saurabhmsft > MyApp

MyApp | Authentication

Search (Ctrl+.) Save Discard Got feedback? e.g. https://example.com/logout

Implicit grant and hybrid flows

Request a token directly from the authorization endpoint. If the application has a single-page architecture (SPA) and doesn't use the authorization code flow, or if it invokes a web API via JavaScript, select both access tokens and ID tokens. For ASP.NET Core web apps and other web apps that use hybrid authentication, select only ID tokens. [Learn more](#).

Select the tokens you would like to be issued by the authorization endpoint:

Access tokens (used for implicit flows)

ID tokens (used for implicit and hybrid flows)

Supported account types

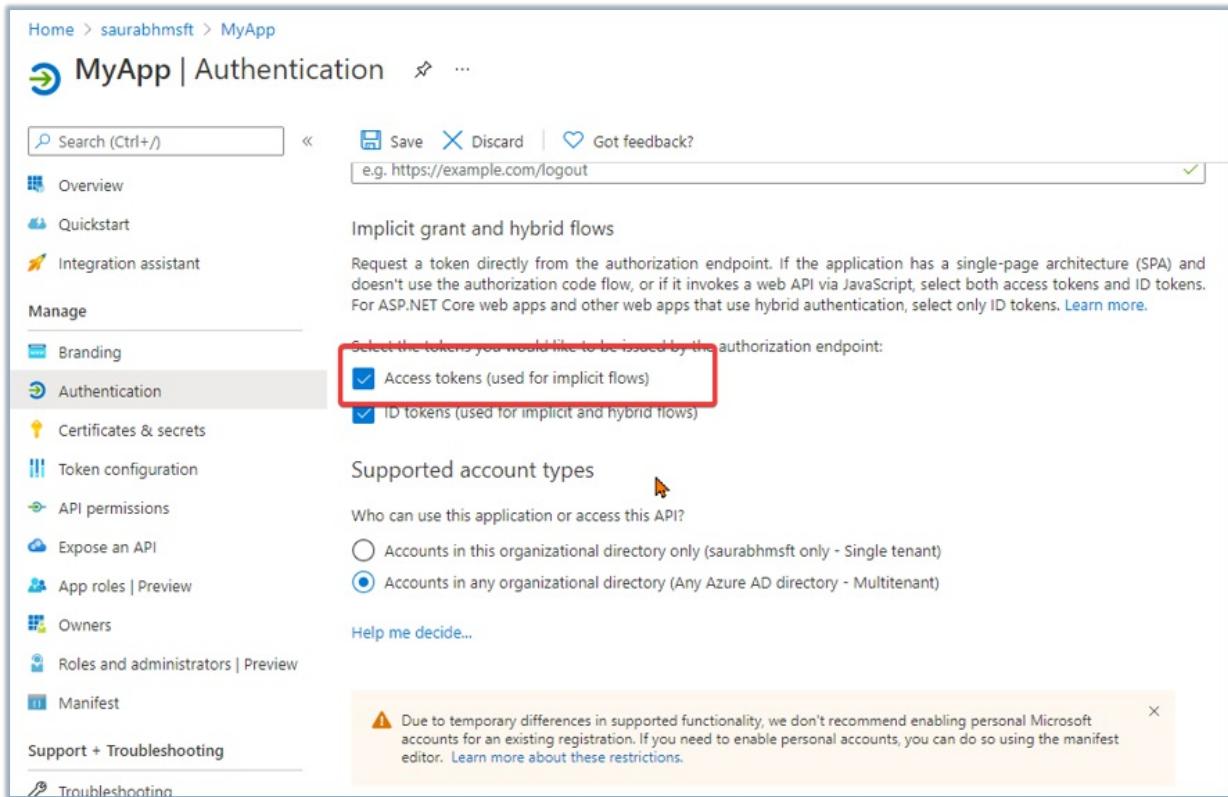
Who can use this application or access this API?

Accounts in this organizational directory only (saurabhmsft only - Single tenant)

Accounts in any organizational directory (Any Azure AD directory - Multitenant)

Help me decide...

⚠ 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. [Learn more about these restrictions.](#)



Implicit grant flow summary

DO	DON'T
Understand if implicit flow is required	Use implicit flow unless explicitly required
Separate app registration for (valid) implicit flow scenarios	-----
Turn off unused implicit flow	-----

Credential configuration

Credentials are a vital part of an application registration when your application is used as a confidential client. If your app registration is used only as a Public Client App (allows users to sign in using a public endpoint), ensure that you don't have any credentials on your application object. Review the credentials used in your applications for freshness of use and their expiration. An unused credential on an application can result in security breach. While it's convenient to use password secrets as a credential, we strongly recommend that you use x509 certificates as the only credential type for getting tokens for your application. Monitor your production pipelines to ensure credentials of any kind are never committed into code repositories. If using Azure, we strongly recommend using Managed Identity so application credentials are automatically managed. Refer to the [managed identities documentation](#) for more details. [Credential Scanner](#) is a static analysis tool that you can use to detect credentials (and other sensitive content) in your source code and build output.

DO	DON'T
Use certificate credentials	Use Password credentials
Use Key Vault with Managed identities	Share credentials across apps
Rollover frequently	Have many credentials on one app
-----	Leave stale credentials available
-----	Commit credentials in code

AppId URI configuration

Certain applications can expose resources (via WebAPI) and as such need to define an AppId URI that uniquely identifies the resource in a tenant. We recommend using either of the following URI schemes: api or https, and set the AppId URI in the following formats to avoid URI collisions in your organization. The AppId URI acts as the prefix for the scopes referenced in the API's code, and it must use a verified customer owned domain. For multi-tenant applications the value must also be globally unique.

The following API and HTTP scheme-based application ID URI formats are supported. Replace the placeholder values as described in the list following the table.

SUPPORTED APPLICATION ID URI FORMATS	EXAMPLE APP ID URIS
<code>api://<appId></code>	<code>api://fc4d2d73-d05a-4a9b-85a8-4f2b3a5f38ed</code>
<code>api://<tenantId>/<appId></code>	<code>api://a8573488-ff46-450a-b09a-6eca0c6a02dc/fc4d2d73-d05a-4a9b-85a8-4f2b3a5f38ed</code>
<code>api://<tenantId>/<string></code>	<code>api://a8573488-ff46-450a-b09a-6eca0c6a02dc/api</code>

SUPPORTED APPLICATION ID URI FORMATS	EXAMPLE APP ID URIS
<code>api://<string>/<appId></code>	<code>api://productapi/fc4d2d73-d05a-4a9b-85a8-4f2b3a5f38ed</code>
<code>https://<tenantInitialDomain>.onmicrosoft.com/<string></code>	<code>https://contoso.onmicrosoft.com/productsapi</code>
<code>https://<verifiedCustomDomain>/<string></code>	<code>https://contoso.com/productsapi</code>
<code>https://<string>.<verifiedCustomDomain></code>	<code>https://product.contoso.com</code>
<code>https://<string>.<verifiedCustomDomain>/<string></code>	<code>https://product.contoso.com/productsapi</code>

- `<appId>` - The application identifier (`appId`) property of the application object.
- `<string>` - The string value for the host or the api path segment.
- `<tenantId>` - A GUID generated by Azure to represent the tenant within Azure.
- `<tenantInitialDomain>` - `<tenantInitialDomain>.onmicrosoft.com`, where `<tenantInitialDomain>` is the initial domain name the tenant creator specified at tenant creation.
- `<verifiedCustomDomain>` - A [verified custom domain](#) configured for your Azure AD tenant.

NOTE

If you use the `api://` scheme, you add a string value directly after the "api://". For example, `api://<string>`. That string value can be a GUID or an arbitrary string. If you add a GUID value, it must match either the app ID or the tenant ID. The application ID URI value must be unique for your tenant. If you add `api://<tenantId>` as the application ID URI, no one else will be able to use that URI in any other app. The recommendation is to use `api://<appId>`, instead, or the HTTP scheme.

Home > saurabhmsft > MyApp

MyApp | Expose an API

Search (Ctrl+ /) <> Got feedback?

Overview Quickstart Integration assistant

Manage

Branding Authentication Certificates & secrets Token configuration API permissions

Expose an API

App roles | Preview Owners Roles and administrators | Preview Manifest

Support + Troubleshooting

Troubleshooting New support request

Application ID URI Set

Scopes defined by this API

Define custom scopes to restrict access to data and functionality protected by the API. An application that requires access to parts of this API can request that a user or admin consent to one or more of these.

Adding a scope here creates only delegated permissions. If you are looking to create application-only scopes, use 'App roles' and define app roles assignable to application type. [Go to App roles](#).

+ Add a scope

Scopes	Who can consent	Admin consent display ...	User consent display ...	State
No scopes have been defined				

Authorized client applications

Authorizing a client application indicates that this API trusts the application and users should not be asked to consent when the client calls this API.

+ Add a client application

Client Id	Scopes
No client applications have been authorized	

AppId URI summary

DO	DON'T
Avoid collisions by using valid URI formats.	Use wildcard AppId URI
Use verified domain in Line of Business (LoB) apps	Malformed URI
Inventory your AppId URIs	-----
Use AppId URI to expose WebApi in your organization	Use AppId URI to identify the application, instead use the appId property

App ownership configuration

Ensure app ownership is kept to a minimal set of people within the organization. It's recommended to run through the owners list once every few months to ensure owners are still part of the organization and their charter accounts for ownership of the application registration. Check out [Azure AD access reviews](#) for more details.

Owner	Name	Email	Title
AL	Annie Lange	AnnaL@M365x709460.onmicrosoft.com	Paralegal
AN	Adriana Napolitani	AdrianaN@M365x709460.onmicrosoft.com	Salesperson
BP	Bianca Pisani	BiancaP@M365x709460.onmicrosoft.com	Salesperson
AW	Adam Wallen	AdamW@M365x709460.onmicrosoft.com	Engineer
AM	Anne Matthews	AnneM@M365x709460.onmicrosoft.com	Salesperson
CT	Clarissa Trentini	ClarissaT@M365x709460.onmicrosoft.com	Salesperson
CW	Cameron White	CameronW@M365x709460.onmicrosoft.com	Salesperson
CC	Caterina Costa	CaterinaC@M365x709460.onmicrosoft.com	Salesperson
BP	Bianca Pagnotto	BiancaPa@M365x709460.onmicrosoft.com	Salesperson
CP	Claudia Pugliesi	ClaudiaP@M365x709460.onmicrosoft.com	Salesperson
AG	Alisha Guerrero	AlishaG@M365x709460.onmicrosoft.com	Engineer
BD	Betsy Drake	BetsyD@M365x709460.onmicrosoft.com	Salesperson
CD	Candy Dominguez	CandyD@M365x709460.onmicrosoft.com	Engineer
BJ	Brian Johnson (TAILSPIN)	BrianJ@M365x709460.onmicrosoft.com	
CR	Conf Room Adams	Adams@M365x709460.onmicrosoft.com	
AN	Anagha	Anagha@M365x709460.onmicrosoft.com	

App ownership summary

DO	DON'T
Keep it small	-----
Monitor owners list	-----

Checklist

App developers can use the *Checklist* available in Azure portal to ensure their app registration meets a high quality bar and provides guidance to integrate securely. The integration assistant highlights best practices and

recommendation that help avoid common oversights when integrating with Microsoft identity platform.

The screenshot shows the Microsoft Azure portal interface. In the top left, the navigation bar includes 'Home > saurabhmsft > MyApp'. The main title is 'MyApp | Integration assistant'. On the left, a sidebar lists various management options like Overview, Quickstart, and Integration assistant (which is highlighted with a red box). The main content area is titled 'Effectively integrate your application' with a sub-instruction: 'Use the integration assistant to help you get to a high-quality and secure integration. The integration assistant highlights best practices and recommendations and helps you avoid common oversights when integrating with the Microsoft identity platform.' Below this, there's a section titled 'What type of application are you building?' containing several checkboxes for different application types: Single-page app (SPA), Web app, Mobile app (Android, iOS, Xamarin, UWP), Desktop app (Win, Linux, Mac), Web API, and Daemon (background process or automation). Another section asks 'Is this application calling APIs?' with a toggle switch set to 'No'. At the bottom right of the main content area is a button labeled 'Evaluate my app registration'.

Checklist summary

DO	DON'T
Use checklist to get scenario-based recommendation	-----
Deep link into app registration blades	-----

Next steps

For more information on Auth code flow, see the [OAuth 2.0 authorization code flow](#).

Secure access control using groups in Azure AD

4/12/2022 • 3 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) allows the use of groups to manage access to resources in an organization. You should use groups for access control when you want to manage and minimize access to applications. When groups are used, only members of those groups can access the resource. Using groups also allows you to benefit from several Azure AD group management features, such as attribute-based dynamic groups, external groups synced from on-premises Active Directory, and Administrator managed or self-service managed groups. To learn more about the benefits of groups for access control, see [manage access to an application](#).

While developing an application, you can authorize access with the [groups claim](#). To learn more, see how to [configure group claims for applications with Azure AD](#).

Today, many applications select a subset of groups with the *securityEnabled* flag set to *true* to avoid scale challenges, that is, to reduce the number of groups returned in the token. Setting the *securityEnabled* flag to be *true* for a group doesn't guarantee that the group is securely managed. Therefore, we suggest following the best practices described below:

Best practices to mitigate risk

This table presents several security best practices for security groups and the potential security risks each practice mitigates.

SECURITY BEST PRACTICE	SECURITY RISK MITIGATED
Ensure resource owner and group owner are the same principal. Applications should build their own group management experience and create new groups to manage access. For example, an application can create groups with <i>Group.Create</i> permission and add itself as the owner of the group. This way the application has control over its groups without being over privileged to modify other groups in the tenant.	When group owners and resource owners are different users or entities, group owners can add users to the group who aren't supposed to get access to the resource and thus give access to the resource unintentionally.
Build an implicit contract between resource owner(s) and group owner(s). The resource owner and the group owner should align on the group purpose, policies, and members that can be added to the group to get access to the resource. This level of trust is non-technical and relies on human or business contract.	When group owners and resource owners have different intentions, the group owner may add users to the group the resource owner didn't intend on giving access to. This can result in unnecessary and potentially risky access.
Use private groups for access control. Microsoft 365 groups are managed by the visibility concept . This property controls the join policy of the group and visibility of group resources. Security groups have join policies that either allow anyone to join or require owner approval. On-premises-synced groups can also be public or private. When they're used to give access to a resource in the cloud, users joining this group on-premises can get access to the cloud resource as well.	When you use a <i>Public</i> group for access control, any member can join the group and get access to the resource. When a <i>Public</i> group is used to give access to an external resource, the risk of elevation of privilege exists.

SECURITY BEST PRACTICE	SECURITY RISK MITIGATED
<p>Group nesting. When you use a group for access control and it has other groups as its members, members of the subgroups can get access to the resource. In this case, there are multiple group owners - owners of the parent group and the subgroups.</p>	<p>Aligning with multiple group owners on the purpose of each group and how to add the right members to these groups is more complex and more prone to accidental grant of access. Therefore, you should limit the number of nested groups or don't use them at all if possible.</p>

Next steps

For more information about groups in Azure AD, see the following:

- [Manage app and resource access using Azure Active Directory groups](#)
- [Access with Azure Active Directory groups](#)
- [Restrict your Azure AD app to a set of users in an Azure AD tenant](#)

Microsoft identity platform best practices and recommendations

4/12/2022 • 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](#).

TIP

The *Integration assistant* in the Azure portal can help you apply many of these best practices and recommendations. Select any of your [app registrations](#) in the Azure portal, and then select the **Integration assistant** menu item to get started with the assistant.

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

Manage your redirect URIs:

- Maintain ownership of all your redirect URIs and keep the DNS records for them up-to-date.
- Don't use wildcards (*) in your URIs.
- For web apps, make sure all URIs are secure and encrypted (for example, using https schemes).

- For public clients, use platform-specific redirect URIs if applicable (mainly for iOS and Android). Otherwise, use redirect URIs with a high amount of randomness to prevent collisions when calling back to your app.
- If your app is being used from an isolated web agent, you may use <https://login.microsoftonline.com/common/oauth2/nativeclient>.
- Review and trim all unused or unnecessary redirect URIs on a regular basis.

- If your app is registered in a directory, minimize and manually monitor the list of app registration owners.
- Don't enable support for the [OAuth2 implicit grant flow](#) unless explicitly required. Learn about the valid scenario [here](#).
- Move beyond username/password. Don't use [resource owner password credential flow \(ROPC\)](#), 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 [Authentication flows and application scenarios](#).
- Protect and manage your confidential app credentials for web apps, web APIs and daemon apps. Use [certificate credentials](#), 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 [managed identities for Azure resources](#) or [Azure Key Vault](#) to store and regularly rotate your credentials.
- 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 [types of permissions](#). Only use application permissions if necessary; use delegated permissions where possible. For a full list of Microsoft Graph permissions, see this [permissions reference](#).
- 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

- Use modern authentication solutions (OAuth 2.0, [OpenID Connect](#)) to securely sign in users.
- Don't program directly against protocols such as OAuth 2.0 and Open ID. Instead, leverage the [Microsoft Authentication Library \(MSAL\)](#). The MSAL libraries securely wrap security protocols in an easy-to-use library, and you get built-in support for [Conditional Access](#) scenarios, device-wide [single sign-on \(SSO\)](#), and built-in token caching support. For more info, see the list of Microsoft-supported [client libraries](#). If you must hand-code for the authentication protocols, you should follow the [Microsoft SDL](#) or similar development methodology. Pay close attention to the security considerations in the standards specifications for each protocol.
- Migrate existing apps from [Azure Active Directory Authentication Library \(ADAL\)](#) to the [Microsoft Authentication Library](#). 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 [ADAL.NET](#), [ADAL.js](#), and [ADAL.NET and iOS broker](#) apps.
- 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.
- 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 [token cache serialization](#).

- If the data your app requires is available through [Microsoft Graph](#), request permissions for this data using the Microsoft Graph endpoint rather than the individual API.
- 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

- Understand the [consent experience](#) 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.
- 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.
- 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).
- Where applicable, enrich your application with user data. Using the [Microsoft Graph API](#) is an easy way to do this. The [Graph Explorer](#) tool that can help you get started.
- Register the full set of permissions that your app requires so admins can grant consent easily to their tenant. Use [incremental consent](#) at run time to help users understand why your app is requesting permissions that may concern or confuse users when requested on first start.
- Implement a [clean single sign-out experience](#). It's a privacy and a security requirement, and makes for a good user experience.

Testing

- Test for [Conditional Access policies](#) that may affect your users' ability to use your application.
- 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 \(overview\)](#)
- [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](#)

Microsoft identity platform authentication libraries

4/12/2022 • 4 minutes to read • [Edit Online](#)

The following tables show Microsoft Authentication Library support for several application types. They include links to library source code, where to get the package for your app's project, and whether the library supports user sign-in (authentication), access to protected web APIs (authorization), or both.

The Microsoft identity platform has been certified by the OpenID Foundation as a [certified OpenID provider](#). If you prefer to use a library other than the Microsoft Authentication Library (MSAL) or another Microsoft-supported library, choose one with a [certified OpenID Connect implementation](#).

If you choose to hand-code your own protocol-level implementation of [OAuth 2.0 or OpenID Connect 1.0](#), pay close attention to the security considerations in each standard's specification and follow a software development lifecycle (SDL) methodology like the [Microsoft SDL](#).

Single-page application (SPA)

A single-page application runs entirely in the browser and fetches page data (HTML, CSS, and JavaScript) dynamically or at application load time. It can call web APIs to interact with back-end data sources.

Because a SPA's code runs entirely in the browser, it's considered a *public client* that's unable to store secrets securely.

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
Angular	MSAL Angular v2²	msal-angular	Tutorial	✓	✓	GA
Angular	MSAL Angular³	msal-angular	—	✓	✓	GA
AngularJS	MSAL AngularJS³	msal-angularjs	—	✓	✓	Public preview
JavaScript	MSAL.js v2²	msal-browser	Tutorial	✓	✓	GA
JavaScript	MSAL.js 1.0³	msal-core	—	✓	✓	GA
React	MSAL React²	msal-react	Tutorial	✓	✓	GA

¹ [Supplemental terms of use for Microsoft Azure Previews](#) apply to libraries in *Public preview*.

² [Auth code flow with PCKE only \(Recommended\).](#)

³ [Implicit grant flow only.](#)

Web application

A web application runs code on a server that generates and sends HTML, CSS, and JavaScript to a user's web

browser to be rendered. The user's identity is maintained as a session between the user's browser (the front end) and the web server (the back end).

Because a web application's code runs on the web server, it's considered a *confidential client* that can store secrets securely.

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
.NET	MSAL.NET	Microsoft.Identity.Client	—	✗	✓	GA
.NET	Microsoft.IdentityModel	Microsoft.IdentityModel	—	✗ ²	✗ ²	GA
ASP.NET Core	ASP.NET Core	Microsoft.AspNetCore.Authentication	Quickstart	✓	✗	GA
ASP.NET Core	Microsoft.Identity.Web	Microsoft.Identity.Web	Quickstart	✓	✓	GA
Java	MSAL4J	msal4j	Quickstart	✓	✓	GA
Node.js	MSAL Node	msal-node	Quickstart	✓	✓	GA
Python	MSAL Python	msal	Quickstart	✓	✓	GA

¹ Supplemental terms of use for Microsoft Azure Previews apply to libraries in *Public preview*.

² The [Microsoft.IdentityModel](#) library only *validates* tokens - it cannot request ID or access tokens.

Desktop application

A desktop application is typically binary (compiled) code that displays a user interface and is intended to run on a user's desktop.

Because a desktop application runs on the user's desktop, it's considered a *public client* that's unable to store secrets securely.

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
Electron	MSAL Node.js	msal-node	—	✓	✓	Public preview
Java	MSAL4J	msal4j	—	✓	✓	GA
macOS (Swift/Obj-C)	MSAL for iOS and macOS	MSAL	Tutorial	✓	✓	GA
UWP	MSAL.NET	Microsoft.Identity.Client	Tutorial	✓	✓	GA

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW
WPF	MSAL.NET	Microsoft.Identity.Client	Tutorial	✓	✓	GA

¹ Supplemental terms of use for Microsoft Azure Previews apply to libraries in *Public preview*.

Mobile application

A mobile application is typically binary (compiled) code that displays a user interface and is intended to run on a user's mobile device.

Because a mobile application runs on the user's mobile device, it's considered a *public client* that's unable to store secrets securely.

PLATFORM	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
Android (Java)	MSAL Android	MSAL	Quickstart	✓	✓	GA
Android (Kotlin)	MSAL Android	MSAL	—	✓	✓	GA
iOS (Swift/Obj-C)	MSAL for iOS and macOS	MSAL	Tutorial	✓	✓	GA
Xamarin (.NET)	MSAL.NET	Microsoft.Identity.Client	—	✓	✓	GA

¹ Supplemental terms of use for Microsoft Azure Previews apply to libraries in *Public preview*.

Service / daemon

Services and daemons are commonly used for server-to-server and other unattended (sometimes called *headless*) communication. Because there's no user at the keyboard to enter credentials or consent to resource access, these applications authenticate as themselves, not a user, when requesting authorized access to a web API's resources.

A service or daemon that runs on a server is considered a *confidential client* that can store its secrets securely.

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
.NET	MSAL.NET	Microsoft.Identity.Client	Quickstart	✗	✓	GA
Java	MSAL4J	msal4j	—	✗	✓	GA

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW
Node	MSAL Node	<code>msal-node</code>	Quickstart			GA
Python	MSAL Python	<code>msal-python</code>	Quickstart			GA

¹ [Supplemental terms of use for Microsoft Azure Previews](#) apply to libraries in *Public preview*.

Next steps

For more information about the Microsoft Authentication Library, see the [Overview of the Microsoft Authentication Library \(MSAL\)](#).

Quickstart: Set up a tenant

4/12/2022 • 2 minutes to read • [Edit Online](#)

To build apps that use the Microsoft identity platform for identity and access management, you need access to an Azure Active Directory (Azure AD) *tenant*. It's in the Azure AD tenant that you register and manage your apps, configure their access to data in Microsoft 365 and other web APIs, and enable features like Conditional Access.

A tenant represents an organization. It's a dedicated instance of Azure AD that an organization or app developer receives at the beginning of a relationship with Microsoft. That relationship could start with signing up for Azure, Microsoft Intune, or Microsoft 365, for example.

Each Azure AD tenant is distinct and separate from other Azure AD tenants. It 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 your tenant can allow authentications only from accounts within your tenant or all tenants.

Prerequisites

An Azure account that has an active subscription. [Create an account for free](#).

Determining the environment type

You can create two types of environments. The environment depends solely on the types of users your app will authenticate.

This quickstart addresses two scenarios for the type of app you want to build:

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

Work and school accounts, or personal Microsoft accounts

To build an environment for either work and school accounts or personal Microsoft accounts, you can use an existing Azure AD tenant or create a new one.

Use an existing Azure AD tenant

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

To check the tenant:

1. Sign in to the [Azure portal](#). Use the account you'll use to manage your application.
2. Check the upper-right corner. If you have a tenant, you'll automatically be signed in. You see the tenant name directly under your account name.
 - Hover over your account name to see your name, email address, directory or tenant ID (a GUID), and 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

To find the tenant ID, you can:

- Hover over your account name to get the directory or tenant ID.
- Search and select **Azure Active Directory > Properties > Tenant ID** in the Azure portal.

If you don't have a tenant associated with your account, you'll see a GUID under your account name. You won't be able to do actions like registering apps until you create an Azure AD tenant.

Create a new Azure AD tenant

If you don't already have an Azure AD tenant or if you want to create a new one for development, see [Create a new tenant in Azure AD](#). Or use the [directory creation experience](#) in the Azure portal.

You'll provide the following information to create your new tenant:

- **Organization name**
- **Initial domain** - Initial domain `<domainname>.onmicrosoft.com` can't be edited or deleted. You can add a customized domain name later.
- **Country or region**

NOTE

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

Social and local accounts

To begin building apps that sign in social and local accounts, create an Azure AD B2C tenant. To begin, see [Create an Azure AD B2C tenant](#).

Next steps

[Register an app](#) to integrate with Microsoft identity platform.

Quickstart: Register an application with the Microsoft identity platform

4/12/2022 • 7 minutes to read • [Edit Online](#)

Get started with the Microsoft identity platform by registering an application in the Azure portal.

The Microsoft identity platform performs identity and access management (IAM) only for registered applications. Whether it's a client application like a web or mobile app, or it's a web API that backs a client app, registering it establishes a trust relationship between your application and the identity provider, the Microsoft identity platform.

TIP

To register an application for Azure AD B2C, follow the steps in [Tutorial: Register a web application in Azure AD B2C](#).

Prerequisites

- An Azure account that has an active subscription. [Create an account for free](#).
- The Azure account must have permission to manage applications in Azure Active Directory (Azure AD). Any of the following Azure AD roles include the required permissions:
 - [Application administrator](#)
 - [Application developer](#)
 - [Cloud application administrator](#)
- Completion of the [Set up a tenant](#) quickstart.

Register an application

Registering your application establishes a trust relationship between your app and the Microsoft identity platform. The trust is unidirectional: your app trusts the Microsoft identity platform, and not the other way around.

Follow these steps to create the app registration:

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a display **Name** for your application. Users of your application might see the display name when they use the app, for example during sign-in. You can change the display name at any time and multiple app registrations can share the same name. The app registration's automatically generated Application (client) ID, not its display name, uniquely identifies your app within the identity platform.
6. Specify who can use the application, sometimes called its *sign-in audience*.

SUPPORTED ACCOUNT TYPES	DESCRIPTION
Accounts in this organizational directory only	<p>Select this option if you're building an application for use only by users (or guests) in <i>your</i> tenant.</p> <p>Often called a <i>line-of-business</i> (LOB) application, this app is a <i>single-tenant</i> application in the Microsoft identity platform.</p>
Accounts in any organizational directory	<p>Select this option if you want users in <i>any</i> Azure Active Directory (Azure AD) tenant to be able to use your application. This option is appropriate if, for example, you're building a software-as-a-service (SaaS) application that you intend to provide to multiple organizations.</p> <p>This type of app is known as a <i>multitenant</i> application in the Microsoft identity platform.</p>
Accounts in any organizational directory and personal Microsoft accounts	<p>Select this option to target the widest set of customers.</p> <p>By selecting this option, you're registering a <i>multitenant</i> application that can also support users who have personal <i>Microsoft accounts</i>.</p>
Personal Microsoft accounts	<p>Select this option if you're building an application only for users who have personal Microsoft accounts.</p> <p>Personal Microsoft accounts include Skype, Xbox, Live, and Hotmail accounts.</p>

7. Don't enter anything for **Redirect URI (optional)**. You'll configure a redirect URI in the next section.

8. Select **Register** to complete the initial app registration.

The screenshot shows the Microsoft Azure portal with the URL <https://portal.azure.com>. The user is logged in as meganb@contoso.com (CONTOSO AD (DEV)). The page title is "Register an application". The main form has a required field for "Name" and a section for "Supported account types" with four options. Below that is a "Redirect URI (optional)" field and a "By proceeding, you agree to the Microsoft Platform Policies" link. At the bottom is a "Register" button.

* 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 (Contoso AD (dev) only - Single tenant)
 Accounts in any organizational directory (Any Azure AD directory - Multitenant)
 Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)
 Personal Microsoft accounts only

[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

By proceeding, you agree to the Microsoft Platform Policies [↗](#)

[Register](#)

When registration finishes, the Azure portal displays the app registration's **Overview** pane. You see the **Application (client) ID**. Also called the *client ID*, this value uniquely identifies your application in the Microsoft identity platform.

IMPORTANT

New app registrations are hidden to users by default. When you are ready for users to see the app on their [My Apps](#) page you can enable it. To enable the app, in the Azure portal navigate to [Azure Active Directory > Enterprise applications](#) and select the app. Then on the [Properties](#) page toggle **Visible to users?** to Yes.

Your application's code, or more typically an authentication library used in your application, also uses the client ID. The ID is used as part of validating the security tokens it receives from the identity platform.

Contoso App 1

Search (Ctrl+/)

Overview Quickstart Integration assistant (preview) Manage Branding Authentication Certificates & secrets Token configuration API permissions Expose an API

Delete Endpoints

Display name: Contoso App 1 Application (client) ID: 11111111-1111-1111-1111-111111111111 Directory (tenant) ID: 00000000-0000-0000-0000-000000000000 Object ID: 2222222-2222-2222-2222-222222222222

Supported account types: My organization only Redirect URLs: Add a Redirect URI Application ID URI: Add an Application ID URI Managed application in local directory: Contoso App 1

Info Starting June 30th, 2020 we will no longer add any new features to Azure Active Directory Authentication Library (ADAL) and Azure AD Graph. We will continue to provide technical support and security updates but we will no longer provide feature updates. Applications will need to be upgraded to Microsoft Authentication Library (MSAL) and Microsoft Graph. [Learn more](#)

Add a redirect URI

A *redirect URI* is the location where the Microsoft identity platform redirects a user's client and sends security tokens after authentication.

In a production web application, for example, the redirect URI is often a public endpoint where your app is running, like `https://contoso.com/auth-response`. During development, it's common to also add the endpoint where you run your app locally, like `https://127.0.0.1/auth-response` or `http://localhost/auth-response`.

You add and modify redirect URIs for your registered applications by configuring their [platform settings](#).

Configure platform settings

Settings for each application type, including redirect URIs, are configured in **Platform configurations** in the Azure portal. Some platforms, like **Web** and **Single-page applications**, require you to manually specify a redirect URI. For other platforms, like mobile and desktop, you can select from redirect URIs generated for you when you configure their other settings.

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

1. In the Azure portal, in **App registrations**, select your application.
2. Under **Manage**, select **Authentication**.
3. Under **Platform configurations**, select **Add a platform**.
4. Under **Configure platforms**, select the tile for your application type (platform) to configure its settings.

Configure platforms

X

Web applications



Web

Build, host, and deploy a web server application. .NET, Java, Python



Single-page application

Configure browser client applications and progressive web applications. Javascript.

Mobile and desktop applications



iOS / macOS

Objective-C, Swift, Xamarin



Android

Java, Kotlin, Xamarin



Mobile and desktop applications

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

PLATFORM	CONFIGURATION SETTINGS
Web	<p>Enter a Redirect URI for your app. This URI is the location where the Microsoft identity platform redirects a user's client and sends security tokens after authentication.</p> <p>Select this platform for standard web applications that run on a server.</p>
Single-page application	<p>Enter a Redirect URI for your app. This URI is the location where the Microsoft identity platform redirects a user's client and sends security tokens after authentication.</p> <p>Select this platform if you're building a client-side web app by using JavaScript or a framework like Angular, Vue.js, React.js, or Blazor WebAssembly.</p>
iOS / macOS	<p>Enter the app Bundle ID. Find it in Build Settings or in Xcode in <i>Info.plist</i>.</p> <p>A redirect URI is generated for you when you specify a Bundle ID.</p>
Android	<p>Enter the app Package name. Find it in the <i>AndroidManifest.xml</i> file. Also generate and enter the Signature hash.</p> <p>A redirect URI is generated for you when you specify these settings.</p>

PLATFORM	CONFIGURATION SETTINGS
Mobile and desktop applications	<p>Select one of the Suggested redirect URIs. Or specify a Custom redirect URI.</p> <p>For desktop applications using embedded browser, we recommend <code>https://login.microsoftonline.com/common/oauth2/nativeclient</code></p> <p>For desktop applications using system browser, we recommend <code>http://localhost</code></p> <p>Select this platform for mobile applications that aren't using the latest Microsoft Authentication Library (MSAL) or aren't using a broker. Also select this platform for desktop applications.</p>

5. Select **Configure** to complete the platform configuration.

Redirect URI restrictions

There are some restrictions on the format of the redirect URIs you add to an app registration. For details about these restrictions, see [Redirect URI \(reply URL\) restrictions and limitations](#).

Add credentials

Credentials are used by [confidential client applications](#) that access a web API. Examples of confidential clients are web apps, other web APIs, or service-type and daemon-type applications. Credentials allow your application to authenticate as itself, requiring no interaction from a user at runtime.

You can add both certificates and client secrets (a string) as credentials to your confidential client app registration.

Contoso App 1 | Certificates & secrets

Overview Quickstart Integration assistant

Branding Authentication

Certificates & secrets (0) Client secrets (0) Federated credentials (0)

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	Secret ID
No client secrets have been created for this application.			

Add a certificate

Sometimes called a *public key*, a certificate is the recommended credential type because they're considered more secure than client secrets. For more information about using a certificate as an authentication method in your application, see [Microsoft identity platform application authentication certificate credentials](#).

1. In the Azure portal, in **App registrations**, select your application.
2. Select **Certificates & secrets** > **Certificates** > **Upload certificate**.
3. Select the file you want to upload. It must be one of the following file types: *.cer*, *.pem*, *.crt*.
4. Select **Add**.

Add a client secret

Sometimes called an *application password*, a client secret is a string value your app can use in place of a certificate to identify itself.

Client secrets are considered less secure than certificate credentials. Application developers sometimes use client secrets during local app development because of their ease of use. However, you should use certificate credentials for any of your applications that are running in production.

1. In the Azure portal, in **App registrations**, select your application.
2. Select **Certificates & secrets** > **Client secrets** > **New client secret**.
3. Add a description for your client secret.
4. Select an expiration for the secret or specify a custom lifetime.
 - Client secret lifetime is limited to two years (24 months) or less. You can't specify a custom lifetime longer than 24 months.
 - Microsoft recommends that you set an expiration value of less than 12 months.
5. Select **Add**.
6. *Record the secret's value* for use in your client application code. This secret value is *never displayed again* after you leave this page.

For application security recommendations, see [Microsoft identity platform best practices and recommendations](#).

Next steps

Client applications typically need to access resources in a web API. You can protect your client application by using the Microsoft identity platform. You can also use the platform for authorizing scoped, permissions-based access to your web API.

Go to the next quickstart in the series to create another app registration for your web API and expose its scopes.

[Configure an application to expose a web API](#)

Quickstart: Configure an application to expose a web API

4/12/2022 • 5 minutes to read • [Edit Online](#)

In this quickstart, you register a web API with the Microsoft identity platform and expose it to client apps by adding an example scope. By registering your web API and exposing it through scopes, you can provide permissions-based access to its resources to authorized users and client apps that access your API.

Prerequisites

- An Azure account with an active subscription - [create an account for free](#)
- Completion of [Quickstart: Set up a tenant](#)

Register the web API

To provide scoped access to the resources in your web API, you first need to register the API with the Microsoft identity platform.

1. Perform the steps in the **Register an application** section of [Quickstart: Register an app with the Microsoft identity platform](#).
2. Skip the **Add a redirect URI** and **Configure platform settings** sections. You don't need to configure a redirect URI for a web API since no user is logged in interactively.
3. Skip the **Add credentials** section for now. Only if your API accesses a downstream API would it need its own credentials, a scenario not covered in this article.

With your web API registered, you're ready to add the scopes that your API's code can use to provide granular permission to consumers of your API.

Add a scope

The code in a client application requests permission to perform operations defined by your web API by passing an access token along with its requests to the protected resource (the web API). Your web API then performs the requested operation only if the access token it receives contains the scopes required for the operation.

First, follow these steps to create an example scope named `Employees.Read.All`:

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to select the tenant containing your client app's registration.
3. Select **Azure Active Directory > App registrations**, and then select your API's app registration.
4. Select **Expose an API**
5. Select **Set** next to **Application ID URI** if you haven't yet configured one.

You can use the default value of `api://<application-client-id>` or another [supported App ID URI pattern](#).

The App ID URI acts as the prefix for the scopes you'll reference in your API's code, and it must be globally unique.

6. Select **Add a scope**:

The screenshot shows the Microsoft Azure portal interface. At the top, it says 'Contoso API 1 - Microsoft Azure' and the URL 'https://portal.azure.com'. Below that is the 'Microsoft Azure' header with a search bar and user info 'meganb@contoso.com CONTOSO AD (DEV)'. The main content area has a title 'Contoso API 1 | Expose an API'. On the left, there's a sidebar with 'Overview', 'Quickstart', 'Integration assistant (preview)', 'Manage' (with 'Branding', 'Authentication', 'Certificates & secrets', 'Token configuration', 'API permissions', and 'Expose an API' which is selected and highlighted with a red box), 'Owners', 'Roles and administrators (Preview)', and 'Manifest'. Under 'Support + Troubleshooting', there are 'Troubleshooting' and 'New support request'. The main content area shows 'Scopes defined by this API' with a sub-section 'Add a scope' (which is also highlighted with a red box). It includes tabs for 'Scopes', 'Who can consent', 'Admin consent disp...', 'User consent displa...', and 'State'. A message says 'No scopes have been defined'. Below this, there's a section for 'Authorized client applications' with a sub-section 'Add a client application'. It includes 'Client Id' and 'Scopes' tabs, with a message 'No client applications have been authorized'.

7. Next, specify the scope's attributes in the **Add a scope** pane. For this walk-through, you can use the example values or specify your own.

FIELD	DESCRIPTION	EXAMPLE
Scope name	The name of your scope. A common scope naming convention is <code>resource.operation.constraint</code> .	<code>Employees.Read.All</code>
Who can consent	Whether this scope can be consented to by users or if admin consent is required. Select Admins only for higher-privileged permissions.	Admins and users
Admin consent display name	A short description of the scope's purpose that only admins will see.	Read-only access to Employee records
Admin consent description	A more detailed description of the permission granted by the scope that only admins will see.	Allow the application to have read-only access to all Employee data.
User consent display name	A short description of the scope's purpose. Shown to users only if you set Who can consent to Admins and users .	Read-only access to your Employee records
User consent description	A more detailed description of the permission granted by the scope. Shown to users only if you set Who can consent to Admins and users .	Allow the application to have read-only access to your Employee data.

- Set the **State** to **Enabled**, and then select **Add scope**.
- (Optional) To suppress prompting for consent by users of your app to the scopes you've defined, you can *pre-authorize* the client application to access your web API. Pre-authorize *only* those client applications you trust since your users won't have the opportunity to decline consent.
 - Under **Authorized client applications**, select **Add a client application**
 - Enter the **Application (client) ID** of the client application you want to pre-authorize. For example, that of a web application you've previously registered.
 - Under **Authorized scopes**, select the scopes for which you want to suppress consent prompting, then select **Add application**.

If you followed this optional step, the client app is now a pre-authorized client app (PCA), and users won't be prompted for their consent when signing in to it.

Add a scope requiring admin consent

Next, add another example scope named `Employees.Write.All` that only admins can consent to. Scopes that require admin consent are typically used for providing access to higher-privileged operations, and often by client applications that run as backend services or daemons that don't sign in a user interactively.

To add the `Employees.Write.All` example scope, follow the steps in the [Add a scope](#) section and specify these values in the **Add a scope** pane:

FIELD	EXAMPLE VALUE
Scope name	<code>Employees.Write.All</code>
Who can consent	Admins only
Admin consent display name	Write access to Employee records
Admin consent description	Allow the application to have write access to all Employee data.
User consent display name	<i>None (leave empty)</i>
User consent description	<i>None (leave empty)</i>

Verify the exposed scopes

If you successfully added both example scopes described in the previous sections, they'll appear in the [Expose an API](#) pane of your web API's app registration, similar to this image:

Scopes	Who can consent	Admin consent display ...	User consent display na...	State
https://contoso.com/api1/Employees.Read.All	Admins and users	Read-only access to Empl...	Read-only access to your ...	Enabled
https://contoso.com/api1/Employees.Write.All	Admins only	Write access to Employee...		Enabled

As shown in the image, a scope's full string is the concatenation of your web API's **Application ID URI** and the scope's **Scope name**.

For example, if your web API's application ID URI is `https://contoso.com/api` and the scope name is `Employees.Read.All`, the full scope is:

`https://contoso.com/api/Employees.Read.All`

Using the exposed scopes

In the next article in the series, you configure a client app's registration with access to your web API and the scopes you defined by following the steps this article.

Once a client app registration is granted permission to access your web API, the client can be issued an OAuth 2.0 access token by the Microsoft identity platform. When the client calls the web API, it presents an access token whose scope (`scp`) claim is set to the permissions you've specified in the client's app registration.

You can expose additional scopes later as necessary. Consider that your web API can expose multiple scopes associated with several operations. Your resource can control access to the web API at runtime by evaluating the scope (`scp`) claim(s) in the OAuth 2.0 access token it receives.

Next steps

Now that you've exposed your web API by configuring its scopes, configure your client app's registration with permission to access the scopes.

[Configure an app registration for web API access](#)

Quickstart: Configure a client application to access a web API

4/12/2022 • 7 minutes to read • [Edit Online](#)

In this quickstart, you provide a client app registered with the Microsoft identity platform with scoped, permissions-based access to your own web API. You also provide the client app access to Microsoft Graph.

By specifying a web API's scopes in your client app's registration, the client app can obtain an access token containing those scopes from the Microsoft identity platform. Within its code, the web API can then provide permission-based access to its resources based on the scopes found in the access token.

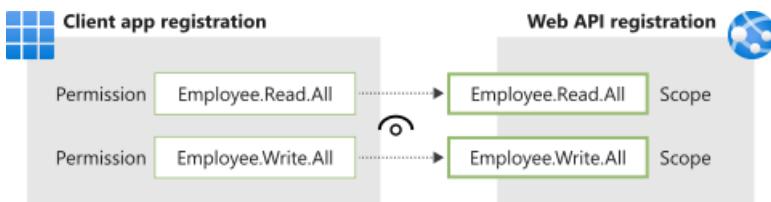
Prerequisites

- An Azure account with an active subscription - [create an account for free](#)
- Completion of [Quickstart: Register an application](#)
- Completion of [Quickstart: Configure an application to expose a web API](#)

Add permissions to access your web API

In the first scenario, you grant a client app access to your own web API, both of which you should have registered as part of the prerequisites. If you don't yet have both a client app and a web API registered, complete the steps in the two [Prerequisites](#) articles.

This diagram shows how the two app registrations relate to one another. In this section, you add permissions to the client app's registration.



Once you've registered both your client app and web API and you've exposed the API by creating scopes, you can configure the client's permissions to the API by following these steps:

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the [Directory + subscription](#) filter in the top menu to select the tenant containing your client app's registration.
3. Select **Azure Active Directory > App registrations**, and then select your client application (*not* your web API).
4. Select **API permissions > Add a permission > My APIs**.
5. Select the web API you registered as part of the prerequisites.

Delegated permissions is selected by default. Delegated permissions are appropriate for client apps that access a web API as the signed-in user, and whose access should be restricted to the permissions you select in the next step. Leave **Delegated permissions** selected for this example.

Application permissions are for service- or daemon-type applications that need to access a web API as

themselves, without user interaction for sign-in or consent. Unless you've defined application roles for your web API, this option is disabled.

6. Under **Select permissions**, expand the resource whose scopes you defined for your web API, and select the permissions the client app should have on behalf of the signed-in user.

If you used the example scope names specified in the previous quickstart, you should see **Employees.Read.All** and **Employees.Write.All**. Select **Employees.Read.All** or another permission you might have created when completing the prerequisites.

7. Select **Add permissions** to complete the process.

After adding permissions to your API, you should see the selected permissions under **Configured permissions**. The following image shows the example *Employees.Read.All* delegated permission added to the client app's registration.

Configured permissions				
Applications are authorized to call APIs when they are granted permissions by users/admins as part of the consent process. The list of configured permissions should include all the permissions the application needs. Learn more about permissions and consent				
+	Add a permission	✓ Grant admin consent for Contoso AD (dev)		
✓	Contoso API 1 (1)		Admin consent req...	Status
	Employees.Read.All	Delegated	Read-only access to Employee records	-
✓	Microsoft Graph (1)			...
	User.Read	Delegated	Sign in and read user profile	-

You might also notice the *User.Read* permission for the Microsoft Graph API. This permission is added automatically when you register an app in the Azure portal.

Add permissions to access Microsoft Graph

In addition to accessing your own web API on behalf of the signed-in user, your application might also need to access or modify the user's (or other) data stored in Microsoft Graph. Or you might have service or daemon app that needs to access Microsoft Graph as itself, performing operations without any user interaction.

Delegated permission to Microsoft Graph

Configure delegated permission to Microsoft Graph to enable your client application to perform operations on behalf of the logged-in user, for example reading their email or modifying their profile. By default, users of your client app are asked when they sign in to consent to the delegated permissions you've configured for it.

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directory + subscription** filter  in the top menu to select the tenant containing your client app's registration.
3. Select **Azure Active Directory > App registrations**, and then select your client application.
4. Select **API permissions > Add a permission > Microsoft Graph**
5. Select **Delegated permissions**. Microsoft Graph exposes many permissions, with the most commonly used shown at the top of the list.
6. Under **Select permissions**, select the following permissions:

PERMISSION	DESCRIPTION
email	View users' email address

PERMISSION	DESCRIPTION
<code>offline_access</code>	Maintain access to data you have given it access to
<code>openid</code>	Sign users in
<code>profile</code>	View users' basic profile

7. Select Add permissions to complete the process.

Whenever you configure permissions, users of your app are asked at sign-in for their consent to allow your app to access the resource API on their behalf.

As an admin, you can also grant consent on behalf of *all* users so they're not prompted to do so. Admin consent is discussed later in the [More on API permissions and admin consent](#) section of this article.

Application permission to Microsoft Graph

Configure application permissions for an application that needs to authenticate as itself without user interaction or consent. Application permissions are typically used by background services or daemon apps that access an API in a "headless" manner, and by web APIs that access another (downstream) API.

In the following steps, you grant permission to Microsoft Graph's `Files.Read.All` permission as an example.

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directory + subscription** filter  in the top menu to select the tenant containing your client app's registration.
3. Select **Azure Active Directory > App registrations**, and then select your client application.
4. Select **API permissions > Add a permission > Microsoft Graph > Application permissions**.
5. All permissions exposed by Microsoft Graph are shown under **Select permissions**.
6. Select the permission or permissions you want to grant your application. As an example, you might have a daemon app that scans files in your organization, alerting on a specific file type or name.

Under **Select permissions**, expand **Files**, and then select the `Files.Read.All` permission.

7. Select **Add permissions**.

Some permissions, like Microsoft Graph's `Files.Read.All` permission, require admin consent. You grant admin consent by selecting the **Grant admin consent** button, discussed later in the [Admin consent button](#) section.

Configure client credentials

Apps that use application permissions authenticate as themselves by using their own credentials, without requiring any user interaction. Before your application (or API) can access Microsoft Graph, your own web API, or another API by using application permissions, you must configure that client app's credentials.

For more information about configuring an app's credentials, see the [Add credentials](#) section of [Quickstart: Register an application with the Microsoft identity platform](#).

More on API permissions and admin consent

The **API permissions** pane of an app registration contains a [Configured permissions](#) table, and might also contain an [Other permissions granted](#) table. Both tables and the [Admin consent button](#) are described in the following sections.

Configured permissions

The **Configured permissions** table on the API permissions pane shows the list of permissions that your application requires for basic operation - the *required resource access* (RRA) list. Users, or their admins, will need to consent to these permissions before using your app. Other, optional permissions can be requested later at runtime (using dynamic consent).

This is the minimum list of permissions people will have to consent to for your app. There could be more, but these will always be required. For security and to help users and admins feel more comfortable using your app, never ask for anything you don't need.

You can add or remove the permissions that appear in this table by using the steps outlined above or from [Other permissions granted](#) (described in the next section). As an admin, you can grant admin consent for the full set of an API's permissions that appear in the table, and revoke consent for individual permissions.

Other permissions granted

You might also see a table titled **Other permissions granted for {your tenant}** on the API permissions pane. The **Other permissions granted for {your tenant}** table shows permissions granted tenant-wide for the tenant that haven't been explicitly configured on the application object. These permissions were dynamically requested and consented to by an admin, on behalf of all users. This section appears only if there is at least one permission that applies.

You can add the full set of an API's permissions or individual permissions appearing this table to the **Configured permissions** table. As an admin, you can revoke admin consent for APIs or individual permissions in this section.

Admin consent button

The **Grant admin consent for {your tenant}** button allows an admin to grant admin consent to the permissions configured for the application. When you select the button, a dialog is shown requesting that you confirm the consent action.

Configured permissions					
Applications are authorized to call APIs when they are granted permissions by users/admins as part of the consent process. The list of configured permissions should include all the permissions the application needs. Learn more about permissions and consent					
+ Add a permission		Grant admin consent for Contoso AD (dev)			
API / Permissions name	Type	Description	Admin consent req...	Status	
Microsoft Graph (1)					...
Files.Read.All	Application	Read files in all site collections	Yes	⚠ Not granted for Contoso AD

After granting consent, the permissions that required admin consent are shown as having consent granted:

Configured permissions					
Applications are authorized to call APIs when they are granted permissions by users/admins as part of the consent process. The list of configured permissions should include all the permissions the application needs. Learn more about permissions and consent					
+ Add a permission		Grant admin consent for Contoso AD (dev)			
API / Permissions name	Type	Description	Admin consent req...	Status	
Microsoft Graph (1)					...
Files.Read.All	Application	Read files in all site collections	Yes	✓ Granted for Contoso AD

The **Grant admin consent** button is *disabled* if you aren't an admin or if no permissions have been configured for the application. If you have permissions that have been granted but not yet configured, the admin consent button prompts you to handle these permissions. You can add them to configured permissions or remove them.

Next steps

Advance to the next quickstart in the series to learn how to configure which account types can access your application. For example, you might want to limit access only to those users in your organization (single-tenant)

or allow users in other Azure AD tenants (multi-tenant) and those with personal Microsoft accounts (MSA).

[Modify the accounts supported by an application](#)

Create a self-signed public certificate to authenticate your application

4/12/2022 • 4 minutes to read • [Edit Online](#)

Azure Active Directory (Azure AD) supports two types of authentication for service principals: **password-based authentication** (app secret) and **certificate-based authentication**. While app secrets can easily be created in the Azure portal, it's recommended that your application uses a certificate.

For testing, you can use a self-signed public certificate instead of a Certificate Authority (CA)-signed certificate. This article shows you how to use Windows PowerShell to create and export a self-signed certificate.

Caution

Using a self-signed certificate is only recommended for development, not production.

You configure various parameters for the certificate. For example, the cryptographic and hash algorithms, the certificate validity period, and your domain name. Then export the certificate with or without its private key depending on your application needs.

The application that initiates the authentication session requires the private key while the application that confirms the authentication requires the public key. So, if you're authenticating from your PowerShell desktop app to Azure AD, you only export the public key (`.cer` file) and upload it to the Azure portal. Your PowerShell app uses the private key from your local certificate store to initiate authentication and obtain access tokens for Microsoft Graph.

Your application may also be running from another machine, such as Azure Automation. In this scenario, you export the public and private key pair from your local certificate store, upload the public key to the Azure portal, and the private key (a `.pfx` file) to Azure Automation. Your application running in Azure Automation will use the private key to initiate authentication and obtain access tokens for Microsoft Graph.

This article uses the `New-SelfSignedCertificate` PowerShell cmdlet to create the self-signed certificate and the `Export-Certificate` cmdlet to export it to a location that is easily accessible. These cmdlets are built-in to modern versions of Windows (Windows 8.1 and greater, and Windows Server 2012R2 and greater). The self-signed certificate will have the following configuration:

- A 2048-bit key length. While longer values are supported, the 2048-bit size is highly recommended for the best combination of security and performance.
- Uses the RSA cryptographic algorithm. Azure AD currently supports only RSA.
- The certificate is signed with the SHA256 hash algorithm. Azure AD also supports certificates signed with SHA384 and SHA512 hash algorithms.
- The certificate is valid for only one year.
- The certificate is supported for use for both client and server authentication.

NOTE

To customize the start and expiry date as well as other properties of the certificate, see the [New-SelfSignedCertificate](#) reference.

Option 1: Create and export your public certificate without a private key

Use the certificate you create using this method to authenticate from an application running from your machine. For example, authenticate from Windows PowerShell.

In an elevated PowerShell prompt, run the following command and leave the PowerShell console session open. Replace `{certificateName}` with the name that you wish to give to your certificate.

```
$certname = "{certificateName}" ## Replace {certificateName}
$cert = New-SelfSignedCertificate -Subject "CN=$certname" -CertStoreLocation "Cert:\CurrentUser\My" -
KeyExportPolicy Exportable -KeySpec Signature -KeyLength 2048 -KeyAlgorithm RSA -HashAlgorithm SHA256
```

The `$cert` variable in the previous command stores your certificate in the current session and allows you to export it. The command below exports the certificate in `.cer` format. You can also export it in other formats supported on the Azure portal including `.pem` and `.crt`.

```
Export-Certificate -Cert $cert -FilePath "C:\Users\admin\Desktop\$certname.cer" ## Specify your preferred
location
```

Your certificate is now ready to upload to the Azure portal. Once uploaded, retrieve the certificate thumbprint for use to authenticate your application.

Option 2: Create and export your public certificate with its private key

Use this option to create a certificate and its private key if your application will be running from another machine or cloud, such as Azure Automation.

In an elevated PowerShell prompt, run the following command and leave the PowerShell console session open. Replace `{certificateName}` with name that you wish to give your certificate.

```
$certname = "{certificateName}" ## Replace {certificateName}
$cert = New-SelfSignedCertificate -Subject "CN=$certname" -CertStoreLocation "Cert:\CurrentUser\My" -
KeyExportPolicy Exportable -KeySpec Signature -KeyLength 2048 -KeyAlgorithm RSA -HashAlgorithm SHA256
```

The `$cert` variable in the previous command stores your certificate in the current session and allows you to export it. The command below exports the certificate in `.cer` format. You can also export it in other formats supported on the Azure portal including `.pem` and `.crt`.

```
Export-Certificate -Cert $cert -FilePath "C:\Users\admin\Desktop\$certname.cer" ## Specify your preferred
location
```

Still in the same session, create a password for your certificate private key and save it in a variable. In the following command, replace `{myPassword}` with the password that you wish to use to protect your certificate private key.

```
$mypwd = ConvertTo-SecureString -String "{myPassword}" -Force -AsPlainText ## Replace {myPassword}
```

Now, using the password you stored in the `$mypwd` variable, secure, and export your private key.

```
Export-PfxCertificate -Cert $cert -FilePath "C:\Users\admin\Desktop\$certname.pfx" -Password $mypwd ##  
Specify your preferred location
```

Your certificate (.cer file) is now ready to upload to the Azure portal. You also have a private key (.pfx file) that is encrypted and can't be read by other parties. Once uploaded, retrieve the certificate thumbprint for use to authenticate your application.

Optional task: Delete the certificate from the keystore.

If you created the certificate using Option 2, you can delete the key pair from your personal store. First, run the following command to retrieve the certificate thumbprint.

```
Get-ChildItem -Path "Cert:\CurrentUser\My" | Where-Object {$_.Subject -Match "$certname"} | Select-Object  
Thumbprint, FriendlyName
```

Then, copy the thumbprint that is displayed and use it to delete the certificate and its private key.

```
Remove-Item -Path Cert:\CurrentUser\My\{pasteTheCertificateThumbprintHere} -DeleteKey
```

Know your certificate expiry date

The self-signed certificate you created following the steps above has a limited lifetime before it expires. In the **App registrations** section of the Azure portal, the **Certificates & secrets** screen displays the expiration date of the certificate. If you're using Azure Automation, the **Certificates** screen on the Automation account displays the expiration date of the certificate. Follow the previous steps to create a new self-signed certificate.

Next steps

[Manage certificates for federated single sign-on in Azure Active Directory](#)

Supported account types

4/12/2022 • 2 minutes to read • [Edit Online](#)

This article explains what account types (sometimes called *audiences*) are supported in the Microsoft identity platform applications.

Account types in the public cloud

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 called *single-tenant*.
- If you're an ISV, you can write an application that signs in users:
 - In any organization. Such an application is called a *multitenant* 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 accounts.
 - With only personal Microsoft accounts.
- If you're writing a business-to-consumer application, you can also sign in users with their social identities, by using Azure Active Directory B2C (Azure AD B2C).

Account type support in authentication flows

Some account types can't be used with certain authentication flows. For instance, in desktop, UWP, or daemon applications:

- Daemon applications can be used only with Azure AD organizations. It doesn't make sense to try to use daemon applications to manipulate Microsoft personal accounts. The admin consent will never be granted.
- You can use the integrated Windows authentication flow only with work or school accounts (in your organization or any organization). Integrated Windows authentication works with domain accounts, and it 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 Credentials grant](#) (username/password) can't be used with personal Microsoft accounts. 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.

Account types in national clouds

Apps can also sign in users in [national clouds](#). However, Microsoft personal accounts aren't supported in these clouds. That's why the supported account types are reduced, for these clouds, to your organization (single tenant) or any organizations (multitenant applications).

Next steps

- Learn more about [tenancy in Azure Active Directory](#).
- Learn more about [national clouds](#).

App sign-in flow with the Microsoft identity platform

4/12/2022 • 4 minutes to read • [Edit Online](#)

This topic discusses the basic sign-in flow for web, desktop, and mobile apps using Microsoft identity platform. See [Authentication flows and app scenarios](#) to learn about sign-in scenarios supported by Microsoft identity platform.

Web app sign-in flow

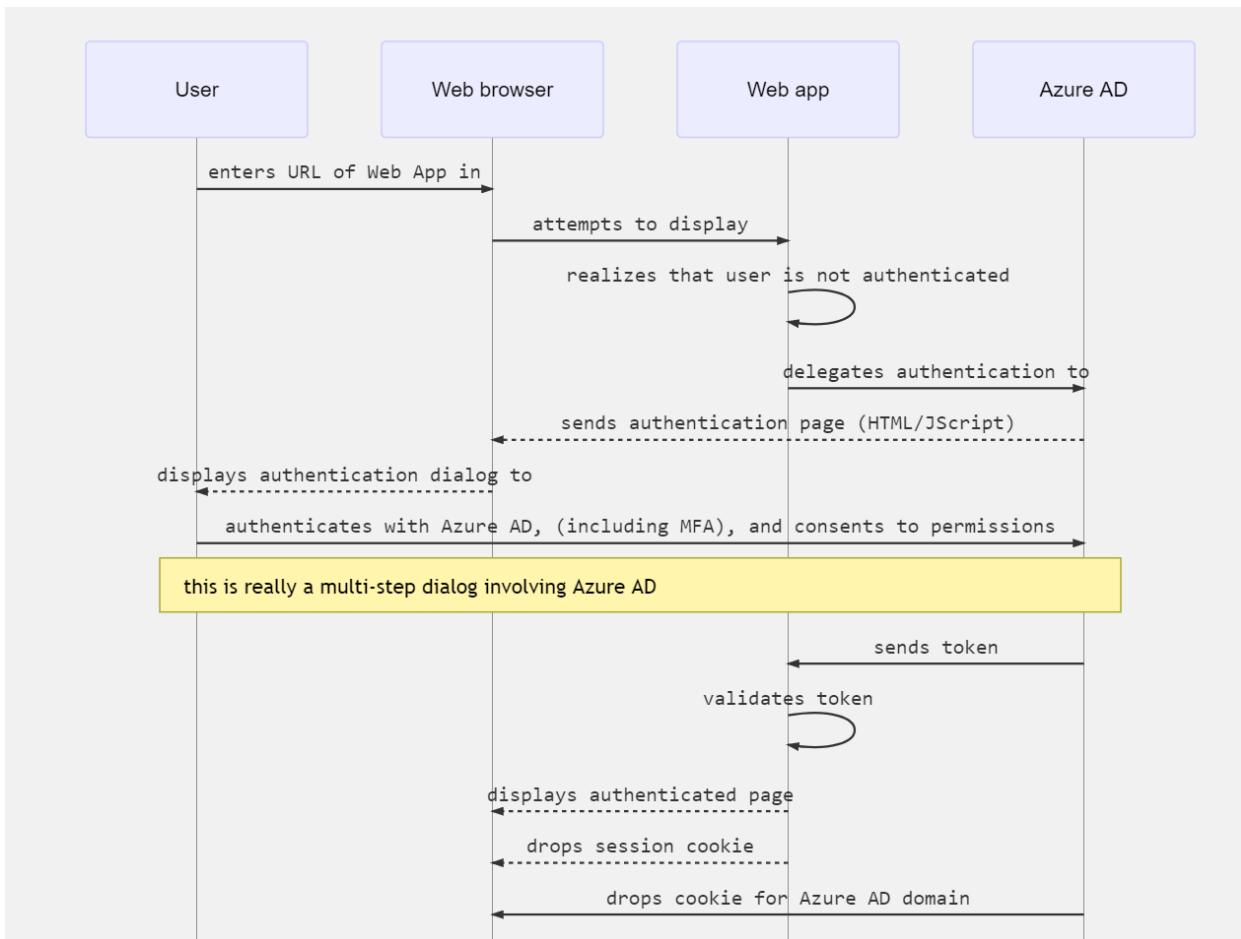
When a user navigates in the browser to a web app, the following happens:

- 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](#) (sometimes referred to as two-factor authentication or 2FA), 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 the Microsoft identity platform can deliver tokens representing the access that the user has consented to.

When the user has successfully authenticated:

- The Microsoft identity platform 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 Microsoft identity platform authorization endpoint, 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 the Microsoft identity platform and obtains a token

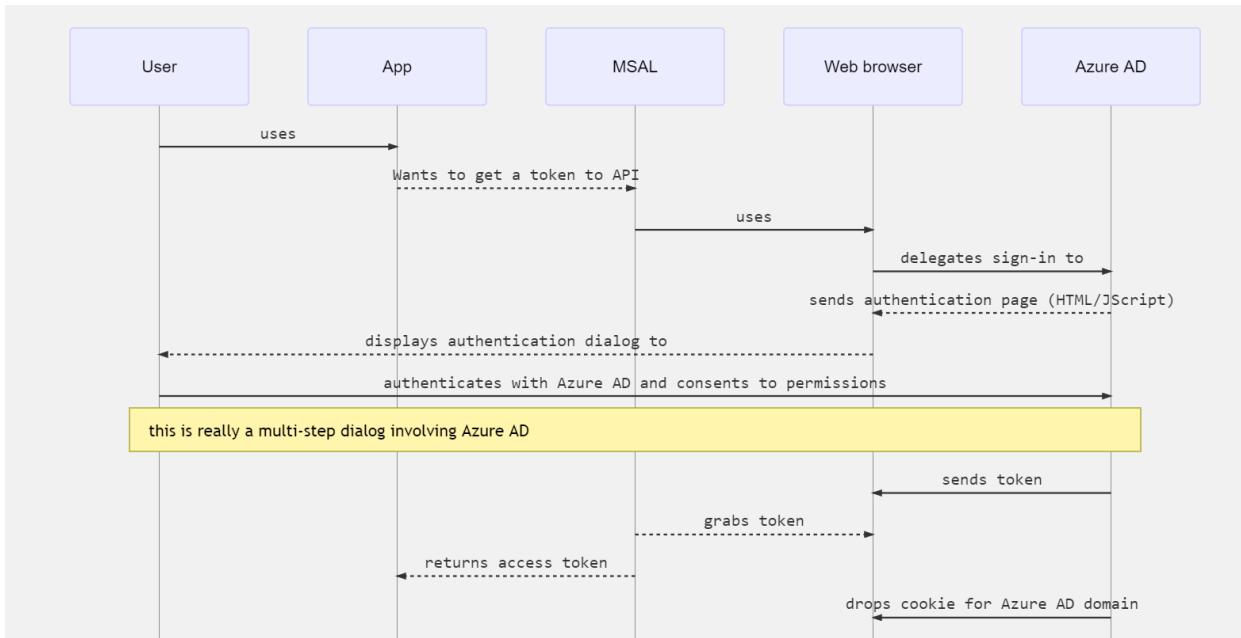
User authentication happens via the browser. The OpenID protocol uses standard HTTP protocol messages.

- The web app sends an HTTP 302 (redirect) to the browser to use Microsoft identity platform.
- When the user is authenticated, the Microsoft identity platform 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

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. As with web apps, authentication is delegated to Microsoft identity platform.

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. The exception is .NET Framework desktop applications where an embedded control is used to provide a more integrated user experience.

Next steps

For other topics covering authentication and authorization basics:

- See [Authentication vs. authorization](#) to learn about the basic concepts of authentication and authorization in Microsoft identity platform.
- See [Security tokens](#) to learn how access tokens, refresh tokens, and ID tokens are used in authentication and authorization.
- See [Application model](#) to learn about the process of registering your application so it can integrate with Microsoft identity platform.

To learn more about app sign-in flow:

- See [Authentication flows and app scenarios](#) to learn more about other scenarios for authenticating users supported by 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.

Restrict your Azure AD app to a set of users in an Azure AD tenant

4/12/2022 • 2 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 a [multi-tenant](#) application, all users in the Azure AD tenant where the application is provisioned can access the application once they successfully authenticate in their respective tenant.

Tenant administrators and developers often have requirements where an application must be restricted to a certain set of users. There are two ways to restrict an application to a certain set of users or security groups:

- Developers can use popular authorization patterns like [Azure role-based access control \(Azure RBAC\)](#).
- Tenant administrators and developers can use built-in feature of Azure AD.

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.

Update the app to require user assignment

To update an application to require user assignment, you must be owner of the application under Enterprise apps, or be assigned one of **Global administrator**, **Application administrator**, or **Cloud application administrator** directory roles.

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch the tenant in which you want to register an application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **Enterprise Applications > All applications**.
5. Select the application you want to configure to require assignment. Use the filters at the top of the window to search for a specific application.
6. On the application's **Overview** page, under **Manage**, select **Properties**.
7. Locate the setting **User assignment required?** and set it to **Yes**. When this option is set to **Yes**, users and services attempting to access the application or services must first be assigned for this application, or they won't be able to sign-in or obtain an access token.
8. Select **Save**.

When an application requires assignment, user consent for that application isn't allowed. This is true even if users consent for that app would have otherwise been allowed. Be sure to [grant tenant-wide admin consent](#) to apps that require assignment.

Assign the app to users and groups

Once you've configured your app to enable user assignment, you can go ahead and assign the app to users and groups.

1. Under **Manage**, select the **Users and groups > Add user/group**.

2. Select the **Users** selector.

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.

3. Once you're done selecting the users and groups, select **Select**.

4. (Optional) If you have defined app roles in your application, you can use the **Select role** option to assign the app role to the selected users and groups.

5. Select **Assign** to complete the assignments of the app to the users and groups.

6. Confirm that the users and groups you added are showing up in the updated **Users and groups** list.

More information

For more information about roles and security groups, see:

- [How to: Add app roles in your application](#)
- [Using Security Groups and Application Roles in your apps \(Video\)](#)
- [Azure Active Directory app manifest](#)

Add app roles to your application and receive them in the token

4/12/2022 • 8 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 less effort.

Another approach is to use Azure AD Groups and Group Claims as shown in the [active-directory-aspnetcore-webapp-openidconnect-v2](#) code sample on GitHub. Azure AD Groups and Application Roles are not mutually exclusive; they can be used in tandem to provide even finer-grained access control.

Declare roles for an application

You define app roles by using the [Azure portal](#) during the [app registration process](#). App roles are defined on an application registration representing a service, app or API. When a user signs in to the application, Azure AD emits a `roles` claim for each role that the user or service principal has been granted individually to the user and the user's group memberships. This can be used to implement claim-based authorization. App roles can be assigned [to a user or a group of users](#). App roles can also be assigned to the service principal for another application, or [to the service principal for a managed identity](#).

IMPORTANT

Currently if you add a service principal to a group, and then assign an app role to that group, Azure AD does not add the `roles` claim to tokens it issues.

App roles are declared using the app roles by using [App roles UI](#) in the Azure portal:

The number of roles you add counts toward application manifest limits enforced by Azure Active Directory. For information about these limits, see the [Manifest limits](#) section of [Azure Active Directory app manifest reference](#).

App roles UI

To create an app role by using the Azure portal's user interface:

1. Sign in to the [Azure portal](#).
2. Select the **Directory + subscription** filter in top menu, and then choose the Azure Active Directory tenant that contains the app registration to which you want to add an app role.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations**, and then select the application you want to define app roles in.
5. Select **App roles**, and then select **Create app role**.

Home > saurabhmsft > MyApp

MyApp | App roles

Search (Ctrl+ /) Create app role Got feedback?

Overview Quickstart Integration assistant

Manage

- Branding
- Authentication
- Certificates & secrets
- Token configuration
- API permissions
- Expose an API
- App roles**
- Owners
- Roles and administrators | Preview
- Manifest

Support + Troubleshooting

Troubleshooting New support request

6. In the **Create app role** pane, enter the settings for the role. The table following the image describes each setting and their parameters.

Create app role

Display name * ⓘ
Survey Writer

Allowed member types * ⓘ
 Users/Groups
 Applications
 Both (Users/Groups + Applications)

Value * ⓘ
Survey.Create

Description * ⓘ
Writers can create surveys.

Do you want to enable this app role? ⓘ

Apply **Cancel**

FIELD	DESCRIPTION	EXAMPLE
Display name	Display name for the app role that appears in the admin consent and app assignment experiences. This value may contain spaces.	Survey Writer

FIELD	DESCRIPTION	EXAMPLE
Allowed member types	<p>Specifies whether this app role can be assigned to users, applications, or both.</p> <p>When available to <code>applications</code>, app roles appear as application permissions in an app registration's Manage section > API permissions > Add a permission > My APIs > Choose an API > Application permissions.</p>	<code>Users/Groups</code>
Value	Specifies the value of the roles claim that the application should expect in the token. The value should exactly match the string referenced in the application's code. The value cannot contain spaces.	<code>Survey.Create</code>
Description	A more detailed description of the app role displayed during admin app assignment and consent experiences.	<code>Writers can create surveys.</code>
Do you want to enable this app role?	Specifies whether the app role is enabled. To delete an app role, deselect this checkbox and apply the change before attempting the delete operation.	<i>Checked</i>

7. Select **Apply** to save your changes.

Assign users and groups to roles

Once you've added app roles in your application, you can assign users and groups to the roles. Assignment of users and groups to roles can be done through the portal's UI, or programmatically using [Microsoft Graph](#). When the users assigned to the various app roles sign in to the application, their tokens will have their assigned roles in the `roles` claim.

To assign users and groups to roles by using the Azure portal:

1. Sign in to the [Azure portal](#).
2. In **Azure Active Directory**, select **Enterprise applications** in the left-hand navigation menu.
3. Select **All applications** to view a list of all your applications. If your application doesn't appear in the list, use the filters at the top of the **All applications** list to restrict the list, or scroll down the list to locate your application.
4. Select the application in which you want to assign users or security group to roles.
5. Under **Manage**, select **Users and groups**.
6. Select **Add user** to open the **Add Assignment** pane.
7. Select the **Users and groups** selector from the **Add Assignment** pane. A list of users and security groups is displayed. You can search for a certain user or group as well as select multiple users and groups that appear in the list.
8. Once you've selected users and groups, select the **Select** button to proceed.

9. Select **Select a role** in the **Add assignment** pane. All the roles that you've defined for the application are displayed.
10. Choose a role and select the **Select** button.
11. Select the **Assign** button to finish the assignment of users and groups to the app.

Confirm that the users and groups you added appear in the **Users and groups** list.

Assign app roles to applications

Once you've added app roles in your application, you can assign an app role to a client app by using the Azure portal or programmatically by using [Microsoft Graph](#).

When you assign app roles to an application, you create *application permissions*. Application permissions are typically used by daemon apps or back-end services that need to authenticate and make authorized API calls as themselves, without the interaction of a user.

To assign app roles to an application by using the Azure portal:

1. Sign in to the [Azure portal](#).
2. In **Azure Active Directory**, select **App registrations** in the left-hand navigation menu.
3. Select **All applications** to view a list of all your applications. If your application doesn't appear in the list, use the filters at the top of the **All applications** list to restrict the list, or scroll down the list to locate your application.
4. Select the application to which you want to assign an app role.
5. Select **API permissions > Add a permission**.
6. Select the **My APIs** tab, and then select the app for which you defined app roles.
7. Select **Application permissions**.
8. Select the role(s) you want to assign.
9. Select the **Add permissions** button complete addition of the role(s).

The newly added roles should appear in your app registration's **API permissions** pane.

Grant admin consent

Because these are *application permissions*, not delegated permissions, an admin must grant consent to use the app roles assigned to the application.

1. In the app registration's **API permissions** pane, select **Grant admin consent for <tenant name>**.
2. Select **Yes** when prompted to grant consent for the requested permissions.

The **Status** column should reflect that consent has been **Granted for <tenant name>**.

Usage scenario of app roles

If you're implementing app role business logic that signs in the users in your application scenario, first define the app roles in **App registration**. Then, an admin assigns them to users and groups in the **Enterprise applications** pane. These assigned app roles are included with any token that's issued for your application, either access tokens when your app is the API being called by an app or ID tokens when your app is signing in a user.

If you're implementing app role business logic in an app-calling-API scenario, you have two app registrations. One app registration is for the app, and a second app registration is for the API. In this case, define the app roles and assign them to the user or group in the app registration of the API. When the user authenticates with the app and requests an access token to call the API, a roles claim is included in the access token. Your next step is to add code to your web API to check for those roles when the API is called.

To learn how to add authorization to your web API, see [Protected web API: Verify scopes and app roles](#).

App roles vs. groups

Though you can use app roles or groups for authorization, key differences between them can influence which you decide to use for your scenario.

APP ROLES	GROUPS
They are specific to an application and are defined in the app registration. They move with the application.	They are not specific to an app, but to an Azure AD tenant.
App roles are removed when their app registration is removed.	Groups remain intact even if the app is removed.
Provided in the <code>roles</code> claim.	Provided in <code>groups</code> claim.

Developers can use app roles to control whether a user can sign in to an app or an app can obtain an access token for a web API. To extend this security control to groups, developers and admins can also assign security groups to app roles.

App roles are preferred by developers when they want to describe and control the parameters of authorization in their app themselves. For example, an app using groups for authorization will break in the next tenant as both the group ID and name could be different. An app using app roles remains safe. In fact, assigning groups to app roles is popular with SaaS apps for the very same reasons as it allows the SaaS app to be provisioned in multiple tenants.

Next steps

Learn more about app roles with the following resources.

- Code samples on GitHub
 - [Add authorization using app roles & roles claims to an ASP.NET Core web app](#)
 - [Add authorization using groups and group claims to an ASP.NET Core web app](#)
 - [Angular single-page application \(SPA\) calling a .NET Core web API and using app roles and security groups](#)
 - [React single-page application \(SPA\) calling a Node.js web API and using app roles and security groups](#)
- Reference documentation
 - [Azure AD app manifest](#)
 - [Azure AD access tokens](#)
 - [Azure AD ID tokens](#)
 - [Provide optional claims to your app](#)
- Video: [Implement authorization in your applications with Microsoft identity platform](#) (1:01:15)

Implement role-based access control in apps

4/12/2022 • 4 minutes to read • [Edit Online](#)

Role-based access control (RBAC) allows users or groups to have specific permissions regarding which resources they have access to, what they can do with those resources, and who manages which resources.

Typically, when we talk about implementing RBAC to protect a resource, we're looking to protect either a web application, a single-page application (SPA), or an API. This could be either for the entire application or API, or specific areas, features, or API methods.

This article explains how to implement application-specific role-based access control. For more information about the basics of authorization, see [Authorization basics](#).

Implementing RBAC using the Microsoft identity platform

As discussed in [Role-based access control for application developers](#), there are three ways to implement RBAC using the Microsoft identity platform:

- **App Roles** – using the [App Roles feature in an application registration](#) in conjunction with logic within your application to interpret incoming App Role assignments.
- **Groups** – using an incoming identity's group assignments in conjunction with logic within your application to interpret the group assignments.
- **Custom Data Store** – retrieve and interpret role assignments using logic within your application.

The preferred approach is to use *App Roles* as it is the easiest to implement. This approach is supported directly by the SDKs that are used in building apps utilizing the Microsoft identity platform. For more information on how to choose an approach, see [Choosing an approach](#).

The rest of this article will show you how to define app roles and implement RBAC within your application using the app roles.

Defining roles for your application

The first step for implementing RBAC for your application is to define the roles your application needs and assign users or groups to those roles. This process is outlined in [How to: Add app roles to your application and receive them in the token](#). Once you have defined your roles and assigned users or groups, you can access the role assignments in the tokens coming into your application and act on them accordingly.

Implementing RBAC in ASP.NET Core

ASP.NET Core supports adding RBAC to an ASP.NET Core web application or web API. This allows for easy implementation of RBAC using [role checks](#) with the ASP.NET Core *Authorize* attribute. It is also possible to use ASP.NET Core's support for [policy-based role checks](#).

ASP.NET Core MVC web application

Implementing RBAC in an ASP.NET Core MVC web application is straightforward. It mainly involves using the *Authorize* attribute to specify which roles should be allowed to access specific controllers or actions in the controllers. Follow these steps to implement RBAC in your ASP.NET Core MVC application:

1. Create an app registration with app roles and assignments as outlined in *Defining roles for your application* above.

2. Do one of the following steps:

- Create a new ASP.NET Core MVC web app project using the **dotnet cli**. Specify the **--auth** flag with either *SingleOrg* for single tenant authentication or *MultiOrg* for multi-tenant authentication, the **--client-id** flag with the client id from your app registration, and the **--tenant-id** flag with your tenant id from your Azure AD tenant:

```
dotnet new mvc --auth SingleOrg --client-id <YOUR-APPLICATION-CLIENT-ID> --tenant-id <YOUR-TENANT-ID>
```

- Add the `Microsoft.Identity.Web` and `Microsoft.Identity.Web.UI` libraries to an existing ASP.NET Core MVC project:

```
dotnet add package Microsoft.Identity.Web  
dotnet add package Microsoft.Identity.Web.UI
```

And then follow the instructions specified in [Quickstart: Add sign-in with Microsoft to an ASP.NET Core web app](#) to add authentication to your application.

3. Add role checks on your controller actions as outlined in [Adding role checks](#).

4. Test the application by trying to access one of the protected MVC routes.

ASP.NET Core web API

Implementing RBAC in an ASP.NET Core web API mainly involves utilizing the `Authorize` attribute to specify which roles should be allowed to access specific controllers or actions in the controllers. Follow these steps to implement RBAC in your ASP.NET Core web API:

1. Create an app registration with app roles and assignments as outlined in [Defining roles for your application](#) above.

2. Do one of the following steps:

- Create a new ASP.NET Core MVC web API project using the **dotnet cli**. Specify the **--auth** flag with either *SingleOrg* for single tenant authentication or *MultiOrg* for multi-tenant authentication, the **--client-id** flag with the client id from your app registration, and the **--tenant-id** flag with your tenant id from your Azure AD tenant:

```
dotnet new webapi --auth SingleOrg --client-id <YOUR-APPLICATION-CLIENT-ID> --tenant-id <YOUR-TENANT-ID>
```

- Add the `Microsoft.Identity.Web` and `Swashbuckle.AspNetCore` libraries to an existing ASP.NET Core web API project:

```
dotnet add package Microsoft.Identity.Web  
dotnet add package Swashbuckle.AspNetCore
```

And then follow the instructions specified in [Quickstart: Add sign-in with Microsoft to an ASP.NET Core web app](#) to add authentication to your application.

3. Add role checks on your controller actions as outlined in [Adding role checks](#).
4. Call the API from a client app. See [Angular single-page application calling .NET Core web API and using App Roles to implement Role-Based Access Control](#) for an end to end sample.

Implementing RBAC in other platforms

Angular SPA using MsalGuard

Implementing RBAC in an Angular SPA involves the use of [msal-angular](#) to authorize access to the Angular routes contained within the application. This is shown in the [Enable your Angular single-page application to sign-in users and call APIs with the Microsoft identity platform](#) sample.

NOTE

Client-side RBAC implementations should be paired with server-side RBAC to prevent unauthorized applications from accessing sensitive resources.

Node.js with Express application

Implementing RBAC in a Node.js with express application involves the use of MSAL to authorize access to the Express routes contained within the application. This is shown in the [Enable your Node.js web app to sign-in users and call APIs with the Microsoft identity platform](#) sample.

Next steps

- Read more on [permissions and consent in the Microsoft identity platform](#).
- Read more on [role-based access control for application developers](#).

Support passwordless authentication with FIDO2 keys in apps you develop

4/12/2022 • 2 minutes to read • [Edit Online](#)

These configurations and best practices will help you avoid common scenarios that block [FIDO2 passwordless authentication](#) from being available to users of your applications.

General best practices

Domain hints

Don't use a domain hint to bypass [home-realm discovery](#). This feature is meant to make sign-ins more streamlined, but the federated identity provider may not support passwordless authentication.

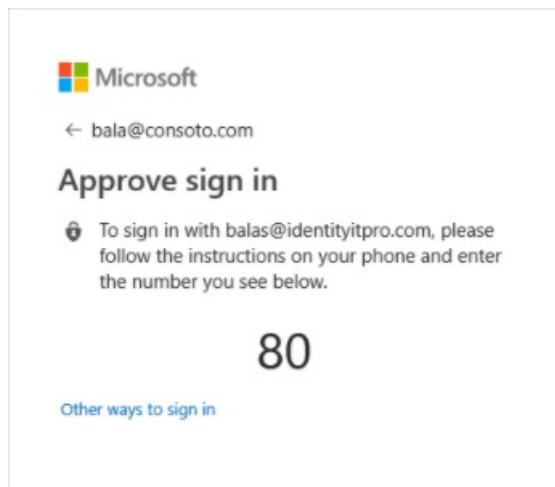
Requiring specific credentials

If you are using SAML, do not specify that a password is required [using the RequestedAuthnContext element](#).

The RequestedAuthnContext element is optional, so to resolve this you can remove it from your SAML authentication requests. This is a general best practice, as using this element can also prevent other authentication options like multi-factor authentication from working correctly.

Using the most recently used authentication method

The sign-in method that was most recently used by a user will be presented to them first. This may cause confusion when users believe they must use the first option presented. However, they can choose another option by selecting "Other ways to sign in" as shown below.



Platform-specific best practices

Desktop

The recommended options for implementing authentication are, in order:

- .NET desktop applications that are using the Microsoft Authentication Library (MSAL) should use the Windows Authentication Manager (WAM). This integration and its benefits are [documented on GitHub](#).
- Use [WebView2](#) to support FIDO2 in an embedded browser.
- Use the system browser. The MSAL libraries for desktop platforms use this method by default. You can consult our page on FIDO2 browser compatibility to ensure the browser you use supports FIDO2 authentication.

Mobile

As of February 2021, FIDO2 is not currently supported for native iOS or Android apps, but it is in development.

To prepare applications for its availability, and as a general best practice, iOS and Android applications should use MSAL with its default configuration of using the system web browser.

If you are not using MSAL, you should still use the system web browser for authentication. Features such as single sign-on and conditional access rely on a shared web surface provided by the system web browser. This means using [Chrome Custom Tabs](#) (Android) or [Authenticating a User Through a Web Service | Apple Developer Documentation](#) (iOS).

Web and single-page apps

The availability of FIDO2 passwordless authentication for applications that run in a web browser will depend on the combination of browser and platform. You can consult our [FIDO2 compatibility matrix](#) to check if the combination your users will encounter is supported.

Next steps

[Passwordless authentication options for Azure Active Directory](#)

Provide optional claims to your app

4/12/2022 • 20 minutes to read • [Edit Online](#)

Application developers can use optional claims in their Azure AD applications 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 the Microsoft identity platform 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 [Microsoft identity platform](#) 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](#), the claims apply to access tokens requested *for* the application (a web API), not claims requested *by* the application. No matter how the client accesses your API, the right data is present in the access token that is used 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
------	-------------	------------	-----------	-------

NAME	DESCRIPTION	TOKEN TYPE	USER TYPE	NOTES
<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>auth_time</code>	Time when the user last authenticated. See OpenID Connect spec.	JWT		
<code>ctry</code>	User's country/region	JWT		Azure AD returns the <code>ctry</code> optional claim if it's present and the value of the field is a standard two-letter country/region code, such as FR, JP, SZ, and so on.
<code>email</code>	The reported email address for this user	JWT, SAML	MSA, Azure AD	This value is included by default if the user is a guest in the tenant. For managed users (the users inside the tenant), it must be requested through this optional claim or, on v2.0 only, with the OpenID scope. This value is not guaranteed to be correct, and is mutable over time - never use it for authorization or to save data for a user. For more information, see Validate the user has permission to access this data . If you require an addressable email address in your app, request this data from the user directly, using this claim as a suggestion or pre-fill in your UX.
<code>fwd</code>	IP address.	JWT		Adds the original IPv4 address of the requesting client (when inside a VNET)

NAME	DESCRIPTION	TOKEN TYPE	USER TYPE	NOTES
<code>groups</code>	Optional formatting for group claims	JWT, SAML		For details see Group claims below. For more information about group claims, see How to configure group claims . Used with the <code>GroupMembershipClaims</code> setting in the application manifest , which must be set as well.
<code>idtyp</code>	Token type	JWT access tokens	Special: only in app-only access tokens	Value is <code>app</code> when the token is an app-only token. This is the most accurate way for an API to determine if a token is an app token or an app+user token.

Name	Description	Token Type	User Type	Notes
<code>login_hint</code>	Login hint	JWT	MSA, Azure AD	An opaque, reliable login hint claim. This claim is the best value to use for the <code>login_hint</code> OAuth parameter in all flows to get SSO. It can be passed between applications to help them silently SSO as well - application A can sign in a user, read the <code>login_hint</code> claim, and then send the claim and the current tenant context to application B in the query string or fragment when the user clicks on a link that takes them to application B. To avoid race conditions and reliability issues, the <code>login_hint</code> claim <i>doesn't</i> include the current tenant for the user, and defaults to the user's home tenant when used. If you are operating in a guest scenario, where the user is from another tenant, then you must still provide a tenant identifier in the sign in request, and pass the same to apps you partner with. This claim is intended for use with your SDK's existing <code>login_hint</code> functionality, however that it exposed.
<code>sid</code>	Session ID, used for per-session user sign-out.	JWT	Personal and Azure AD accounts.	
<code>tenant_ctry</code>	Resource tenant's country/region	JWT		Same as <code>ctry</code> except set at a tenant level by an admin. Must also be a standard two-letter value.

NAME	DESCRIPTION	TOKEN TYPE	USER TYPE	NOTES
<code>tenant_region_scope</code>	Region of the resource tenant	JWT		
<code>upn</code>	UserPrincipalName	JWT, SAML		<p>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 for authorization or to uniquely identify user information (for example, as a database key). Instead, use the user object ID (<code>oid</code>) as a database key. For more information, see Validate the user has permission to access this data.</p> <p>Users signing in with an alternate login ID should not be shown their User Principal Name (UPN). Instead, use the following ID token claims for displaying sign-in state to the user:</p> <ul style="list-style-type: none"> <code>preferred_username</code> or <code>unique_name</code> for v1 tokens and <code>preferred_username</code> for v2 tokens. 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. You should use the <code>login_hint</code> claim for <code>login_hint</code> use - human-readable identifiers like UPN are unreliable.
<code>verified_primary_email</code>	Sourced from the user's PrimaryAuthoritative Email	JWT		

NAME	DESCRIPTION	TOKEN TYPE	USER TYPE	NOTES
<code>verified_secondary_email</code>	Sourced from the user's SecondaryAuthoritativeEmail	JWT		
<code>vnet</code>	VNET specifier information.	JWT		
<code>xms_pdl</code>	Preferred data location	JWT		<p>For Multi-Geo tenants, the preferred data location is the three-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>
<code>xms_p1</code>	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").
<code>xms_tpl</code>	Tenant preferred language	JWT		The resource tenant's preferred language, if set. Formatted LL ("en").
<code>ztdid</code>	Zero-touch Deployment ID	JWT		The device identity used for Windows AutoPilot

v2.0-specific optional claims set

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		

JWT CLAIM	NAME	DESCRIPTION	NOTES
<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.	
<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>family_name</code>	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. Requires the <code>profile</code> scope.
<code>given_name</code>	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. Requires the <code>profile</code> scope.
<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 for authorization or to uniquely identify user information (for example, as a database key). For more information, see Validate the user has permission to access this data . Instead, use the user object ID (<code>oid</code>) as a database key. Users signing in with an alternate login ID should not be shown their User Principal Name (UPN). Instead, use the following <code>preferred_username</code> claim for displaying sign-in state to the user.	See additional properties below for configuration of the claim. Requires the <code>profile</code> scope.

v1.0-specific optional claims set

Some of the improvements of the v2 token format are available to apps that use the v1 token format, as they help improve security and reliability. These will not take effect for ID tokens requested from the v2 endpoint, nor access tokens for APIs that use the v2 token format. These only apply to JWTs, not SAML tokens.

Table 4: v1.0-only optional claims

JWT CLAIM	NAME	DESCRIPTION	NOTES
<code>aud</code>	Audience	Always present in JWTs, but in v1 access tokens it can be emitted in a variety of ways - any appID URI, with or without a trailing slash, as well as the client ID of the resource. This randomization can be hard to code against when performing token validation. Use the additional properties for this claim to ensure it's always set to the resource's client ID in v1 access tokens.	v1 JWT access tokens only
<code>preferred_username</code>	Preferred username	Provides the preferred username claim within v1 tokens. This makes it easier for apps to provide username hints and show human readable display names, regardless of their token type. It's recommended that you use this optional claim instead of using e.g. <code>upn</code> or <code>unique_name</code> .	v1 ID tokens and access tokens

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.com</code>
	<code>include_externally_authenticated_upn_{hash}</code>	Same as above, except that the hash marks (#) are replaced with underscores (_), for example <code>foo_hometenant.com_EXT_@resourcetenant.com</code>

PROPERTY NAME	ADDITIONAL PROPERTY NAME	DESCRIPTION
<code>aud</code>		In v1 access tokens, this is used to change the format of the <code>aud</code> claim. This has no effect in v2 tokens or either version's ID tokens, where the <code>aud</code> claim is always the client ID. Use this configuration to ensure that your API can more easily perform audience validation. Like all optional claims that affect the access token, the resource in the request must set this optional claim, since resources own the access token.
	<code>use_guid</code>	<p>Emits the client ID of the resource (API) in GUID format as the <code>aud</code> claim always instead of it being runtime dependent. For example, if a resource sets this flag, and its client ID is</p> <div style="background-color: #f0f0f0; padding: 5px;"><code>bb0a297b-6a42-4a55-ac40-09a501456577</code></div> <p>, any app that requests an access token for that resource will receive an access token with <code>aud</code> :</p> <div style="background-color: #f0f0f0; padding: 5px;"><code>bb0a297b-6a42-4a55-ac40-09a501456577</code></div> <p>.</p> <p>Without this claim set, an API could get tokens with an <code>aud</code> claim of</p> <div style="background-color: #f0f0f0; padding: 5px;"><code>api://MyApi.com ,</code></div> <div style="background-color: #f0f0f0; padding: 5px;"><code>api://MyApi.com/ ,</code></div> <div style="background-color: #f0f0f0; padding: 5px;"><code>api://myapi.com/AdditionalRegisteredField</code></div> <p>or any other value set as an app ID URI for that API, as well as the client ID of the resource.</p>

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 a `upn` claim with the additional home tenant and resource tenant information. The `upn` claim is only changed in the token if the user is a guest in the tenant (that uses a different IDP for authentication).

Configuring optional claims

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 the Microsoft Graph API. Thus, the access token is created using the Microsoft Graph API manifest, not the client's manifest. Changing the manifest for your application will never cause tokens for the Microsoft Graph API to look different. In order to validate that your `accessToken` changes are in effect, request a token for your application, not another app.

You can configure optional claims for your application through the UI or application manifest.

1. Go to the [Azure portal](#).
2. Search for and select **Azure Active Directory**.
3. Under **Manage**, select **App registrations**.
4. Select the application you want to configure optional claims for in the list.

Configuring optional claims through the UI:

The screenshot shows the Azure portal interface for managing an app registration. On the left, the navigation menu under 'Manage' has 'Token configuration' selected. The main area displays the 'Litware HR App - Token configuration' blade, which includes a search bar, an 'Overview' section, and a 'Manage' sidebar with various options like Branding, Authentication, Certificates & secrets, API permissions, Expose an API, Owners, Roles and administrators, Manifest, and Support + Troubleshooting. Below these is a 'Token configuration' section with a 'Search (Ctrl+ /)' field and a 'Optional claims' table showing 'No results.' A 'Quickstart' link is also present. To the right, a modal window titled 'Add optional claim' is open. It has a 'Token type' section with three radio buttons: 'ID' (selected), 'Access', and 'SAML'. Below this is a table with columns 'Claim ↑↓' and 'Description'. The table lists several optional claims with checkboxes: 'acct', 'auth_time', 'ctry', 'email', 'enfpolids', 'family_name', 'fwd', 'given_name', and 'home_oid'. At the bottom of the modal are 'Add' and 'Cancel' buttons.

1. Under **Manage**, select **Token configuration**.
 - The UI option **Token configuration** blade is not available for apps registered in an Azure AD B2C tenant which can be configured by modifying the application manifest. For more information see [Add claims and customize user input using custom policies in Azure Active Directory B2C](#)
2. Select **Add optional claim**.
3. Select the token type you want to configure.
4. Select the optional claims to add.
5. Select **Add**.

Configuring optional claims through the application manifest:

```

27 "oauth2RequirePostResponse": false,
28 "optionalClaims": {
29   "idToken": [
30     {
31       "name": "auth_time",
32       "source": null,
33       "essential": false,
34       "additionalProperties": []
35     }
36   ],
37   "accessToken": [
38     {
39       "name": "ipaddr",
40       "source": null,
41       "essential": false,
42       "additionalProperties": []
43     }
44   ],
45   "saml2Token": [
46     {
47       "name": "upn",
48       "source": null,
49       "essential": false,
50       "additionalProperties": []
51     },
52     {
53       "name": "extension_ab603c56068041afb2f6832e2a17e237_skypeId",
54       "source": "user",
55       "essential": false,
56       "additionalProperties": []
57     }
58   ]
59 }

```

1. Under **Manage**, select **Manifest**. A web-based manifest editor opens, allowing you to edit the manifest.

Optionally, you can select **Download** and edit the manifest locally, and then use **Upload** to reapply it to your application. For more information on the application manifest, see the [Understanding the Azure AD application manifest article](#).

The following application manifest entry adds the `auth_time`, `ipaddr`, and `upn` optional claims to ID, access, and SAML tokens.

```

"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
    }
  ]
}

```

2. When finished, select **Save**. Now the specified optional claims will be included in the tokens for your application.

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) that 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 extensions. For more info, see [the Microsoft Graph extensionProperty documentation](#).

Schema and open extensions are not supported by optional claims, only the AAD-Graph style directory 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. See the bottom of this page for an example.

Directory schema extensions are an Azure AD-only feature. If your application manifest requests a custom

extension and an MSA user logs in to your app, these extensions will not be returned.

Directory extension formatting

When configuring directory extension optional claims using the application manifest, use the full name of the extension (in the format: `extension_<appid>_<attributename>`). 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 groups optional claims

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. You can configure groups optional claims for your application through the UI or application manifest.

IMPORTANT

Azure AD limits the number of groups emitted in a token to 150 for SAML assertions and 200 for JWT, including nested groups. For more details on group limits and important caveats for group claims from on-premises attributes, see [Configure group claims for applications with Azure AD](#).

Configuring groups optional claims through the UI:

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. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations**.
5. Select the application you want to configure optional claims for in the list.
6. Under **Manage**, select **Token configuration**.
7. Select **Add groups claim**.
8. Select the group types to return (**Security groups**, or **Directory roles**, **All groups**, and/or **Groups assigned to the application**):
 - The **Groups assigned to the application** option includes only groups assigned to the application. The **Groups assigned to the application** option is recommended for large organizations due to the group number limit in token. To change the groups assigned to the application, select the application from the **Enterprise applications** list. Select **Users and groups** and then **Add user/group**. Select the group(s) you want to add to the application from **Users and groups**.
 - The **All Groups** option includes **SecurityGroup**, **DirectoryRole**, and **DistributionList**, but not **Groups assigned to the application**.
9. Optional: select the specific token type properties to modify the groups claim value to contain on premises group attributes or to change the claim type to a role.
10. Select **Save**.

Configuring groups optional claims through the application manifest:

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. Search for and select **Azure Active Directory**.

4. Select the application you want to configure optional claims for in the list.

5. Under **Manage**, select **Manifest**.

6. Add the following entry using the manifest editor:

The valid values are:

- "All" (this option includes SecurityGroup, DirectoryRole, and DistributionList)
- "SecurityGroup"
- "DirectoryRole"
- "ApplicationGroup" (this option includes only groups that are assigned to the application)

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:

7. Set group name configuration optional claims.

If you want 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 access token
- Saml2Token for SAML tokens.

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 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.

If "emit_as_roles" is used, any application roles configured that the user is assigned will not appear in the role claim.

Examples:

1. Emit groups as group names in OAuth access tokens in dnsDomainName\sAMAccountName format

UI configuration:

The screenshot shows the Azure portal's 'Litware HR App - Token configuration' page. In the 'Optional claims' section, there are three claims listed: 'upn', 'extn.skypeID', and 'groups'. Below the list is a button labeled '+ Add groups claim'. To the right, a modal window titled 'Groups claims' is open. It contains a note about applying group claims to all token types. Under 'Select group types to return', 'All' is selected. Under 'Access token properties', 'DNSDomain\sAMAccountName' is selected as the format, indicated by a red box around the radio button. Other options like 'Group ID', 'sAMAccountName', and 'NetBIOSDomain\sAMAccountName' are also listed but not selected.

Application manifest entry:

```
"optionalClaims": {
    "accessToken": [
        {
            "name": "groups",
            "additionalProperties": [
                "dns_domain_and_sam_account_name"
            ]
        }
    ]
}
```

2. Emit group names to be returned in netbiosDomain\sAMAccountName format as the roles claim in SAML and OIDC ID Tokens

UI configuration:

Application manifest entry:

```

"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 use the **Token configuration** UI (see example below)
- You can use the **Manifest** (see example below). 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 [Microsoft Graph API](#) to update your application. The [OptionalClaims](#) type in the Microsoft Graph API reference guide can help you with configuring the optional claims.

Example:

In the example below, you will use the **Token configuration** UI and **Manifest** to add optional claims to the access, ID, and SAML tokens intended for your application. Different optional claims will be 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 `skypeld` 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`.

UI configuration:

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. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations**.
5. Find the application you want to configure optional claims for in the list and select it.
6. Under **Manage**, select **Token configuration**.
7. Select **Add optional claim**, select the ID token type, select `upn` from the list of claims, and then select **Add**.
8. Select **Add optional claim**, select the Access token type, select `auth_time` from the list of claims, then select **Add**.
9. From the Token Configuration overview screen, select the pencil icon next to `upn`, select the **Externally authenticated** toggle, and then select **Save**.
10. Select **Add optional claim**, select the SAML token type, select `extn.skypeID` from the list of claims (only applicable if you've created an Azure AD user object called `skypeld`), and then select **Add**.

Claim ↑	Description	Token type ↑↓
auth_time	Time when the user last authenticated: See OpenID Connect spec	Access
extension_ab603c56068041afb2f6832e2a17e237_skypeId	An identifier for the user that can be used with the username_hint parameter, not a dura... ID	SAML
upn	An identifier for the user that can be used with the username_hint parameter, not a dura... ID	ID

Manifest configuration:

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. Search for and select **Azure Active Directory**.
4. Find the application you want to configure optional claims for in the list and select it.
5. Under **Manage**, select **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 automatically 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  
        }  
    ]  
}
```

7. When you're finished updating the manifest, select **Save** to save the manifest.

Next steps

Learn more about the standard claims provided by Azure AD.

- [ID tokens](#)
- [Access tokens](#)

Claims mapping policy type

4/12/2022 • 11 minutes to read • [Edit Online](#)

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.
Restricted claim set	Can't be modified using policy. The data source cannot be changed, and no transformation is applied when generating these claims.

This section lists:

- [Table 1: JSON Web Token \(JWT\) restricted claim set](#)
- [Table 2: SAML restricted claim set](#)

Table 1: JSON Web Token (JWT) restricted claim set

NOTE

Any claim starting with "xms_" is restricted.

CLAIM TYPE (NAME)

.

_claim_names

_claim_sources

aai

access_token

account_type

CLAIM TYPE (NAME)

acct

acr

acrs

actor

ageGroup

aio

altsecid

amr

app_chain

app_displayname

app_res

appctx

appctxsender

appid

appidacr

at_hash

auth_time

azp

azpacr

c_hash

ca_enf

ca_policy_result

capolids_latebind

capolids

cc

CLAIM TYPE (NAME)

cnf

code

controls_auds

controls

credential_keys

ctry

deviceid

domain_dns_name

domain_netbios_name

e_exp

email

endpoint

enfpolids

expires_on

fido_auth_data

fwd_appidacr

fwd

graph

group_sids

groups

hasgroups

haswids

home_oid

home_puid

home_tid

CLAIM TYPE (NAME)

identityprovider

idp

idtyp

in_corp

instance

inviteTicket

ipaddr

isbrowserhostedapp

isViral

login_hint

mam_compliance_url

mam_enrollment_url

mam_terms_of_use_url

mdm_compliance_url

mdm_enrollment_url

mdm_terms_of_use_url

msproxy

nameid

nickname

nonce

oid

on_prem_id

onprem_sam_account_name

onprem_sid

openid2_id

CLAIM TYPE (NAME)

origin_header

platf

polids

pop_jwk

preferred_username

primary_sid

prov_data

puid

pwd_exp

pwd_url

rdp_bt

refresh_token_issued_on

refreshtoken

rh

roles

rt_type

scp

secaud

sid

sid

signin_state

source_anchor

src1

src2

sub

CLAIM TYPE (NAME)

target_deviceid

tbid

tbidv2

tenant_ctry

tenant_display_name

tenant_region_scope

tenant_region_sub_scope

thumbnail_photo

tid

tokenAutologonEnabled

trustedfordelegation

ttr

unique_name

upn

user_setting_sync_url

uti

ver

verified_primary_email

verified_secondary_email

vnet

wamcompat_client_info

wamcompat_id_token

wamcompat_scopes

wids

xcb2b_rclient

CLAIM TYPE (NAME)

xcb2b_rcloud

xcb2b_rtenant

ztdid

Table 2: SAML restricted claim set

The following table lists the SAML claims that are by default in the restricted claim set.

CLAIM TYPE (URI)`http://schemas.microsoft.com/2012/01/devicecontext/claims/ismanaged``http://schemas.microsoft.com/2014/02/devicecontext/claims/isknown``http://schemas.microsoft.com/2014/03/pss0``http://schemas.microsoft.com/2014/09/devicecontext/claims/iscompliant``http://schemas.microsoft.com/claims/authnmethodsreferences``http://schemas.microsoft.com/claims/groups.link``http://schemas.microsoft.com/identity/claims/accesstoken``http://schemas.microsoft.com/identity/claims/acct``http://schemas.microsoft.com/identity/claims/agegroup``http://schemas.microsoft.com/identity/claims/aio``http://schemas.microsoft.com/identity/claims/identityprovider``http://schemas.microsoft.com/identity/claims/objectidentifier``http://schemas.microsoft.com/identity/claims/openid2_id``http://schemas.microsoft.com/identity/claims/puid``http://schemas.microsoft.com/identity/claims/tenantid``http://schemas.microsoft.com/identity/claims/xms_et``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`

CLAIM TYPE (URI)

`http://schemas.microsoft.com/ws/2008/06/identity/claims/groups`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/role`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/wids`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsaccountname`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid`

`http://schemas.microsoft.com/ws/2008/06/identity/claims/primarygroupsid`

`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/sid`

`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/role`

These claims are restricted by default, but are not restricted if you [set the AcceptMappedClaims property to true](#) in your app manifest *or have a custom signing key*:

- `http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsaccountname`
- `http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid`
- `http://schemas.microsoft.com/ws/2008/06/identity/claims/primarygroupsid`
- `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/sid`
- `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/x500distinguishedname`

These claims are restricted by default, but are not restricted if you have a [custom signing key](#):

- `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn`
- `http://schemas.microsoft.com/ws/2008/06/identity/claims/role`

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.

NOTE

Claims in the core claim set are present in every token, regardless of what this property is set to.

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.

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**, **Source/ID pair**, or **Source/ExtensionID 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.

Source/ExtensionID pair: The Source and ExtensionID elements define the directory schema extension attribute where the data in the claim is sourced from. For more information, see [Using directory schema extension attributes in claims](#).

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.

WARNING

Currently, the only available multi-valued claim sources on a user object are multi-valued extension attributes which have been synced from AADConnect. Other properties, such as OtherMails and tags, are multi-valued but only one value is emitted when selected as a source.

Table 3: Valid ID values per source

SOURCE	ID	DESCRIPTION
User	surname	Family Name
User	givenname	Given Name
User	displayname	Display Name

SOURCE	ID	DESCRIPTION
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
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
User	othermail	Other Mail
User	country	Country/Region
User	city	City
User	state	State
User	jobtitle	Job Title
User	employeeid	Employee ID
User	facsimiletelephonenumber	Facsimile Telephone Number
User	assignedroles	list of App roles assigned to user
User	accountEnabled	Account Enabled
User	consentprovidedforminor	Consent Provided For Minor
User	createddatetime	Created Date/Time
User	creationtype	Creation Type
User	lastpasswordchangedatetime	Last Password Change Date/Time
User	mobilephone	Mobile Phone
User	officelocation	Office Location
User	onpremisesdomainname	On-Premises Domain Name
User	onpremisesimmutableid	On-Premises Immutable ID
User	onpremisessyncenabled	On-Premises Sync Enabled
User	preferreddatalocation	Preferred Data Location
User	proxyaddresses	Proxy Addresses
User	usertype	User Type

SOURCE	ID	DESCRIPTION
application, resource, audience	displayname	Display Name
application, resource, audience	objectid	ObjectID
application, resource, audience	tags	Service Principal Tag
Company	tenantcountry	Tenant's country/region

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.
- **onPremisesUserPrincipalName attribute:** When using an Alternate ID, the on-premises attribute userPrincipalName is synchronized with the Azure AD attribute onPremisesUserPrincipalName. This attribute is only available when Alternate ID is configured but is also available through MS Graph Beta: <https://graph.microsoft.com/beta/me/>.

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.

Group Filter (Preview)

String: GroupFilter

Data type: JSON blob

Summary: Use this property to apply a filter on the user's groups to be included in the group claim. This can be a useful means of reducing the token size.

MatchOn: The MatchOn property identifies the group attribute on which to apply the filter.

Set the **MatchOn** property to one of the following values:

- "displayname": The group display name.
- "samaccountname": The On-premises SAM Account Name

Type: The Type property selects the type of filter you wish to apply to the attribute selected by the **MatchOn** property.

Set the **Type** property to one of the following values:

- "prefix": Include groups where the **MatchOn** property starts with the provided **Value** property.
- "suffix": Include groups where the **MatchOn** property ends with the provided **Value** property.
- "contains": Include groups where the **MatchOn** property contains with the provided **Value** property.

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.sandbox"
ExtractMailPrefix	Email or UPN	extracted string	ExtensionAttributes 1-15 or any other Schema Extensions which are storing a UPN or email address value for the user e.g. johndoe@contoso.com. 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 three attributes: **ClaimTypeReferenceId**, **TransformationClaimType** and **TreatAsMultiValue** (Preview)

- **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.
- **TreatAsMultiValue** is a Boolean flag indicating if the transform should be applied to all values or just the first. By default, transformations will only be applied to the first element in a multi value claim, by setting this value to true it ensures it is applied to all. ProxyAddresses and groups are 2 examples for input claims that you would likely want to treat as a multi value.

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
User	extensionattribute12	Extension Attribute 12
User	extensionattribute13	Extension Attribute 13
User	extensionattribute14	Extension Attribute 14

SOURCE	ID	DESCRIPTION
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.

Next steps

- To learn how to customize the claims emitted in tokens for a specific application in their tenant using PowerShell, see [How to: Customize claims emitted in tokens for a specific app in a tenant](#)
- 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](#)
- To learn more about extension attributes, see [Using directory schema extension attributes in claims](#).

Using directory schema extension attributes in claims

4/12/2022 • 4 minutes to read • [Edit Online](#)

Directory schema extension attributes provide a way to store additional data in Azure Active Directory on user objects and other directory objects such as groups, tenant details, service principals. Only extension attributes on user objects can be used for emitting claims to applications. This article describes how to use directory schema extension attributes for sending user data to applications in token claims.

NOTE

Microsoft Graph provides two other extension mechanisms to customize Graph objects. These are known as Microsoft Graph open extensions and Microsoft Graph schema extensions. See the [Microsoft Graph documentation](#) for details. Data stored on Microsoft Graph objects using these capabilities are not available as sources for claims in tokens.

Directory schema extension attributes are always associated with an application in the tenant and are referenced by the application's *applicationId* in their name.

The identifier for a directory schema extension attribute is of the form *Extension_xxxxxxxxxx_Attributename*. Where *xxxxxxxxx* is the *applicationId* of the application the extension was defined for.

Registering and using directory schema extensions

Directory schema extension attributes can be registered and populated in one of two ways:

- By configuring AD Connect to create them and to sync data into them from on premises AD. See [Azure AD Connect Sync Directory Extensions](#).
- By using Microsoft Graph to register, set the values of, and read from [schema extensions](#). [PowerShell cmdlets](#) are also available.

Emitting claims with data from directory schema extension attributes created with AD Connect

Directory schema extension attributes created and synced using AD Connect are always associated with the application ID used by AD Connect. They can be used as a source for claims both by configuring them as claims in the **Enterprise Applications** configuration in the Portal UI for SAML applications registered using the Gallery or the non-Gallery application configuration experience under **Enterprise Applications**, and via a claims-mapping policy for applications registered via the Application registration experience. Once a directory extension attribute created via AD Connect is in the directory, it will show in the SAML SSO claims configuration UI.

Emitting claims with data from directory schema extension attributes created for an application using Graph or PowerShell

If a directory schema extension attribute is registered for an application using Microsoft Graph or PowerShell (via an applications initial setup or provisioning step for instance), the same application can be configured in Azure Active Directory to receive data in that attribute from a user object in a claim when the user signs in. The application can be configured to receive data in directory schema extensions that are registered on that same application using [optional claims](#). These can be set in the application manifest. This enables a multi-tenant application to register directory schema extension attributes for its own use. When the application is provisioned into a tenant the associated directory schema extensions become available to be set on users in that tenant, and to be consumed. Once it's configured in the tenant and consent granted, it can be used to store and

retrieve data via graph and to map to claims in tokens the Microsoft identity platform emits to applications.

Directory schema extension attributes can be registered and populated for any application.

If an application needs to send claims with data from an extension attribute registered on a different application, a [claims mapping policy](#) must be used to map the extension attribute to the claim. A common pattern for managing directory schema extension attributes is to create an application specifically to be the point of registration for all the schema extensions you need. It doesn't have to be a real application and this technique means that all the extensions have the same application ID in their name.

For example, here is a claims-mapping policy to emit a single claim from a directory schema extension attribute in an OAuth/OIDC token:

```
{  
    "ClaimsMappingPolicy": {  
        "Version": 1,  
        "IncludeBasicClaimSet": "false",  
        "ClaimsSchema": [{  
            "Source": "User",  
            "ExtensionID": "extension_xxxxxxx_test",  
            "JWTClaimType": "http://schemas.contoso.com/identity/claims/exampleclaim"  
        }]  
    }  
}
```

Where `xxxxxx` is the application ID the extension was registered with.

TIP

Case consistency is important when setting directory extension attributes on objects. Extension attribute names aren't case sensitive when being set up, but they are case sensitive when being read from the directory by the token service. If an extension attribute is set on a user object with the name "LegacyId" and on another user object with the name "legacyid", when the attribute is mapped to a claim using the name "LegacyId" the data will be successfully retrieved and the claim included in the token for the first user but not the second.

The "Id" parameter in the claims schema used for built-in directory attributes is "ExtensionID" for directory extension attributes.

Next steps

- Learn how to [add custom or additional claims to the SAML 2.0 and JSON Web Tokens \(JWT\) tokens](#).
- Learn how to [customize claims emitted in tokens for a specific app](#).

Customize claims issued in the SAML token for enterprise applications

4/12/2022 • 12 minutes to read • [Edit Online](#)

Today, the Microsoft identity platform 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 the Microsoft identity platform using the SAML 2.0 protocol, the Microsoft identity platform 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, the Microsoft identity platform 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 configuration interface for a SAML-based application. At the top, there are three buttons: 'Change single sign-on mode', 'Switch to the old experience', and 'Test this application'. Below this 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 red arrow points to the 'Provide feedback' link. The main area is titled 'Set up Single Sign-On with SAML - Preview'. It contains two sections: 'Basic SAML Configuration' and 'User Attributes & Claims'. The 'Basic SAML Configuration' section includes fields for Reply URL (Assertion Consumer Service URL), Identifier (Entity ID), Sign on URL, and Relay State. The 'User Attributes & Claims' section maps attributes like Givenname, Surname, Emailaddress, Name, and Unique User Identifier to their corresponding SAML values (user.givenname, user.surname, user.mail, user.userprincipalname, user.userprincipalname).

Attribute	Value
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: Attribute (selected) Transformation

* Source attribute: user.employeeid

Claim Conditions

NameID format

If the SAML request contains the element NameIDPolicy with a specific format, then the Microsoft identity platform will honor the format in the request.

If the SAML request doesn't contain an element for NameIDPolicy, then the Microsoft identity platform will issue the NameID with the format you specify. If no format is specified, the Microsoft identity platform will use the default source format associated with the claim source selected. If a transformation results in a null or illegal value, Azure AD will send a persistent pairwise identifier in the nameidentifier.

From the Choose name identifier format dropdown, you can select one of the following options.

NAMEID FORMAT	DESCRIPTION
Default	Microsoft identity platform will use the default source format.
Persistent	Microsoft identity platform will use Persistent as the NameID format.
Email address	Microsoft identity platform will use EmailAddress as the NameID format.
Unspecified	Microsoft identity platform will use Unspecified as the NameID format.
Windows domain qualified name	Microsoft identity platform will use the WindowsDomainQualifiedName 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
onpremisessamaccountname	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).
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

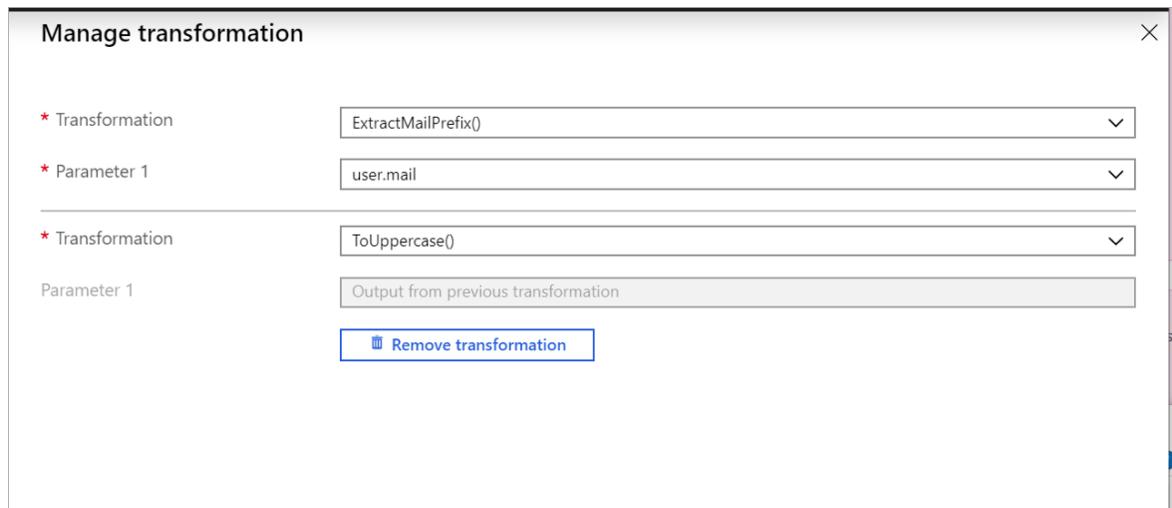
To add application-specific claims:

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. (preview) **Treat source as multivalued** is a checkbox indicating if the transform should be applied to all values or just the first. By default, transformations will only be applied to the first element in a multi value claim, by checking this box it ensures it is applied to all. This checkbox will only be enabled for multi valued attributes, for example `user.proxyaddresses`.
4. 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() function has specific behaviour when the transformation input has a domain part. It will remove the domain part from input before joining it with the separator and the selected parameter. For example, if the input of the transformation is 'joe_smith@contoso.com' and the separator is '@' and the parameter is 'fabrikam.com', this will result in joe_smith@fabrikam.com.
ToLowercase()	Converts the characters of the selected attribute into lowercase characters.
ToUppercase()	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> Parameter 2 (output): user.email Parameter 3 (output if there's no match): user:userprincipalname

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	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".
Extract() - Before matching	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".
Extract() - Between matching	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".
ExtractAlpha() - Prefix	Returns the prefix alphabetical part of the string. For example, if the input's value is "BSimon_123", then it returns "BSimon".
ExtractAlpha() - Suffix	Returns the suffix alphabetical part of the string. For example, if the input's value is "123_Simon", then it returns "Simon".
ExtractNumeric() - Prefix	Returns the prefix numerical part of the string. For example, if the input's value is "123_BSimon", then it returns "123".
ExtractNumeric() - Suffix	Returns the suffix numerical part of the string. For example, if the input's value is "BSimon_123", then it returns "123".

FUNCTION	DESCRIPTION
IfEmpty()	<p>Outputs an attribute or constant if the input is null or empty.</p> <p>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.</p> <p>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
Substring() – Fixed Length (Preview)	<p>Extracts parts of a string claim type, beginning at the character at the specified position, and returns the specified number of characters.</p> <p>SourceClaim - The claim source which the transform should be executed.</p> <p>StartIndex - The zero-based starting character position of a substring in this instance.</p> <p>Length - The length in characters of the substring.</p> <p>For example:</p> <ul style="list-style-type: none"> sourceClaim – PleaseExtractThisNow StartIndex – 6 Length – 11 Output: ExtractThis
Substring() – EndOfString (Preview)	<p>Extracts parts of a string claim type, beginning at the character at the specified position, and returns the rest of the claim from the specified start index.</p> <p>SourceClaim - The claim source which the transform should be executed.</p> <p>StartIndex - The zero-based starting character position of a substring in this instance.</p> <p>For example:</p> <ul style="list-style-type: none"> sourceClaim – PleaseExtractThisNow StartIndex – 6 Output: ExtractThisNow

If you need additional transformations, submit your idea in the [feedback forum in Azure AD](#) under the *SaaS application* category.

Add the UPN claim to SAML tokens

The `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn` claim is part of the [SAML restricted claim set](#), so you can not add it in the **User Attributes & Claims** section. As a workaround, you can add it as an [optional claim](#) through **App registrations** in the Azure portal.

Open the app in **App registrations** and select **Token configuration** and then **Add optional claim**. Select the **SAML** token type, choose **upn** from the list, and click **Add** to get the claim in the token.

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 50 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 first evaluates all conditions with source `Attribute` and then evaluates all conditions with source `Transformation` to decide which value to emit in the claim. Conditions with the same source are evaluated from top to bottom. The last value which matches the expression will be emitted in the claim. Transformations such as `IsNotEmpty` and `Contains` act like additional restrictions.

For example, Britta 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 Britta tries to sign in to Fabrikam, the Microsoft identity platform will evaluate the conditions as follow.

First, the Microsoft identity platform verifies if Britta's user type is **All guests**. Since, this is true then the Microsoft identity platform assigns the source for the claim to `user.extensionattribute1`. Second, the Microsoft identity platform verifies if Britta's user type is **AAD guests**, since this is also true then the Microsoft identity platform assigns the source for the claim to `user.mail`. Finally, the claim is emitted with value `user.mail` for Britta.

Manage claim

Save Discard changes

* Name: `Username` ✓

Namespace: `Enter a namespace URI`

* Source: Attribute Transformation

* Source attribute: `Select from drop down or type a constant` ▾

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	<code>user.extensionattribute1</code>	...
AAD Guests	0 groups	Attribute	<code>user.mail</code>	...

`Select from drop down` ▾ `Select Groups` Attribute Transformation

As another example, consider when Britta Simon tries to sign in and the following configuration is used. Azure AD first evaluates all conditions with source `Attribute`. Because Britta's user type is **AAD guests**, `user.mail` is assigned as the source for the claim. Next, Azure AD evaluates the transformations. Because Britta is a guest, `user.extensionattribute1` is now the new source for the claim. Because Britta is in **AAD guests**, `user.othermail` is now the source for this claim. Finally, the claim is emitted with value `user.othermail` for Britta.

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	Transformation	IfNotEmpty (user.extensionattribute1)	...
AAD guests	0 groups	Transformation	IfNotEmpty (user.othermail)	...
AAD guests	0 groups	Attribute	user.mail	...

Select from drop down Select groups Attribute Transformation Select a User type and Source to enable the list

As a final example, let's consider what happens if Britta has no `user.othermail` configured or it is empty. In both cases the condition entry is ignored, and the claim will fall back to `user.extensionattribute1` instead.

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](#)

Customize claims emitted in tokens for a specific app in a tenant

4/12/2022 • 12 minutes to read • [Edit Online](#)

A claim is information that an identity provider states about a user inside the token they issue for that user. Claims customization 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.

Claims customization supports configuring claim-mapping policies for the WS-Fed, SAML, OAuth, and OpenID Connect protocols.

NOTE

This feature replaces and supersedes the [claims customization](#) offered through the Azure portal. On the same application, if you customize claims using the portal in addition to the Microsoft 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.

In this article, we walk through a few common scenarios that can help you understand how to use the [claims-mapping policy type](#).

Get started

In the following examples, you create, update, link, and delete policies for service principals. Claims-mapping policies can only be assigned to service principal objects. 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.

When creating a claims-mapping policy, you can also emit a claim from a directory schema extension attribute in tokens. Use *ExtensionID* for the extension attribute instead of *ID* in the `ClaimsSchema` element. For more info on extension attributes, see [Using directory schema extension attributes](#).

NOTE

The [Azure AD PowerShell Module public preview release](#) is required to configure claims-mapping policies. The PowerShell module is in preview, while the claims mapping and token creation runtime in Azure is generally available. Updates to the preview PowerShell module could require you to update or change your configuration scripts.

To get started, do the following steps:

1. Download the latest [Azure AD PowerShell Module public preview release](#).
2. Run the `Connect-AzureAD` 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
```

Next, create a claims mapping policy and assign it to a service principal. See these examples for common scenarios:

- [Omit the basic claims from tokens](#)
- [Include the EmployeeID and TenantCountry as claims in tokens](#)
- [Use a claims transformation in tokens](#)

After creating a claims mapping policy, configure your application to acknowledge that tokens will contain customized claims. For more information, read [security considerations](#).

Omit the basic claims from tokens

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 the Microsoft Graph API](#). Or, in [Microsoft 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>
```

Include the EmployeeID and TenantCountry as claims in tokens

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/region 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/employeeid","JwtClaimType":"employeeid"},  
         {"Source":"company","ID":"tenantcountry","SamlClaimType":"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/country","JwtClaimType":"country"}]}}) -DisplayName "ExtraClaimsExample" -Type "ClaimsMappingPolicy"
```

WARNING

When you define a claims mapping policy for a directory extension attribute, use the `ExtensionID` property instead of the `ID` property within the body of the `ClaimsSchema` array.

- 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 the Microsoft Graph API](#). Or, in [Microsoft 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>
```

Use a claims transformation in tokens

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 the Microsoft Graph API](#). Or, in [Microsoft 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>
```

Security considerations

Applications that receive tokens rely on the fact that the claim values are authoritatively issued by Azure AD and cannot be tampered with. However, when you modify the token contents through claims-mapping policies, these assumptions may no longer be correct. Applications must explicitly acknowledge that tokens have been modified by the creator of the claims-mapping policy to protect themselves from claims-mapping policies created by malicious actors. This can be done in one of the following ways:

- [Configure a custom signing key](#)
- Or, [update the application manifest](#) to accept mapped claims.

Without this, Azure AD will return an [AADSTS50146](#) error code.

Configure a custom signing key

For multi-tenant apps, a custom signing key should be used. Do not set `acceptMappedClaims` in the app manifest. If you set up an app in the Azure portal, you get an app registration object and a service principal in your tenant. That app is using the Azure global sign-in key, which cannot be used for customizing claims in tokens. To get custom claims in tokens, create a custom sign-in key from a certificate and add it to service principal. For testing purposes, you can use a self-signed certificate. After configuring the custom signing key, your application code needs to [validate the token signing key](#).

Add the following information to the service principal:

- Private key (as a [key credential](#))
- Password (as a [password credential](#))
- Public key (as a [key credential](#))

Extract the private and public key base-64 encoded from the PFX file export of your certificate. Make sure that the `keyId` for the `keyCredential` used for "Sign" matches the `keyId` of the `passwordCredential`. You can generate the `customKeyIdentifier` by getting the hash of the cert's thumbprint.

Request

The following shows the format of the HTTP PATCH request to add a custom signing key to a service principal. The "key" value in the `keyCredentials` property is shortened for readability. The value is base-64 encoded. For the private key, the property usage is "Sign". For the public key, the property usage is "Verify".

```
PATCH https://graph.microsoft.com/v1.0/servicePrincipals/f47a6776-bca7-4f2e-bc6c-eec59d058e3e
```

```
Content-type: servicePrincipals/json
```

```
Authorization: Bearer {token}
```

```
{
  "keyCredentials": [
    {
      "customKeyIdentifier": "1Y85bR8r6yWTW6jnciNEONwlVhDyiQjdVLgPDnkI5mA=",
      "endDateTime": "2021-04-22T22:10:13Z",
      "keyId": "4c266507-3e74-4b91-aeba-18a25b450f6e",
      "startDateTime": "2020-04-22T21:50:13Z",
      "type": "X509CertAndPassword",
      "usage": "Sign",
      "key": "MIKIAIBAz.....HBgUrDgMCERE20nuTptI9MEFCh2Ih2jaaLZBZGeZBRFVNxeZmAAgIHOA==",
      "displayName": "CN=contoso"
    },
    {
      "customKeyIdentifier": "1Y85bR8r6yWTW6jnciNEONwlVhDyiQjdVLgPDnkI5mA=",
      "endDateTime": "2021-04-22T22:10:13Z",
      "keyId": "e35a7d11-fef0-49ad-9f3e-aacbe0a42c42",
      "startDateTime": "2020-04-22T21:50:13Z",
      "type": "AsymmetricX509Cert",
      "usage": "Verify",
      "key": "MIIDJzCCAg+gAw.....CTxQvJ/zN3bafeesMSueR83h1CSyg==",
      "displayName": "CN=contoso"
    }
  ],
  "passwordCredentials": [
    {
      "customKeyIdentifier": "1Y85bR8r6yWTW6jnciNEONwlVhDyiQjdVLgPDnkI5mA=",
      "keyId": "4c266507-3e74-4b91-aeba-18a25b450f6e",
      "endDateTime": "2022-01-27T19:40:33Z",
      "startDateTime": "2020-04-20T19:40:33Z",
      "secretText": "mypassword"
    }
  ]
}
```

Configure a custom signing key using PowerShell

Use PowerShell to [instantiate an MSAL Public Client Application](#) and use the [Authorization Code Grant](#) flow to obtain a delegated permission access token for Microsoft Graph. Use the access token to call Microsoft Graph and configure a custom signing key for the service principal. After configuring the custom signing key, your application code needs to [validate the token signing key](#).

To run this script you need:

1. The object ID of your application's service principal, found in the **Overview** blade of your application's entry in [Enterprise Applications](#) in the Azure portal.
2. An app registration to sign in a user and get an access token to call Microsoft Graph. Get the application (client) ID of this app in the **Overview** blade of the application's entry in [App registrations](#) in the Azure portal. The app registration should have the following configuration:
 - A redirect URI of "http://localhost" listed in the **Mobile and desktop applications** platform configuration
 - In **API permissions**, Microsoft Graph delegated permissions **Application.ReadWrite.All** and **User.Read** (make sure you grant Admin consent to these permissions)
3. A user who logs in to get the Microsoft Graph access token. The user should be one of the following Azure AD administrative roles (required to update the service principal):
 - Cloud Application Administrator

- Application Administrator
 - Global Administrator
4. A certificate to configure as a custom signing key for our application. You can either create a self-signed certificate or obtain one from your trusted certificate authority. The following certificate components are used in the script:
- public key (typically a .cer file)
 - private key in PKCS#12 format (in .pfx file)
 - password for the private key (pfx file)

IMPORTANT

The private key must be in PKCS#12 format since Azure AD does not support other format types. Using the wrong format can result in the error "Invalid certificate: Key value is invalid certificate" when using Microsoft Graph to PATCH the service principal with a `keyCredentials` containing the certificate info.

```
$fqdn="fourthcoffeetest.onmicrosoft.com" # this is used for the 'issued to' and 'issued by' field of the
certificate
$pwd="mypassword" # password for exporting the certificate private key
$location="C:\\\\temp" # path to folder where both the pfx and cer file will be written to

# Create a self-signed cert
$cert = New-SelfSignedCertificate -certstorelocation cert:\\currentuser\\my -DnsName $fqdn
$pwdSecure = ConvertTo-SecureString -String $pwd -Force -AsPlainText
$path = 'cert:\\currentuser\\my\\' + $cert.Thumbprint
$cerFile = $location + "\\\" + $fqdn + ".cer"
$pfxFile = $location + "\\\" + $fqdn + ".pfx"

# Export the public and private keys
Export-PfxCertificate -cert $path -FilePath $pfxFile -Password $pwdSecure
Export-Certificate -cert $path -FilePath $cerFile

$ClientID = "<app-id>"
$loginURL      = "https://login.microsoftonline.com"
$tenantdomain   = "fourthcoffeetest.onmicrosoft.com"
$redirectURL = "http://localhost" # this reply URL is needed for PowerShell Core
[string[]] $Scopes = "https://graph.microsoft.com/.default"
$pfxxpath = $pfxFile # path to pfx file
$cerpath = $cerFile # path to cer file
$SPOID = "<service-principal-id>"
$graphuri = "https://graph.microsoft.com/v1.0/serviceprincipals/$SPOID"
$password = $pwd # password for the pfx file

# choose the correct folder name for MSAL based on PowerShell version 5.1 (.Net) or PowerShell Core (.Net
Core)

if ($PSVersionTable.PSVersion.Major -gt 5)
{
    $core = $true
    $foldername = "netcoreapp2.1"
}
else
{
    $core = $false
    $foldername = "net45"
}

# Load the MSAL/microsoft.identity/client assembly -- needed once per PowerShell session
[System.Reflection.Assembly]::LoadFrom((Get-ChildItem
C:/Users/<username>/nuget/packages/microsoft.identity.client/4.32.1/lib/$foldername/Microsoft.Identity.Client.dll).fullname) | out-null
```

```

$global:app = $null

$ClientApplicationBuilder = [Microsoft.Identity.Client.PublicClientApplicationBuilder]::Create($ClientID)
[void]$ClientApplicationBuilder.WithAuthority($"$loginURL/$tenantdomain")
[void]$ClientApplicationBuilder.WithRedirectUri($redirectURL)

$global:app = $ClientApplicationBuilder.Build()

Function Get-GraphAccessTokenFromMSAL {
    [Microsoft.Identity.Client.AuthenticationResult] $authResult = $null
    $AquireTokenParameters = $global:app.AcquireTokenInteractive($Scopes)
    [IntPtr] $ParentWindow = [System.Diagnostics.Process]::GetCurrentProcess().MainWindowHandle
    if ($ParentWindow)
    {
        [void]$AquireTokenParameters.WithParentActivityOrWindow($ParentWindow)
    }
    try {
        $authResult = $AquireTokenParameters.ExecuteAsync().GetAwaiter().GetResult()
    }
    catch {
        $ErrorMessage = $_.Exception.Message
        Write-Host $ErrorMessage
    }

    return $authResult
}

$myvar = Get-GraphAccessTokenFromMSAL
if ($myvar)
{
    $GraphAccessToken = $myvar.AccessToken
    Write-Host "Access Token: " $myvar.AccessToken
    #$GraphAccessToken = "eyJ0eXAiOiJKV1QiL ... iPxstltKQ"

    # this is for PowerShell Core
    $Secure_String_Pwd = ConvertTo-SecureString $password -AsPlainText -Force

    # reading certificate files and creating Certificate Object
    if ($core)
    {
        $pfx_cert = get-content $pfxpath -AsByteStream -Raw
        $cer_cert = get-content $cerpath -AsByteStream -Raw
        $cert = Get-PfxCertificate -FilePath $pfxpath -Password $Secure_String_Pwd
    }
    else
    {
        $pfx_cert = get-content $pfxpath -Encoding Byte
        $cer_cert = get-content $cerpath -Encoding Byte
        # Write-Host "Enter password for the pfx file..."
        # calling Get-PfxCertificate in PowerShell 5.1 prompts for password
        # $cert = Get-PfxCertificate -FilePath $pfxpath
        $cert = [System.Security.Cryptography.X509Certificates.X509Certificate2]::new($pfxpath, $password)
    }

    # base 64 encode the private key and public key
    $base64pfx = [System.Convert]::ToBase64String($pfx_cert)
    $base64cer = [System.Convert]::ToBase64String($cer_cert)

    # getting id for the keyCredential object
    $guid1 = New-Guid
    $guid2 = New-Guid

    # get the custom key identifier from the certificate thumbprint:
    $hasher = [System.Security.Cryptography.HashAlgorithm]::Create('sha256')
    $hash = $hasher.ComputeHash([System.Text.Encoding]::UTF8.GetBytes($cert.Thumbprint))
    $customKeyIdentifier = [System.Convert]::ToBase64String($hash)
}

```

```

# get end date and start date for our keycredentials
$endDateTime = ($cert.NotAfter).ToUniversalTime().ToString( "yyyy-MM-ddTHH:mm:ssZ" )
$startDateTime = ($cert.NotBefore).ToUniversalTime().ToString( "yyyy-MM-ddTHH:mm:ssZ" )

# building our json payload
$object = [ordered]@{
    keyCredentials = @(
        [ordered]@{
            customKeyIdentifier = $customKeyIdentifier
            endDateTime = $endDateTime
            keyId = $guid1
            startTime = $startDateTime
            type = "X509CertAndPassword"
            usage = "Sign"
            key = $base64pfx
            displayName = "CN=fourthcoffeetest"
        },
        [ordered]@{
            customKeyIdentifier = $customKeyIdentifier
            endDateTime = $endDateTime
            keyId = $guid2
            startTime = $startDateTime
            type = "AsymmetricX509Cert"
            usage = "Verify"
            key = $base64cer
            displayName = "CN=fourthcoffeetest"
        }
    )
    passwordCredentials = @(

        [ordered]@{
            customKeyIdentifier = $customKeyIdentifier
            keyId = $guid1
            endDateTime = $endDateTime
            startTime = $startDateTime
            secretText = $password
        }
    )
}

$json = $object | ConvertTo-Json -Depth 99
Write-Host "JSON Payload:"
Write-Output $json

# Request Header
$header = @{}
$header.Add("Authorization", "Bearer $($GraphAccessToken)")
$header.Add("Content-Type", "application/json")

try
{
    Invoke-RestMethod -Uri $graphuri -Method "PATCH" -Headers $header -Body $json
}
catch
{
    # Dig into the exception to get the Response details.
    # Note that value__ is not a typo.
    Write-Host "StatusCode:" $_.Exception.Response.StatusCode.value__
    Write-Host "StatusDescription:" $_.Exception.Response.StatusDescription
}

Write-Host "Complete Request"
}
else
{
    Write-Host "Fail to get Access Token"
}

```

Validate token signing key

Apps that have claims mapping enabled must validate their token signing keys by appending `appid={client_id}` to their [OpenID Connect metadata requests](#). Below is the format of the OpenID Connect metadata document you should use:

```
https://login.microsoftonline.com/{tenant}/v2.0/.well-known/openid-configuration?appid={client-id}
```

Update the application manifest

For single tenant apps, you can set the `acceptMappedClaims` property to `true` in the [application manifest](#). As documented on the [apiApplication resource type](#), this allows an application to use claims mapping without specifying a custom signing key.

WARNING

Do not set `acceptMappedClaims` property to `true` for multi-tenant apps, which can allow malicious actors to create claims-mapping policies for your app.

This does require the requested token audience to use a verified domain name of your Azure AD tenant, which means you should ensure to set the `Application ID URI` (represented by the `identifierUris` in the application manifest) for example to `https://contoso.com/my-api` or (simply using the default tenant name)

```
https://contoso.onmicrosoft.com/my-api
```

If you're not using a verified domain, Azure AD will return an `AADSTS501461` error code with message *"AcceptMappedClaims is only supported for a token audience matching the application GUID or an audience within the tenant's verified domains. Either change the resource identifier, or use an application-specific signing key."*

Next steps

- Read the [claims-mapping policy type](#) reference article to learn more.
- 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](#)
- To learn more about extension attributes, see [Using directory schema extension attributes in claims](#).

Configure token lifetime policies (preview)

4/12/2022 • 2 minutes to read • [Edit Online](#)

You can specify the lifetime of an access, SAML, or ID token issued by Microsoft identity platform. 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. For more info, read [configurable token lifetimes](#).

In this section, we walk through a common policy scenario that can help you impose new rules for token lifetime. In the example, you learn how to create a policy that requires users to authenticate more frequently in your web app.

Get started

To get started, download the latest [Azure AD PowerShell Module Public Preview release](#).

Next, 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
```

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 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 to two hours.

To create the policy, run the `New-AzureADPolicy` cmdlet:

```
$policy = New-AzureADPolicy -Definition @('{"TokenLifetimePolicy": {"Version":1,"AccessTokenLifetime":"02:00:00"} }') -DisplayName "WebPolicyScenario" -IsOrganizationDefault $false -Type "TokenLifetimePolicy"
```

To see your new policy, and to get the policy `ObjectId`, run the `Get-AzureADPolicy` cmdlet:

```
Get-AzureADPolicy -Id $policy.Id
```

2. Assign the policy to your service principal. You also need to get the `ObjectId` of your service principal.

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>'"
```

When you have the service principal, run the `Add-AzureADServicePrincipalPolicy` cmdlet:

```
# Assign policy to a service principal  
Add-AzureADServicePrincipalPolicy -Id $sp.ObjectId -RefObjectId $policy.Id
```

View existing policies in a tenant

To see all policies that have been created in your organization, run the [Get-AzureADPolicy](#) cmdlet. Any results with defined property values that differ from the defaults listed above are in scope of the retirement.

```
Get-AzureADPolicy -All $true
```

To see which apps and service principals are linked to a specific policy you identified run the following [Get-AzureADPolicyAppliedObject](#) cmdlet by replacing **1a37dad8-5da7-4cc8-87c7-efbc0326cf20** with any of your policy IDs. Then you can decide whether to configure Conditional Access sign-in frequency or remain with the Azure AD defaults.

```
Get-AzureADPolicyAppliedObject -id 1a37dad8-5da7-4cc8-87c7-efbc0326cf20
```

If your tenant has policies which define custom values for the refresh and session token configuration properties, Microsoft recommends you update those policies to values that reflect the defaults described above. If no changes are made, Azure AD will automatically honor the default values.

Troubleshooting

Some users have reported a `Get-AzureADPolicy : The term 'Get-AzureADPolicy' is not recognized` error after running the `Get-AzureADPolicy` cmdlet. As a workaround, run the following to uninstall/re-install the AzureAD module and then install the AzureADPreview module:

```
# Uninstall the AzureAD Module  
UnInstall-Module AzureAD  
  
# Install the AzureAD Preview Module adding the -AllowClobber  
Install-Module AzureADPreview -AllowClobber  
Note: You cannot install both the preview and the GA version on the same computer at the same time.  
  
Connect-AzureAD  
Get-AzureADPolicy -All $true
```

Next steps

Learn about [authentication session management capabilities](#) in Azure AD Conditional Access.

Configure the role claim issued in the SAML token for enterprise applications

4/12/2022 • 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.

NOTE

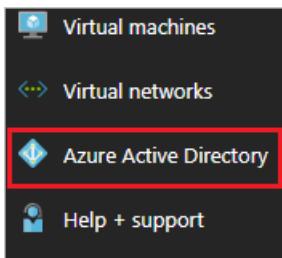
This article explains on how to create/update/delete application roles on the service principal using APIs in Azure AD. If you would like to use the new user interface for App Roles then please see the details [here](#).

When to use this feature

Use this feature if your application expects custom roles in the SAML response returned by Azure AD. You can create as many roles as you need.

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 blades from the Azure Active Directory interface. The left blade is titled 'Contoso' and has a sidebar with various icons. The right blade is also titled 'Contoso' and is specifically for 'Enterprise applications'. It features a search bar at the top, followed by sections for 'Overview', 'Quick start', 'MANAGE' (with 'All applications' highlighted), 'ACTIVITY' (with 'Sign-ins' and 'Audit logs'), and 'SUPPORT + TROUBLESHOOTING' (with 'New support request' and 'Troubleshoot').

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 redacted application name. Below it, a message says '1 applications matched [redacted]. Choose one below or...'. A table lists the application details:

NAME	CATEGORY
[redacted]	[redacted]

5. After the application is added, go to the **Properties** page and copy the object ID.

MANAGE

- Properties**
- Users and groups
- Single sign-on
- Provisioning
- Self-service

SECURITY

- Conditional access
- Permissions

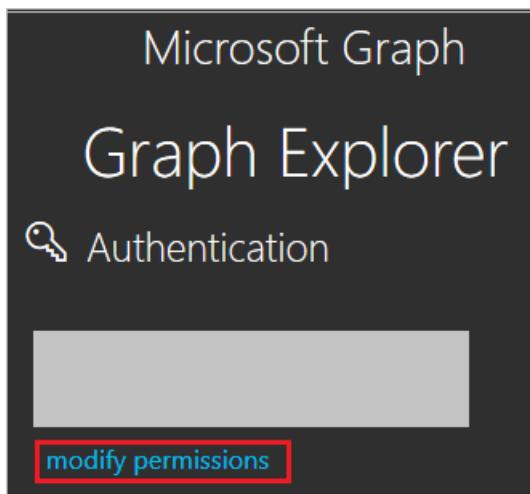
ACTIVITY

- Sign-ins
- Audit logs

Name	[REDACTED]
Publisher	Active Directory Application Registry
Homepage URL	[REDACTED]
Logo	
User access URL	https://myapps.microsoft.com/signin/ [REDACTED]
Application ID	[REDACTED]
Object ID	[REDACTED]

6. Open [Microsoft Graph Explorer](#) in another window and take the following steps:

- a. Sign in to the Graph Explorer site by using the global admin or coadmin credentials for your tenant.
- b. You need sufficient permissions to create the roles. Select **modify permissions** to get the permissions.



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.

- c. 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 checked="" type="checkbox"/> Group.Read.All	Admin
ⓘ 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

- 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:

- 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>>

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/0002ce4f-9f27-4900-80...'. A red box highlights the 'Run query' button. Below the URL, there are tabs for 'Request body', 'Request headers', 'Modify permissions', and 'Access token'. The 'Request body' tab is selected. In the main area, there is a large text input field. At the bottom of the interface, a green bar indicates a 'OK - 200 - 277ms' response. A message below it says 'When you use Microsoft Graph APIs, you agree to the Microsoft APIs Terms of Use. View the Microsoft Privacy Statement'. The 'Response preview' tab is selected, showing a JSON response snippet:

```
{ "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity", "id": "0002ce4f-9f27-4900-80c1-6bfc048b384d", "deletedDateTime": null, "accountEnabled": true, "alternativeNames": [ ] }
```

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 } ]
```

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.

NOTE

When you are creating multiple roles, please don't modify the default role content just add the new msiam_access block of code for the new role.

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"
    }
  ]
}
```

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 'Run Query' button. Below the URL bar, there are tabs for 'Request Body' and 'Request Headers', with 'Request Body' being active. The 'Request Body' tab contains a JSON payload for the PATCH request:

```
"allowedMemberTypes": [
  "User"
],
"description": "Administrators Only",
"displayName": "Admin",
"id": "4f8f8640-f081-492d-97a0-caf24e9bc134",
"isEnabled": true,
"origin": "ServicePrincipal",
"value": "Administrator"
```

At the bottom of the interface, a green status bar indicates '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.

To assign the role to any user, select the new role and select the **Assign** button on the bottom of the page.

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

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.

- 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](#).
2. Sign in to the Graph Explorer site by using the global admin or coadmin credentials for your tenant.
3. 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
```

- 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. At the top, there are dropdown menus for 'GET' method, 'beta' version, and the URL 'https://graph.microsoft.com/beta/servicePrincipals/0002ce4f-9f27-4900-80...'. A red box highlights the 'Run query' button. Below the header, there are tabs for 'Request body', 'Request headers', 'Modify permissions', and 'Access token', with 'Request body' being the active tab. The main area contains a large text input field. Underneath, a green bar indicates a successful response: 'OK - 200 - 277ms'. A note below it states: 'When you use Microsoft Graph APIs, you agree to the Microsoft APIs Terms of Use. View the Microsoft Privacy Statement'. At the bottom, there are tabs for 'Response preview', 'Response headers', 'Code snippets', 'Toolkit component', 'Adaptive cards', 'Expand', and 'Share', with 'Response preview' being the active tab. The preview area displays the JSON response:

```
{ "@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity", "id": "0002ce4f-9f27-4900-80c1-6bfc048b384d", "deletedDateTime": null, "accountEnabled": true, "alternativeNames": [ ] }
```

5. 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 bar, there are tabs for 'Request Body' and 'Request Headers'. The 'Request Body' tab is active, showing a JSON object with several properties. A red box highlights the 'description' field, which is set to 'student'. Other fields include 'allowedMemberTypes' (User), 'displayName' (student), 'id' (b31a14ab-5319-4bb1-a668-5626904561be), 'isEnabled' (true), and 'origin' (ServicePrincipal). The JSON object starts with a brace '{'.

- 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 after a successful GET request. The top bar has 'GET' 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 bar, there are tabs for 'Request body', 'Request headers', 'Modify permissions', and 'Access token'. The 'Request body' tab is active. The response preview shows a JSON object with an '@odata.context' field pointing to the service principals metadata and an '@odata.nextLink' field providing a link to the next page of results. A red box highlights the '@odata.context' field. The JSON object starts with a brace '{'.

- 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. At the top, there are dropdown menus for 'GET' and 'beta', and a URL bar containing 'https://graph.microsoft.com/beta/servicePrincipals/0002ce4f-9f27-4900-80...'. A large red box highlights the 'Run query' button. Below the URL bar, tabs for 'Request body', 'Request headers', 'Modify permissions', and 'Access token' are visible. The main area contains a large text input field. At the bottom, a green status bar indicates 'OK - 200 - 277ms', and a message about agreeing to Microsoft APIs Terms of Use.

```

{@odata.context": "https://graph.microsoft.com/beta/$metadata#servicePrincipals/$entity",
"id": "0002ce4f-9f27-4900-80c1-6bf048b384d",
"deletedDateTime": null,
"accountEnabled": true,
"alternativeNames": [
]
}

```

5. Extract the `appRoles` property from the service principal object.

The screenshot shows the Microsoft Graph Explorer interface with the 'Response preview' tab selected. It displays the JSON response for the service principal, specifically focusing on the 'appRoles' property which contains two roles: 'msiam_access' and 'Administrators Only'.

```

{
  "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 with the method set to 'PATCH'. The URL is 'https://graph.microsoft.com/beta/servicePrincipals/8164e784-8a01-46c9-89fd-3076bd9656d0'. The 'Request Body' tab is selected, showing the JSON payload for updating the role. The 'isEnabled' field is highlighted with a red box and set to 'false'.

```

{
  "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 **.IsEnabled** value to **false** for the role that you want to delete.
- Select **Run Query**.

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.

The role needs to be disabled before it can be removed.

Next steps

For additional steps, see the [app documentation](#).

Handle ITP in Safari and other browsers where third-party cookies are blocked

4/12/2022 • 5 minutes to read • [Edit Online](#)

Many browsers block *third-party cookies*, cookies on requests to domains other than the domain shown in the browser's address bar. This block breaks the implicit flow and requires new authentication patterns to successfully sign in users. In the Microsoft identity platform, we use the authorization flow with Proof Key for Code Exchange (PKCE) and refresh tokens to keep users signed in when third-party cookies are blocked.

What is Intelligent Tracking Protection (ITP)?

Apple Safari has an on-by-default privacy protection feature called [Intelligent Tracking Protection](#), or *ITP*. ITP blocks "third-party" cookies, cookies on requests that cross domains.

A common form of user tracking is done by loading an iframe to third-party site in the background and using cookies to correlate the user across the Internet. Unfortunately, this pattern is also the standard way of implementing the [implicit flow](#) in single-page apps (SPAs). When a browser blocks third-party cookies to prevent user tracking, SPAs are also broken.

Safari isn't alone in blocking third-party cookies to enhance user privacy. Brave blocks third-party cookies by default, and Chromium (the platform behind Google Chrome and Microsoft Edge) has announced that they as well will stop supporting third-party cookies in the future.

The solution outlined in this article works in all of these browsers, or anywhere third-party cookies are blocked.

Overview of the solution

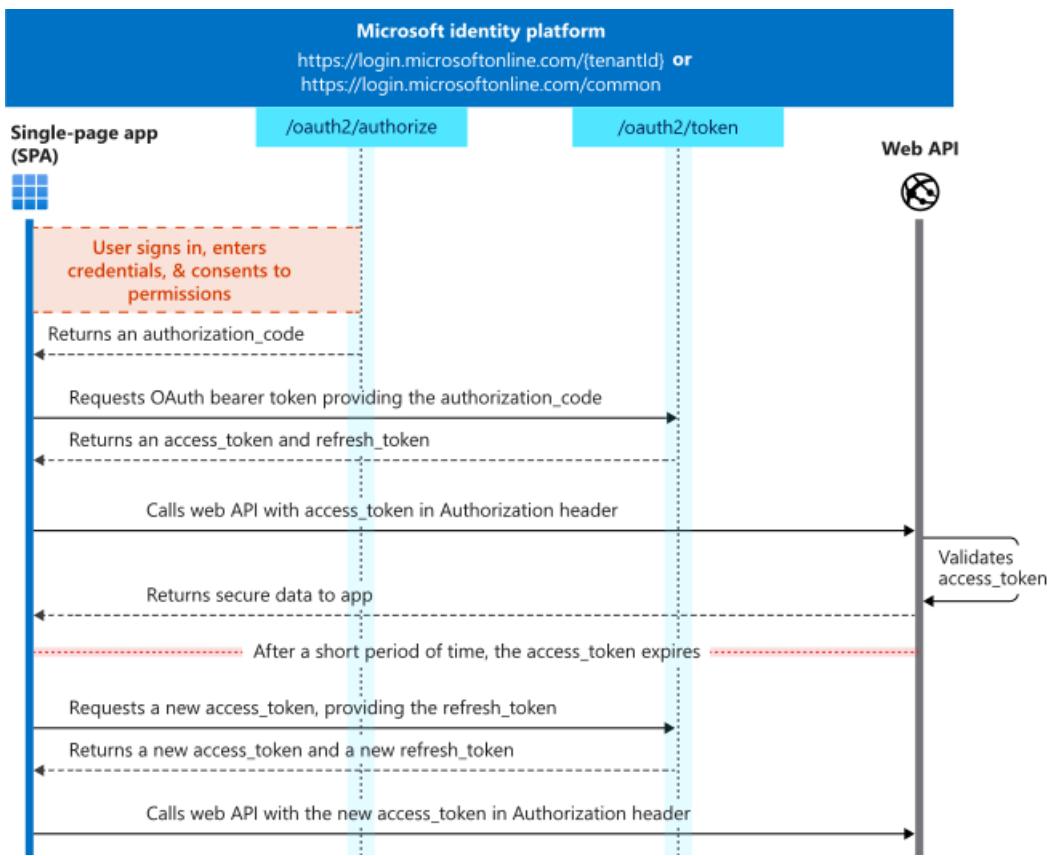
To continue authenticating users in SPAs, app developers must use the [authorization code flow](#). In the auth code flow, the identity provider issues a code, and the SPA redeems the code for an access token and a refresh token. When the app requires additional tokens, it can use the [refresh token flow](#) to get new tokens. Microsoft Authentication Library (MSAL) for JavaScript v2.0, implements the authorization code flow for SPAs and, with minor updates, is a drop-in replacement for MSAL.js 1.x.

For the Microsoft identity platform, SPAs and native clients follow similar protocol guidance:

- Use of a [PKCE code challenge](#)
 - PKCE is *required* for SPAs on the Microsoft identity platform. PKCE is *recommended* for native and confidential clients.
- No use of a client secret

SPAs have two additional restrictions:

- [The redirect URI must be marked as type `spa`](#) to enable CORS on login endpoints.
- Refresh tokens issued through the authorization code flow to `spa` redirect URIs have a 24-hour lifetime rather than a 90-day lifetime.



Performance and UX implications

Some applications using the implicit flow attempt sign-in without redirecting by opening a login iframe using `prompt=none`. In most browsers, this request will respond with tokens for the currently signed-in user (assuming consent has already been granted). This pattern meant applications didn't need a full page redirect to sign the user in, improving performance and user experience - the user visits the web page and is signed in already. Because `prompt=none` in an iframe is no longer an option when third-party cookies are blocked, applications must visit the login page in a top-level frame to have an authorization code issued.

There are two ways of accomplishing sign-in:

- **Full page redirects**
 - On the first load of the SPA, redirect the user to the sign-in page if no session already exists (or if the session is expired). The user's browser will visit the login page, present the cookies containing the user session, and then redirect back to the application with the code and tokens in a fragment.
 - The redirect does result in the SPA being loaded twice. Follow best practices for caching of SPAs so that the app isn't downloaded in-full twice.
 - Consider having a pre-load sequence in the app that checks for a login session and redirects to the login page before the app fully unpacks and executes the JavaScript payload.
- **Popups**
 - If the user experience (UX) of a full page redirect doesn't work for the application, consider using a popup to handle authentication.
 - When the popup finishes redirecting to the application after authentication, code in the redirect handler will store the code and tokens in local storage for the application to use. MSAL.js supports popups for authentication, as do most libraries.
 - Browsers are decreasing support for popups, so they may not be the most reliable option. User interaction with the SPA before creating the popup may be needed to satisfy browser requirements.

Apple [describes a popup method](#) as a temporary compatibility fix to give the original window access to third-party cookies. While Apple may remove this transferral of permissions in the future, it will not impact the guidance here.

Here, the popup is being used as a first party navigation to the login page so that a session is found and an auth code can be provided. This should continue working into the future.

Using iframes

A common pattern in web apps is to use an iframe to embed one app inside another: the top-level frame handles authenticating the user and the application hosted in the iframe can trust that the user is signed in, fetching tokens silently using the implicit flow.

Silent token acquisition no longer works when third-party cookies are blocked - the application embedded in the iframe must switch to using popups to access the user's session as it can't navigate to the login page.

You can achieve single sign-on between iframed and parent apps with same-origin *and* cross-origin JavaScript script API access by passing a user (account) hint from the parent app to the iframed app. For more information, see [Using MSAL.js in iframed apps](#) in the MSAL.js repository on GitHub.

Security implications of refresh tokens in the browser

Issuing refresh tokens to the browser is considered a security issue. Cross-site scripting (XSS) attacks or compromised JS packages can steal the refresh token and use it remotely until it expires or is revoked. In order to minimize the risk of stolen refresh tokens, SPAs will be issued tokens valid for only 24 hours. After 24 hours, the app must acquire a new authorization code via a top-level frame visit to the login page.

This limited-lifetime refresh token pattern was chosen as a balance between security and degraded UX. Without refresh tokens or third-party cookies, the authorization code flow (as recommended by the [OAuth security best current practices draft](#)) becomes onerous when new or additional tokens are required. A full page redirect or popup is needed for every single token, every time a token expires (every hour usually, for the Microsoft identity platform tokens).

Next steps

For more information about authorization code flow and MSAL.js, see:

- [Authorization code flow](#).
- [MSAL.js 2.0 quickstart](#).

Handle SameSite cookie changes in Chrome browser

4/12/2022 • 2 minutes to read • [Edit Online](#)

What is SameSite?

`SameSite` is a property that can be set in HTTP cookies to prevent Cross Site Request Forgery(CSRF) attacks in web applications:

- When `SameSite` is set to **Lax**, the cookie is sent in requests within the same site and in GET requests from other sites. It isn't sent in GET requests that are cross-domain.
- A value of **Strict** ensures that the cookie is sent in requests only within the same site.

By default, the `SameSite` value is NOT set in browsers and that's why there are no restrictions on cookies being sent in requests. An application would need to opt-in to the CSRF protection by setting **Lax** or **Strict** per their requirements.

SameSite changes and impact on authentication

Recent [updates to the standards on SameSite](#) propose protecting apps by making the default behavior of `SameSite` when no value is set to Lax. This mitigation means cookies will be restricted on HTTP requests except GET made from other sites. Additionally, a value of **None** is introduced to remove restrictions on cookies being sent. These updates will soon be released in an upcoming version of the Chrome browser.

When web apps authenticate with the Microsoft Identity platform using the response mode "form_post", the login server responds to the application using an HTTP POST to send the tokens or auth code. Because this request is a cross-domain request (from `login.microsoftonline.com` to your domain - for instance `https://contoso.com/auth`), cookies that were set by your app now fall under the new rules in Chrome. The cookies that need to be used in cross-site scenarios are cookies that hold the *state* and *nonce* values, that are also sent in the login request. There are other cookies dropped by Azure AD to hold the session.

If you don't update your web apps, this new behavior will result in authentication failures.

Mitigation and samples

To overcome the authentication failures, web apps authenticating with the Microsoft identity platform can set the `SameSite` property to `None` for cookies that are used in cross-domain scenarios when running on the Chrome browser. Other browsers (see [here](#) for a complete list) follow the previous behavior of `SameSite` and won't include the cookies if `SameSite=None` is set. That's why, to support authentication on multiple browsers web apps will have to set the `SameSite` value to `None` only on Chrome and leave the value empty on other browsers.

This approach is demonstrated in our code samples below.

- [.NET](#)
- [Python](#)
- [Java](#)

The table below presents the pull requests that worked around the SameSite changes in our ASP.NET and ASP.NET Core samples.

SAMPLE	PULL REQUEST
ASP.NET Core web app incremental tutorial	Same site cookie fix #261
ASP.NET MVC web app sample	Same site cookie fix #35
active-directory-dotnet-admin-restricted-scopes-v2	Same site cookie fix #28

for details on how to handle SameSite cookies in ASP.NET and ASP.NET Core, see also:

- [Work with SameSite cookies in ASP.NET Core .](#)
- [ASP.NET Blog on SameSite issue](#)

Next steps

Learn more about SameSite and the Web app scenario:

- [Google Chrome's FAQ on SameSite](#)
- [Chromium SameSite page](#)
- [Scenario: Web app that signs in users](#)

Workload identity federation (preview)

4/12/2022 • 4 minutes to read • [Edit Online](#)

This article provides an overview of workload identity federation for software workloads. Using workload identity federation allows you to access Azure Active Directory (Azure AD) protected resources without needing to manage secrets (for supported scenarios).

You can use workload identity federation in scenarios such as GitHub Actions and workloads running on Kubernetes.

Why use workload identity federation?

Typically, a software workload (such as an application, service, script, or container-based application) needs an identity in order to authenticate and access resources or communicate with other services. When these workloads run on Azure, you can use managed identities and the Azure platform manages the credentials for you. For a software workload running outside of Azure, you need to use application credentials (a secret or certificate) to access Azure AD protected resources (such as Azure, Microsoft Graph, Microsoft 365, or third-party resources). These credentials pose a security risk and have to be stored securely and rotated regularly. You also run the risk of service downtime if the credentials expire.

You use workload identity federation to configure an Azure AD app registration to trust tokens from an external identity provider (IdP), such as GitHub. Once that trust relationship is created, your software workload can exchange trusted tokens from the external IdP for access tokens from Microsoft identity platform. Your software workload then uses that access token to access the Azure AD protected resources to which the workload has been granted access. This eliminates the maintenance burden of manually managing credentials and eliminates the risk of leaking secrets or having certificates expire.

Supported scenarios

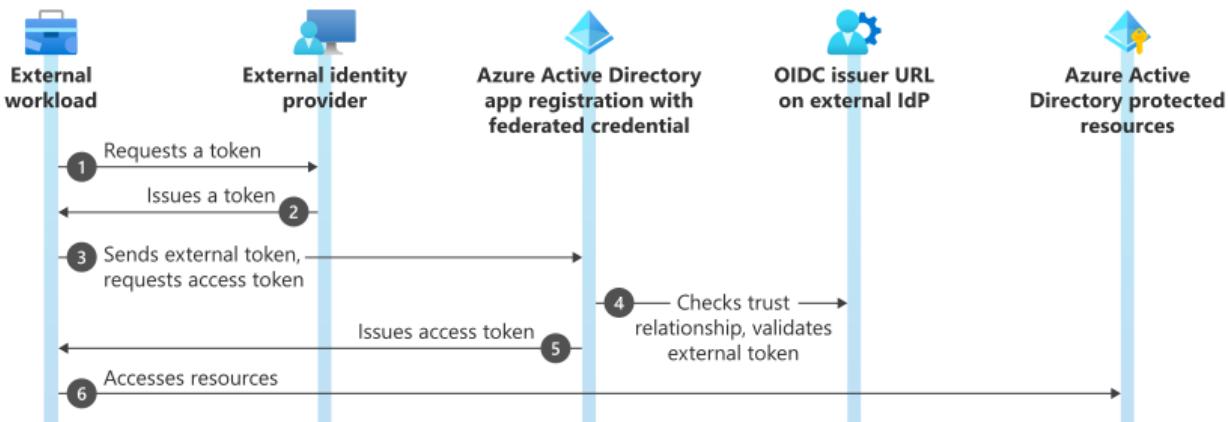
The following scenarios are supported for accessing Azure AD protected resources using workload identity federation:

- GitHub Actions. First, [Configure a trust relationship](#) between your app in Azure AD and a GitHub repo in the Azure portal or using Microsoft Graph. Then [configure a GitHub Actions workflow](#) to get an access token from Microsoft identity provider and access Azure resources.
- Google Cloud. First, configure a trust relationship between your app in Azure AD and an identity in Google Cloud. Then configure your software workload running in Google Cloud to get an access token from Microsoft identity provider and access Azure AD protected resources. See [Access Azure AD protected resources from an app in Google Cloud](#).
- Workloads running on Kubernetes. Install the Azure AD workload identity webhook and establish a trust relationship between your app in Azure AD and a Kubernetes workload (described in the [Kubernetes quickstart](#)).
- Workloads running in compute platforms outside of Azure. [Configure a trust relationship](#) between your Azure AD application registration and the external IdP for your compute platform. You can use tokens issued by that platform to authenticate with Microsoft identity platform and call APIs in the Microsoft ecosystem. Use the [client credentials flow](#) to get an access token from Microsoft identity platform, passing in the identity provider's JWT instead of creating one yourself using a stored certificate.

How it works

Create a trust relationship between the external IdP and an app in Azure AD by configuring a [federated identity credential](#). The federated identity credential is used to indicate which token from the external IdP should be trusted by your application. You configure the federated identity credential on your app registration in the Azure portal or through Microsoft Graph. The steps for configuring the trust relationship will differ, depending on the scenario and external IdP.

The workflow for exchanging an external token for an access token is the same, however, for all scenarios. The following diagram shows the general workflow of a workload exchanging an external token for an access token and then accessing Azure AD protected resources.



1. The external workload (such as a GitHub Actions workflow) requests a token from the external IdP (such as GitHub).
2. The external IdP issues a token to the external workload.
3. The external workload (the login action in a GitHub workflow, for example) [sends the token to Microsoft identity platform](#) and requests an access token.
4. Microsoft identity platform checks the [trust relationship](#) on the app registration and validates the external token against the Open ID Connect (OIDC) issuer URL on the external IdP.
5. When the checks are satisfied, Microsoft identity platform issues an access token to the external workload.
6. The external workload accesses Azure AD protected resources using the access token from Microsoft identity platform. A GitHub Actions workflow, for example, uses the access token to publish a web app to Azure App Service.

The Microsoft identity platform stores only the first 10 signing keys when they're downloaded from the external IdP's OIDC endpoint. If the external IdP exposes more than 10 signing keys, you may experience errors when using Workload Identity Federation.

Next steps

Learn more about how workload identity federation works:

- How Azure AD uses the [OAuth 2.0 client credentials grant](#) and a client assertion issued by another IdP to get a token.
- How to create, delete, get, or update [federated identity credentials](#) on an app registration using Microsoft Graph.
- Read the [GitHub Actions documentation](#) to learn more about configuring your GitHub Actions workflow to get an access token from Microsoft identity provider and access Azure resources.
- For information about the required format of JWTs created by external identity providers, read about the [assertion format](#).

Configure an app to trust an external identity provider (preview)

4/12/2022 • 8 minutes to read • [Edit Online](#)

This article describes how to create a trust relationship between an application in Azure Active Directory (Azure AD) and an external identity provider (IdP). You can then configure an external software workload to exchange a token from the external IdP for an access token from Microsoft identity platform. The external workload can access Azure AD protected resources without needing to manage secrets (in supported scenarios). To learn more about the token exchange workflow, read about [workload identity federation](#). You establish the trust relationship by configuring a federated identity credential on your app registration by using Microsoft Graph or the Azure portal.

Anyone with permissions to create an app registration and add a secret or certificate can add a federated identity credential. If the **Users can register applications** switch in the [User Settings](#) blade is set to **No**, however, you won't be able to create an app registration or configure the federated identity credential. Find an admin to configure the federated identity credential on your behalf. Anyone in the Application Administrator or Application Owner roles can do this.

After you configure your app to trust an external IdP, configure your software workload to get an access token from Microsoft identity provider and access Azure AD protected resources.

Prerequisites

[Create an app registration](#) in Azure AD. Grant your app access to the Azure resources targeted by your external software workload.

Find the object ID of the app (not the application (client) ID), which you need in the following steps. You can find the object ID of the app in the Azure portal. Go to the list of [registered applications](#) in the Azure portal and select your app registration. In **Overview -> Essentials**, find the **Object ID**.

Get the information for your external IdP and software workload, which you need in the following steps.

The Microsoft Graph beta endpoint (<https://graph.microsoft.com/beta>) exposes REST APIs to create, update, delete [federatedIdentityCredentials](#) on applications. Launch [Azure Cloud Shell](#) and sign in to your tenant in order to run Microsoft Graph commands from AZ CLI.

Configure a federated identity credential on an app

When you configure a federated identity credential on an app, there are several important pieces of information to provide.

issuer and *subject* are the key pieces of information needed to set up the trust relationship. *issuer* is the URL of the external identity provider and must match the `issuer` claim of the external token being exchanged. *subject* is the identifier of the external software workload and must match the `sub` (`subject`) claim of the external token being exchanged. *subject* has no fixed format, as each IdP uses their own - sometimes a GUID, sometimes a colon delimited identifier, sometimes arbitrary strings. The combination of `issuer` and `subject` must be unique on the app. When the external software workload requests Microsoft identity platform to exchange the external token for an access token, the *issuer* and *subject* values of the federated identity credential are checked against the `issuer` and `subject` claims provided in the external token. If that validation check passes, Microsoft identity platform issues an access token to the external software workload.

IMPORTANT

If you accidentally add the incorrect external workload information in the *subject* setting the federated identity credential is created successfully without error. The error does not become apparent until the token exchange fails.

audiences lists the audiences that can appear in the external token. This field is mandatory, and defaults to "api://AzureADTokenExchange". It says what Microsoft identity platform must accept in the `aud` claim in the incoming token. This value represents Azure AD in your external identity provider and has no fixed value across identity providers - you may need to create a new application registration in your IdP to serve as the audience of this token.

name is the unique identifier for the federated identity credential, which has a character limit of 120 characters and must be URL friendly. It is immutable once created.

description is the un-validated, user-provided description of the federated identity credential.

GitHub Actions example

- [Azure CLI](#)
- [Portal](#)

Run the [create a new federated identity credential](#) command on your app (specified by the object ID of the app). Specify the *name*, *issuer*, *subject*, and other parameters.

For examples, see [Configure an app to trust a GitHub repo](#).

Kubernetes example

- [Azure CLI](#)
- [Portal](#)

Run the following command to configure a federated identity credential on an app and create a trust relationship with a Kubernetes service account. Specify the following parameters:

- *issuer* is your service account issuer URL (the [OIDC issuer URL](#) for the managed cluster or the [OIDC Issuer URL](#) for a self-managed cluster).
- *subject* is the subject name in the tokens issued to the service account. Kubernetes uses the following format for subject names: `system:serviceaccount:<SERVICE_ACCOUNT_NAMESPACE>:<SERVICE_ACCOUNT_NAME>`.
- *name* is the name of the federated credential, which cannot be changed later.
- *audiences* lists the audiences that can appear in the 'aud' claim of the external token. This field is mandatory, and defaults to "api://AzureADTokenExchange".

```
az rest --method POST --uri 'https://graph.microsoft.com/beta/applications/f6475511-fd81-4965-a00e-41e7792b7b9c/federatedIdentityCredentials' --body '{"name": "Kubernetes-federated-credential", "issuer": "https://aksocicwesteuropa.blob.core.windows.net/9d80a3e1-2a87-46ea-ab16-e629589c541c/", "subject": "system:serviceaccount:erp8asle:pod-identity-sa", "description": "Kubernetes service account federated credential", "audiences": ["api://AzureADTokenExchange"]}'
```

And you get the response:

```
{
  "@odata.context": "https://graph.microsoft.com/beta/$metadata#applications('f6475511-fd81-4965-a00e-41e7792b7b9c')/federatedIdentityCredentials/$entity",
  "audiences": [
    "api://AzureADTokenExchange"
  ],
  "description": "Kubernetes service account federated credential",
  "id": "51ecf9c3-35fc-4519-a28a-8c27c6178bca",
  "issuer": "https://aksoicwesteurope.blob.core.windows.net/9d80a3e1-2a87-46ea-ab16-e629589c541c/",
  "name": "Kubernetes-federated-credential",
  "subject": "system:serviceaccount:erp8asle:pod-identity-sa"
}
```

Other identity providers example

- [Azure CLI](#)
- [Portal](#)

Run the following command to configure a federated identity credential on an app and create a trust relationship with an external identity provider. Specify the following parameters (using a software workload running in Google Cloud as an example):

- *name* is the name of the federated credential, which cannot be changed later.
- *ObjectID*: the object ID of the app (not the application (client) ID) you previously registered in Azure AD.
- *subject*: must match the `sub` claim in the token issued by the external identity provider. In this example using Google Cloud, *subject* is the Unique ID of the service account you plan to use.
- *issuer*: must match the `iss` claim in the token issued by the external identity provider. A URL that complies with the OIDC Discovery spec. Azure AD uses this issuer URL to fetch the keys that are necessary to validate the token. In the case of Google Cloud, the *issuer* is "https://accounts.google.com".
- *audiences*: must match the `aud` claim in the external token. For security reasons, you should pick a value that is unique for tokens meant for Azure AD. The Microsoft recommended value is "api://AzureADTokenExchange".

```
az rest --method POST --uri
'https://graph.microsoft.com/beta/applications/<ObjectID>/federatedIdentityCredentials' --body
'{"name":"GcpFederation","issuer":"https://accounts.google.com","subject":"112633961854638529490","description":"Testing","audiences":["api://AzureADTokenExchange"]}'
```

And you get the response:

```
{
  "@odata.context": "https://graph.microsoft.com/beta/$metadata#applications('f6475511-fd81-4965-a00e-41e7792b7b9c')/federatedIdentityCredentials/$entity",
  "audiences": [
    "api://AzureADTokenExchange"
  ],
  "description": "Testing",
  "id": "51ecf9c3-35fc-4519-a28a-8c27c6178bca",
  "issuer": "https://accounts.google.com",
  "name": "GcpFederation",
  "subject": "112633961854638529490"
}
```

List federated identity credentials on an app

- [Azure CLI](#)

- [Portal](#)

Run the following command to [list the federated identity credential\(s\)](#) for an app (specified by the object ID of the app):

```
az rest -m GET -u 'https://graph.microsoft.com/beta/applications/f6475511-fd81-4965-a00e-41e7792b7b9c/federatedIdentityCredentials'
```

And you get a response similar to:

```
{  
    "@odata.context": "https://graph.microsoft.com/beta/$metadata#applications('f6475511-fd81-4965-a00e-41e7792b7b9c')/federatedIdentityCredentials",  
    "value": [  
        {  
            "audiences": [  
                "api://AzureADTokenExchange"  
            ],  
            "description": "Testing",  
            "id": "1aa3e6a7-464c-4cd2-88d3-90db98132755",  
            "issuer": "https://token.actions.githubusercontent.com/",  
            "name": "Testing",  
            "subject": "repo:octo-org/octo-repo:environment:Production"  
        }  
    ]  
}
```

Delete a federated identity credential

- [Azure CLI](#)
- [Portal](#)

Run the following command to [delete a federated identity credential](#) from an app (specified by the object ID of the app):

```
az rest -m DELETE -u 'https://graph.microsoft.com/beta/applications/f6475511-fd81-4965-a00e-41e7792b7b9c/federatedIdentityCredentials/51ecf9c3-35fc-4519-a28a-8c27c6178bca'
```

Next steps

- To learn how to use workload identity federation for Kubernetes, see [Azure AD Workload Identity for Kubernetes](#) open source project.
- To learn how to use workload identity federation for GitHub Actions, see [Configure a GitHub Actions workflow to get an access token](#).
- For more information, read about how Azure AD uses the [OAuth 2.0 client credentials grant](#) and a client assertion issued by another IdP to get a token.
- For information about the required format of JWTs created by external identity providers, read about the [assertion format](#).

Configure an app to trust a GitHub repo (preview)

4/12/2022 • 6 minutes to read • [Edit Online](#)

This article describes how to create a trust relationship between an application in Azure Active Directory (Azure AD) and a GitHub repo. You can then configure a GitHub Actions workflow to exchange a token from GitHub for an access token from Microsoft identity platform and access Azure AD protected resources without needing to manage secrets. To learn more about the token exchange workflow, read about [workload identity federation](#). You establish the trust relationship by configuring a federated identity credential on your app registration in the Azure portal or by using Microsoft Graph.

Anyone with permissions to create an app registration and add a secret or certificate can add a federated identity credential. If the **Users can register applications** switch in the [User Settings](#) blade is set to **No**, however, you won't be able to create an app registration or configure the federated identity credential. Find an admin to configure the federated identity credential on your behalf. Anyone in the Application Administrator or Application Owner roles can do this.

After you configure your app to trust a GitHub repo, [configure your GitHub Actions workflow](#) to get an access token from Microsoft identity provider and access Azure AD protected resources.

Prerequisites

[Create an app registration](#) in Azure AD. Grant your app access to the Azure resources targeted by your GitHub workflow.

Find the object ID of the app (not the application (client) ID), which you need in the following steps. You can find the object ID of the app in the Azure portal. Go to the list of [registered applications](#) in the Azure portal and select your app registration. In **Overview -> Essentials**, find the **Object ID**.

Get the organization, repository, and environment information for your GitHub repo, which you need in the following steps.

Configure a federated identity credential

- [Azure portal](#)
- [Microsoft Graph](#)

Sign in to the [Azure portal](#). Go to [App registrations](#) and open the app you want to configure.

Go to [Certificates and secrets](#). In the **Federated credentials** tab, select **Add credential**. The **Add a credential** blade opens.

In the **Federated credential scenario** drop-down box select **GitHub actions deploying Azure resources**.

Specify the **Organization** and **Repository** for your GitHub Actions workflow.

For **Entity type**, select **Environment**, **Branch**, **Pull request**, or **Tag** and specify the value. The values must exactly match the configuration in the [GitHub workflow](#). For more info, read the [examples](#).

Add a **Name** for the federated credential.

The **Issuer**, **Audiences**, and **Subject identifier** fields autopopulate based on the values you entered.

Click **Add** to configure the federated credential.

Add a credential ...

Create a federated credential to connect a GitHub Actions workflow with Azure AD, enabling resources to be deployed without storing secrets.

Federated credential scenario *

Github actions deploying Azure resources



i GitHub actions is the only credential currently available during private preview. [Learn more](#)

Connect your GitHub account

Please enter the details of your GitHub Actions workflow that you want to connect with Azure Active Directory. These values will be used by Azure AD to validate the connection and should match your GitHub OIDC configuration.

Organization *

octo-org



Repository *

octo-repo



Entity type *

Environment



Github environment name *

Production



Credential details

Provide a name and description for this credential and review other details.

Name * ⓘ

Test federated credential



Federated credential description

Federated credential for my GitHub Actions workflow.



Issuer

<https://token.actions.githubusercontent.com>

[Edit \(optional\)](#)

Audience ⓘ

<api://AzureADTokenExchange>

[Edit \(optional\)](#)

Subject identifier ⓘ

<repo:octo-org/octo-repo:environment:Production>

This value is generated based on the GitHub account details provided.

Add

Cancel

NOTE

If you accidentally configure someone else's GitHub repo in the *subject* setting (enter a typo that matches someone else's repo) you can successfully create the federated identity credential. But in the GitHub configuration, however, you would get an error because you aren't able to access another person's repo.

IMPORTANT

The **Organization**, **Repository**, and **Entity type** values must exactly match the configuration on the GitHub workflow configuration. Otherwise, Microsoft identity platform will look at the incoming external token and reject the exchange for an access token. You won't get an error, the exchange fails without error.

Entity type examples

Branch example

For a workflow triggered by a push or pull request event on the main branch:

```
on:  
  push:  
    branches: [ main ]  
  pull_request:  
    branches: [ main ]
```

Specify an **Entity type of Branch** and a **GitHub branch name** of "main".

Environment example

For Jobs tied to an environment named "production":

```
on:  
  push:  
    branches:  
      - main  
  
jobs:  
  deployment:  
    runs-on: ubuntu-latest  
    environment: production  
    steps:  
      - name: deploy  
        # ...deployment-specific steps
```

Specify an **Entity type of Environment** and a **GitHub environment name** of "production".

Tag example

For example, for a workflow triggered by a push to the tag named "v2":

```
on:  
  push:  
    # Sequence of patterns matched against refs/heads  
    branches:  
      - main  
      - 'mona/octocat'  
      - 'releases/**'  
    # Sequence of patterns matched against refs/tags  
    tags:  
      - v2  
      - v1.*
```

Specify an **Entity type of Tag** and a **GitHub tag name** of "v2".

Pull request example

For a workflow triggered by a pull request event, specify an **Entity type of Pull request**.

Get the application (client) ID and tenant ID from the Azure portal

Before configuring your GitHub Actions workflow, get the *tenant-id* and *client-id* values of your app registration. You can find these values in the Azure portal. Go to the list of [registered applications](#) and select your app registration. In **Overview->Essentials**, find the **Application (client) ID** and **Directory (tenant) ID**. Set these values in your GitHub environment to use in the Azure login action for your workflow.

Next steps

For an end-to-end example, read [Deploy to App Service using GitHub Actions](#).

Read the [GitHub Actions documentation](#) to learn more about configuring your GitHub Actions workflow to get an access token from Microsoft identity provider and access Azure resources.

Access Azure AD protected resources from an app in Google Cloud (preview)

4/12/2022 • 7 minutes to read • [Edit Online](#)

Software workloads running in Google Cloud need an Azure Active Directory (Azure AD) application to authenticate and access Azure AD protected resources. A common practice is to configure that application with credentials (a secret or certificate). The credentials are used by a Google Cloud workload to request an access token from Microsoft identity platform. These credentials pose a security risk and have to be stored securely and rotated regularly. You also run the risk of service downtime if the credentials expire.

[Workload identity federation](#) allows you to access Azure AD protected resources from services running in Google Cloud without needing to manage secrets. Instead, you can configure your Azure AD application to trust a token issued by Google and exchange it for an access token from Microsoft identity platform.

Create an app registration in Azure AD

[Create an app registration](#) in Azure AD.

Take note of the *object ID* of the app (not the application (client) ID) which you need in the following steps. Go to the [list of registered applications](#) in the Azure portal, select your app registration, and find the **Object ID** in **Overview->Essentials**.

Grant your app permissions to resources

Grant your app the permissions necessary to access the Azure AD protected resources targeted by your software workload running in Google Cloud. For example, [assign the Storage Blob Data Contributor role](#) to your app if your application needs to read, write, and delete blob data in [Azure Storage](#).

Set up an identity in Google Cloud

You need an identity in Google Cloud that can be associated with your Azure AD application. A [service account](#), for example, used by an application or compute workload. You can either use the default service account of your Cloud project or create a dedicated service account.

Each service account has a unique ID. When you visit the **IAM & Admin** page in the Google Cloud console, click on **Service Accounts**. Select the service account you plan to use, and copy its **Unique ID**.

The screenshot shows the Google Cloud Platform IAM & Admin interface. On the left, there's a sidebar with options like IAM, Identity & Organization, Policy Troubleshooter, Policy Analyzer, Organization Policies, Service Accounts (which is selected), Workload Identity Federation, and Labels. The main area is titled 'Service account details' for 'mytestsaa'. It shows the Name (mytestsaa), Description (empty), Email (mytestsaa@mytest-pod-338619.iam.gserviceaccount.com), and Unique ID (112633961854638529490). The Unique ID field is highlighted with a red box.

Tokens issued by Google to the service account will have this **Unique ID** as the *subject* claim.

The *issuer* claim in the tokens will be <https://accounts.google.com>.

You need these claim values to configure a trust relationship with an Azure AD application, which allows your application to trust tokens issued by Google to your service account.

Configure an Azure AD app to trust a Google Cloud identity

Configure a federated identity credential on your Azure AD application to set up the trust relationship.

The most important fields for creating the federated identity credential are:

- *object ID*: the object ID of the app (not the application (client) ID) you previously registered in Azure AD.
- *subject*: must match the `sub` claim in the token issued by another identity provider, in this case Google. This is the Unique ID of the service account you plan to use.
- *issuer*: must match the `iss` claim in the token issued by the identity provider. A URL that complies with the [OIDC Discovery spec](#). Azure AD uses this issuer URL to fetch the keys that are necessary to validate the token. In the case of Google Cloud, the issuer is <https://accounts.google.com>.
- *audiences*: must match the `aud` claim in the token. For security reasons, you should pick a value that is unique for tokens meant for Azure AD. The Microsoft recommended value is <api://AzureADTokenExchange>.

The following command configures a federated identity credential:

```
az rest --method POST --uri 'https://graph.microsoft.com/beta/applications/41be38fd-caac-4354-aa1e-1fdb20e43bfa/federatedIdentityCredentials' --body
'{"name":"GcpFederation","issuer":"https://accounts.google.com","subject":"112633961854638529490","description":"Testing","audiences":["api://AzureADTokenExchange"]}'
```

For more information and examples, see [Create a federated identity credential](#).

Exchange a Google token for an access token

Now that you have configured the Azure AD application to trust the Google service account, you are ready to get a token from Google and exchange it for an access token from Microsoft identity platform. This code runs in an application deployed to Google Cloud and running, for example, on [App Engine](#).

Get an ID token for your Google service account

As mentioned earlier, Google cloud resources such as App Engine automatically use the default service account of your Cloud project. You can also configure the app to use a different service account when you deploy your service. Your service can [request an ID token](#) for that service account from the metadata server that handles such requests. With this approach, you don't need any keys for your service account: these are all managed by Google.

- [TypeScript](#)
- [C#](#)

Here's an example in TypeScript of how to request an ID token from the Google metadata server:

```
async function getGoogleIDToken() {
  const headers = new Headers();

  headers.append("Metadata-Flavor", "Google ");

  let aadAudience = "api://AzureADTokenExchange";

  const endpoint="http://metadata.google.internal/computeMetadata/v1/instance/service-
accounts/default/identity?audience="+ aadAudience;

  const options = {
    method: "GET",
    headers: headers,
  };

  return fetch(endpoint, options);
}
```

IMPORTANT

The *audience* here needs to match the *audiences* value you configured on your Azure AD application when [creating the federated identity credential](#).

Exchange the identity token for an Azure AD access token

Now that your app running in Google Cloud has an identity token from Google, exchange it for an access token from Microsoft identity platform. Use the [Microsoft Authentication Library \(MSAL\)](#) to pass the Google token as a client assertion. The following MSAL versions support client assertions:

- [MSAL Go \(Preview\)](#)
- [MSAL Node](#)
- [MSAL.NET](#)
- [MSAL Python](#)
- [MSAL Java](#)

Using MSAL, you write a token class (implementing the `TokenCredential` interface) exchange the ID token. The token class is used to with different client libraries to access Azure AD protected resources.

- [TypeScript](#)
- [C#](#)

The following TypeScript sample code snippet implements the `TokenCredential` interface, gets an ID token from Google (using the `getGoogleIDToken` method previously defined), and exchanges the ID token for an access token.

```

const msal = require("@azure/msal-node");
import {TokenCredential, GetTokenOptions, AccessToken} from "@azure/core-auth"

class ClientAssertionCredential implements TokenCredential {

    constructor(clientID:string, tenantID:string, aadAuthority:string) {
        this.clientID = clientID;
        this.tenantID = tenantID;
        this.aadAuthority = aadAuthority; // https://login.microsoftonline.com/
    }

    async getToken(scope: string | string[], _options?: GetTokenOptions):Promise<AccessToken> {

        var scopes:string[] = [];

        if (typeof scope === "string") {
            scopes[0]=scope;
        } else if (Array.isArray(scope)) {
            scopes = scope;
        }

        // Get the ID token from Google.
        return getGoogleIDToken() // calling this directly just for clarity,
                                // this should be a callback
        // pass this as a client assertion to the confidential client app
        .then((clientAssertion:any)=> {
            var msalApp: any;
            msalApp = new msal.ConfidentialClientApplication({
                auth: {
                    clientId: this.clientID,
                    authority: this.aadAuthority + this.tenantID,
                    clientAssertion: clientAssertion,
                }
            });
            return msalApp.acquireTokenByClientCredential({ scopes })
        })
        .then(function(aadToken) {
            // return in form expected by TokenCredential.getToken
            let returnToken = {
                token: aadToken.accessToken,
                expiresOnTimestamp: aadToken.expiresOn.getTime(),
            };
            return (returnToken);
        })
        .catch(function(error) {
            // error stuff
        });
    }
}

export default ClientAssertionCredential;

```

Access Azure AD protected resources

Your application running in Google Cloud now has an access token issued by Microsoft identity platform. Use the access token to access the Azure AD protected resources that your Azure AD app has permissions to access. As an example, here's how you can access Azure Blob storage using the `ClientAssertionCredential` token class and the Azure Blob Storage client library. When you make requests to the `BlobServiceClient` to access storage, the `BlobServiceClient` calls the `getToken` method on the `ClientAssertionCredential` object to get a fresh ID token and exchange it for an access token.

- [TypeScript](#)
- [C#](#)

The following TypeScript example initializes a new `ClientAssertionCredential` object and then creates a new `BlobServiceClient` object.

```
const { BlobServiceClient } = require("@azure/storage-blob");

var storageUrl = "https://<storageaccount>.blob.core.windows.net";
var clientID:any = "<client-id>";
var tenantID:any = "<tenant-id>";
var aadAuthority:any = "https://login.microsoftonline.com/";
var credential = new ClientAssertionCredential(clientID,
                                              tenantID,
                                              aadAuthority);

const blobServiceClient = new BlobServiceClient(storageUrl, credential);

// write code to access Blob storage
```

Next steps

Learn more about [workload identity federation](#).

Exchange a SAML token issued by AD FS for a Microsoft Graph access token

4/12/2022 • 3 minutes to read • [Edit Online](#)

To enable single sign-on (SSO) in applications that use SAML tokens issued by Active Directory Federation Services (AD FS) and also require access to Microsoft Graph, follow the steps in this article.

You'll enable the SAML bearer assertion flow to exchange a SAMLv1 token issued by the federated AD FS instance for an OAuth 2.0 access token for Microsoft Graph. When the user's browser is redirected to Azure Active Directory (Azure AD) to authenticate them, the browser picks up the session from the SAML sign-in instead of asking the user to enter their credentials.

IMPORTANT

This scenario works **only** when AD FS is the federated identity provider that issued the original SAMLv1 token. You **cannot** exchange a SAMLv2 token issued by Azure AD for a Microsoft Graph access token.

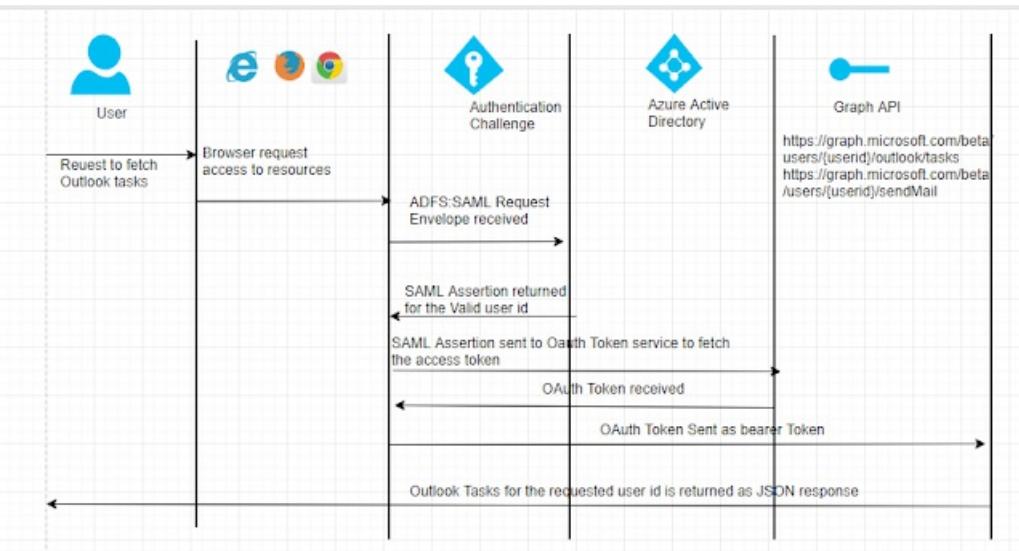
Prerequisites

- AD FS federated as an identity provider for single sign-on; see [Setting up AD FS and Enabling Single Sign-On to Office 365](#) for an example.
- [Postman](#) for testing requests.

Scenario overview

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.



Register the application with Azure AD

Start by registering the application in the [portal](#):

1. Sign in to the [app registration page of the portal](#) (Please note that we are using the v2.0 endpoints for Graph API and hence need to register the application in Azure portal. Otherwise we could have used the registrations in Azure AD).
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 page.
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.

Get the SAML assertion from AD FS

Create a POST request to the AD FS endpoint using SOAP envelope to fetch the SAML assertion:

ADFS Assertion Request

POST https://ADFSFQDN/adfs/services/trust/2005/usernamemixed

Send Save Examples (0)

Params	Authorization	Headers	Body	Pre-request Script	Tests	Cookies	Code
KEY			VALUE		DESCRIPTION	***	Bulk Edit
client-request-id			CLIENT_ID				
Key			Value		Description		

Header values:

ADFS Assertion Request

POST https://ADFSFQDN/adfs/services/trust/2005/usernamemixed

Headers (5)

KEY	VALUE	DESCRIPTION	***	Bulk Edit	Presets
<input checked="" type="checkbox"/> SOAPAction	http://schemas.xmlsoap.org/ws/2005/02/trust/RST/Issue				
<input checked="" type="checkbox"/> Content-Type	application/soap+xml				
<input checked="" type="checkbox"/> client-request-id	CLIENT_ID				
<input checked="" type="checkbox"/> return-client-request-id	true				
<input checked="" type="checkbox"/> Accept	application/json				
Key	Value	Description			

Response

AD FS request body:

POST https://ADFSFQDN/adfs/services/trust/2005/usernamemixed

Body

raw

```

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='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd'
4     <:Header><a:Action s:mustUnderstand='1' href='http://schemas.xmlsoap.org/ws/2005/02/trust/RST/Issue'/>
5     <a:MessageID>urn:uuid:90f330d3-1f9e-466c-9938-c94802822557</a:MessageID>
6     <a:ReplyTo><a:Address href='http://www.w3.org/2005/08/addressing/anonymous'/></a:ReplyTo>
7     <a:To>s:mustUnderstand='1' href='https://ADFSFQDN/adfs/services/trust/2005/usernamemixed/></a:To>
8     <o:UsernameToken u:Id='252585EF-60EA-4420-83BA-60E026F40069'>
9       <o:Username>USERNAME</o:Username>
10      <o>Password>PASSWORD</o:Password></o:UsernameToken>
11    </o:Security>
12  </s:Header>
13  <s:Body>
14    <trust:RequestSecurityToken xmlns:trust='http://schemas.xmlsoap.org/ws/2005/02/trust'>
15      <wsp:AppliesTo xmlns:wsp='http://schemas.xmlsoap.org/ws/2004/09/policy'>
16        <a:EndpointReference>
17          <a:Address>urn:federation:MicrosoftOnline</a:Address>
18        </a:EndpointReference>
19      </wsp:AppliesTo>
20      <trust:KeyType>http://schemas.xmlsoap.org/ws/2005/05/identity/NoProofKey</trust:KeyType>
21      <trust:RequestType>http://schemas.xmlsoap.org/ws/2005/02/trust/Issue</trust:RequestType>
22      <trust:RequestSecurityToken>
23    </s:Body>
24  </s:Envelope>

```

Once the request is posted successfully, you should receive a SAML assertion from AD FS. Only the **SAML:Assertion** tag data is required, convert it to base64 encoding to use in further requests.

Get the OAuth 2.0 token using the SAML assertion

Fetch an OAuth 2.0 token using the AD FS assertion response.

- Create a POST request as shown below with the header values:

POST https://login.microsoftonline.com/<TENANTID>/OAuth2/v2.0/token

Headers (2)

KEY	VALUE	DESCRIPTION	***	Bulk Edit	Presets
<input checked="" type="checkbox"/> Host	login.microsoftonline.com				
<input checked="" type="checkbox"/> Content-Type	application/x-www-form-urlencoded				
Key	Value	Description			

Response

- In the body of the request, replace **client_id**, **client_secret**, and **assertion** (the base64 encoded SAML assertion obtained the previous step):

POST https://login.microsoftonline.com/<TENANTID>/OAuth2/v2.0/token

Params Authorization Headers (2) **Body** Pre-request Script Tests Cookies Code

none form-data x-www-form-urlencoded raw binary

Key	Value	Description	...	Bulk Edit
grant_type	urn:sieft:params:oauth:grant-type:saml1_1-bearer		x	
client_id	CLIENTID		x	
client_secret	CLIENTSECRET		x	
assertion	ASSERTION		x	
scope	openid https://graph.microsoft.com/default		x	
Key	Value	Description		

3. Upon successful request, you'll receive an access token from Azure active directory.

Get the data with the OAuth 2.0 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
Content-Type	application/x-www-form-urlencoded		x		
Authorization	ACCESS TOKEN		x		
Key	Value	Description			

2. Upon successful request, you'll receive a JSON response.

Next steps

For more information about app registration and authentication flow, see:

- [Register an application with the Microsoft identity platform](#)
- [Authentication flows and application scenarios](#)

Using SCIM and Microsoft Graph together to provision users and enrich your application with the data it needs

4/12/2022 • 4 minutes to read • [Edit Online](#)

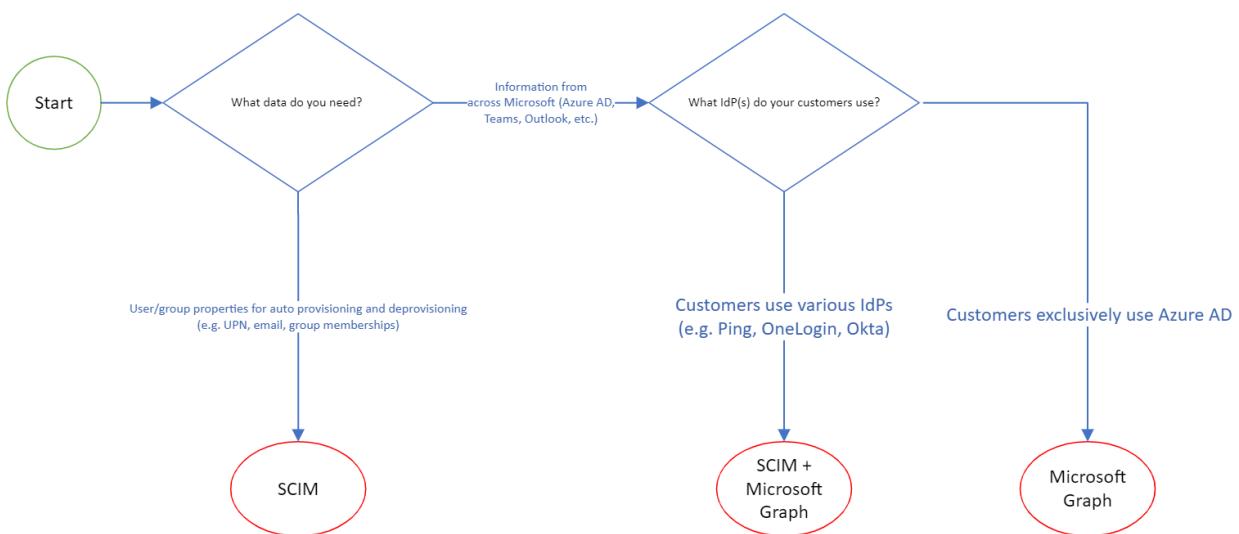
Target audience: This article is targeted towards developers building applications to be integrated with Azure Active Directory (Azure AD). If you're looking to use applications already integrated with Azure AD, such as Zoom, ServiceNow, and DropBox, you can skip this article and review the application specific [tutorials](#) or review [how the provisioning service works](#).

Common scenarios

Azure AD provides an out of the box service for provisioning and an extensible platform to build your applications on. The decision tree outlines how a developer would use [SCIM](#) and the [Microsoft Graph](#) to automate provisioning.

- Automatically create users in my application
- Automatically remove users from my application when they shouldn't have access anymore
- Integrate my application with multiple identity providers for provisioning
- Enrich my application with data from Microsoft services such as Teams, Outlook, and Office.
- Automatically create, update, and delete users and groups in Azure AD and Active Directory

Scenario: Automatically create, update, and delete users and groups in your application.



Scenario 1: Automatically create users in my app

Today, IT admins provision users by manually creating user accounts or periodically uploading CSV files into my application. The process is time consuming for customers and slows down adoption of my application. All I need is basic user information such as name, email, and userPrincipalName to create a user.

Recommendation:

- If your customers use various IdPs and you do not want to maintain a sync engine to integrate with each, support a SCIM compliant [/Users](#) endpoint. Your customers will be able to easily use this endpoint to integrate with the Azure AD provisioning service and automatically create user accounts when they need

access. You can build the endpoint once and it will be compatible with all IdPs. Check out the example request below for how a user would be created using SCIM.

- If you require user data found on the user object in Azure AD and other data from across Microsoft, consider building a SCIM endpoint for user provisioning and calling into the Microsoft Graph to get the rest of the data.

```
POST /Users
{
  "schemas": [
    "urn:ietf:params:scim:schemas:core:2.0:User",
    "urn:ietf:params:scim:schemas:extension:enterprise:2.0:User"],
  "externalId": "0a21f0f2-8d2a-4f8e-bf98-7363c4aed4ef",
  "userName": "BillG",
  "active": true,
  "meta": {
    "resourceType": "User"
  },
  "name": {
    "formatted": "Bill Gates",
    "familyName": "Gates",
    "givenName": "Bill"
  },
  "roles": []
}
```

Scenario 2: Automatically remove users from my app

The customers using my application are security focused and have governance requirements to remove accounts when employees don't need them anymore. How can I automate deprovisioning from my application?

Recommendation: Support a SCIM compliant /Users endpoint. The Azure AD provisioning service will send requests to disable and delete when the user shouldn't have access anymore. We recommend supporting both disabling and deleting users. See the examples below for what a disable and delete request look like.

Disable user

```
PATCH /Users/5171a35d82074e068ce2 HTTP/1.1
{
  "Operations": [
    {
      "op": "Replace",
      "path": "active",
      "value": false
    }
  ],
  "schemas": [
    "urn:ietf:params:scim:api:messages:2.0:PatchOp"
  ]
}
```

Delete user

```
DELETE /Users/5171a35d82074e068ce2 HTTP/1.1
```

Scenario 3: Automate managing group memberships in my app

My application relies on groups for access to various resources, and customers want to reuse the groups that they have in Azure AD. How can I import groups from Azure AD and keep them updated as the memberships

change?

Recommendation: Support a SCIM compliant /Groups endpoint. The Azure AD provisioning service will take care of creating groups and managing membership updates in your application.

Scenario 4: Enrich my app with data from Microsoft services such as Teams, Outlook, and OneDrive

My application is built into Microsoft Teams and relies on message data. In addition, we store files for users in OneDrive. How can I enrich my application with the data from these services and across Microsoft?

Recommendation: The [Microsoft Graph](#) is your entry point to access Microsoft data. Each workload exposes APIs with the data that you need. The Microsoft graph can be used along with [SCIM provisioning](#) for the scenarios above. You can use SCIM to provision basic user attributes into your application while calling into graph to get any other data that you need.

Scenario 5: Track changes in Microsoft services such as Teams, Outlook, and Azure AD

I need to be able to track changes to Teams and Outlook messages and react to them in real time. How can I get these changes pushed to my application?

Recommendation: The Microsoft Graph provides [change notifications](#) and [change tracking](#) for various resources. Note the following limitations of change notifications:

- If an event receiver acknowledges an event, but fails to act on it for any reason, the event may be lost.
- The order in which changes are received are not guaranteed to be chronological.
- Change notifications don't always contain the [resource data](#) For the reasons above, developers often use change notifications along with change tracking for synchronization scenarios.

Scenario 6: Provision users and groups in Azure AD

My application creates information about a user that customers need in Azure AD. This could be an HR application than manages hiring, a communications app that creates phone numbers for users, or some other app that generates data that would be valuable in Azure AD. How do I populate the user record in Azure AD with that data?

Recommendation The Microsoft graph exposes /Users and /Groups endpoints that you can integrate with today to provision users into Azure AD. Please note that Azure Active Directory doesn't support writing those users back into Active Directory.

NOTE

Microsoft has a provisioning service that pulls in data from HR applications such as Workday and SuccessFactors. These integrations are built and managed by Microsoft. For onboarding a new HR application to our service, you can request it on [UserVoice](#).

Related articles

- [Review the synchronization Microsoft Graph documentation](#)
- [Integrating a custom SCIM app with Azure AD](#)

Configure provisioning using Microsoft Graph APIs

4/12/2022 • 5 minutes to read • [Edit Online](#)

The Azure portal is a convenient way to configure provisioning for individual apps one at a time. But if you're creating several—or even hundreds—of instances of an application, it can be easier to automate app creation and configuration with the Microsoft Graph APIs. This article outlines how to automate provisioning configuration through APIs. This method is commonly used for applications like [Amazon Web Services](#).

Overview of steps for using Microsoft Graph APIs to automate provisioning configuration

STEP	DETAILS
Step 1. Create the gallery application	Sign-in to the API client Retrieve the gallery application template Create the gallery application
Step 2. Create provisioning job based on template	Retrieve the template for the provisioning connector Create the provisioning job
Step 3. Authorize access	Test the connection to the application Save the credentials
Step 4. Start provisioning job	Start the job
Step 5. Monitor provisioning	Check the status of the provisioning job Retrieve the provisioning logs

Step 1: Create the gallery application

Sign in to Microsoft Graph Explorer (recommended), Postman, or any other API client you use

1. Start [Microsoft Graph Explorer](#).
2. Select the "Sign-In with Microsoft" button and sign in using Azure AD global administrator or App Admin credentials.
3. Upon successful sign-in, you'll see the user account details in the left-hand pane.

Retrieve the gallery application template identifier

Applications in the Azure AD application gallery each have an [application template](#) that describes the metadata for that application. Using this template, you can create an instance of the application and service principal in your tenant for management. Retrieve the identifier of the application template for [AWS Single-Account Access](#) and from the response, record the value of the `id` property to use later in this tutorial.

Request

```
GET https://graph.microsoft.com/beta/applicationTemplates?$filter=displayName eq 'AWS Single-Account Access'
```

Response

```
HTTP/1.1 200 OK
Content-type: application/json

{
  "value": [
    {
      "id": "8b1025e4-1dd2-430b-a150-2ef79cd700f5",
      "displayName": "AWS Single-Account Access",
      "homePageUrl": "http://aws.amazon.com/",
      "supportedSingleSignOnModes": [
        "password",
        "saml",
        "external"
      ],
      "supportedProvisioningTypes": [
        "sync"
      ],
      "logoUrl": "https://az495088.vo.msecnd.net/app-logo/aws_215.png",
      "categories": [
        "developerServices"
      ],
      "publisher": "Amazon",
      "description": "Federate to a single AWS account and use SAML claims to authorize access to AWS IAM roles. If you have many AWS accounts, consider using the AWS Single Sign-On gallery application instead."
    }
  ]
}
```

Create the gallery application

Use the template ID retrieved for your application in the last step to [create an instance](#) of the application and service principal in your tenant.

Request

```
POST https://graph.microsoft.com/beta/applicationTemplates/{id}/instantiate
Content-type: application/json

{
  "displayName": "AWS Contoso"
}
```

Response

```
HTTP/1.1 201 OK
Content-type: application/json

{
    "application": {
        "objectId": "cbc071a6-0fa5-4859-8g55-e983ef63df63",
        "appId": "92653dd4-aa3a-3323-80cf-e8cfefcc8d5d",
        "applicationTemplateId": "8b1025e4-1dd2-430b-a150-2ef79cd700f5",
        "displayName": "AWS Contoso",
        "homepage": "https://signin.aws.amazon.com/saml?metadata=aws|ISV9.1|primary|z",
        "replyUrls": [
            "https://signin.aws.amazon.com/saml"
        ],
        "logoutUrl": null,
        "samlMetadataUrl": null,
    },
    "servicePrincipal": {
        "objectId": "f47a6776-bca7-4f2e-bc6c-eec59d058e3e",
        "appDisplayName": "AWS Contoso",
        "applicationTemplateId": "8b1025e4-1dd2-430b-a150-2ef79cd700f5",
        "appRoleAssignmentRequired": true,
        "displayName": "My custom name",
        "homepage": "https://signin.aws.amazon.com/saml?metadata=aws|ISV9.1|primary|z",
        "replyUrls": [
            "https://signin.aws.amazon.com/saml"
        ],
        "servicePrincipalNames": [
            "93653dd4-aa3a-4323-80cf-e8cfefcc8d7d"
        ],
        "tags": [
            "WindowsAzureActiveDirectoryIntegratedApp"
        ],
    }
}
```

Step 2: Create the provisioning job based on the template

Retrieve the template for the provisioning connector

Applications in the gallery that are enabled for provisioning have templates to streamline configuration. Use the request below to [retrieve the template for the provisioning configuration](#). Note that you will need to provide the ID. The ID refers to the preceding resource, which in this case is the servicePrincipal resource.

Request

```
GET https://graph.microsoft.com/beta/servicePrincipals/{id}/synchronization/templates
```

Response

```
HTTP/1.1 200 OK
```

```
{  
    "value": [  
        {  
            "id": "aws",  
            "factoryTag": "aws",  
            "schema": {  
                "directories": [],  
                "synchronizationRules": []  
            }  
        }  
    ]  
}
```

Create the provisioning job

To enable provisioning, you'll first need to [create a job](#). Use the following request to create a provisioning job. Use the templateId from the previous step when specifying the template to be used for the job.

Request

```
POST https://graph.microsoft.com/beta/servicePrincipals/{id}/synchronization/jobs  
Content-type: application/json  
  
{  
    "templateId": "aws"  
}
```

Response

```
HTTP/1.1 201 OK  
Content-type: application/json  
  
{  
    "id": "{jobId}",  
    "templateId": "aws",  
    "schedule": {  
        "expiration": null,  
        "interval": "P10675199DT2H48M5.4775807S",  
        "state": "Disabled"  
    },  
    "status": {  
        "countSuccessiveCompleteFailures": 0,  
        "escrowsPruned": false,  
        "synchronizedEntryCountByType": [],  
        "code": "NotConfigured",  
        "lastExecution": null,  
        "lastSuccessfulExecution": null,  
        "lastSuccessfulExecutionWithExports": null,  
        "steadyStateFirstAchievedTime": "0001-01-01T00:00:00Z",  
        "steadyStateLastAchievedTime": "0001-01-01T00:00:00Z",  
        "quarantine": null,  
        "troubleshootingUrl": null  
    }  
}
```

Step 3: Authorize access

Test the connection to the application

Test the connection with the third-party application. The following example is for an application that requires a client secret and secret token. Each application has its own requirements. Applications often use a base address

in place of a client secret. To determine what credentials your app requires, go to the provisioning configuration page for your application, and in developer mode, click **test connection**. The network traffic will show the parameters used for credentials. For a full list of credentials, see [synchronizationJob: validateCredentials](#). Most applications, such as Azure Databricks, rely on a BaseAddress and SecretToken. The BaseAddress is referred to as a tenant URL in the Azure portal.

Request

```
POST https://graph.microsoft.com/beta/servicePrincipals/{id}/synchronization/jobs/{id}/validateCredentials

{
  "credentials": [
    {
      "key": "ClientSecret", "value": "xxxxxxxxxxxxxxxxxxxxxx"
    },
    {
      "key": "SecretToken", "value": "xxxxxxxxxxxxxxxxxxxxxx"
    }
  ]
}
```

Response

```
HTTP/1.1 204 No Content
```

Save your credentials

Configuring provisioning requires establishing a trust between Azure AD and the application. Authorize access to the third-party application. The following example is for an application that requires a client secret and a secret token. Each application has its own requirements. Review the [API documentation](#) to see the available options.

Request

```
PUT https://graph.microsoft.com/beta/servicePrincipals/{id}/synchronization/secrets

{
  "value": [
    {
      "key": "ClientSecret", "value": "xxxxxxxxxxxxxxxxxxxxxx"
    },
    {
      "key": "SecretToken", "value": "xxxxxxxxxxxxxxxxxxxxxx"
    }
  ]
}
```

Response

```
HTTP/1.1 204 No Content
```

Step 4: Start the provisioning job

Now that the provisioning job is configured, use the following command to [start the job](#).

Request

```
POST https://graph.microsoft.com/beta/servicePrincipals/{id}/synchronization/jobs/{jobId}/start
```

Response

```
HTTP/1.1 204 No Content
```

Step 5: Monitor provisioning

Monitor the provisioning job status

Now that the provisioning job is running, use the following command to track the progress of the current provisioning cycle as well as statistics to date such as the number of users and groups that have been created in the target system.

Request

```
GET https://graph.microsoft.com/beta/servicePrincipals/{id}/synchronization/jobs/{jobId}/
```

Response

```
HTTP/1.1 200 OK
Content-type: application/json

{
  "id": "{jobId}",
  "templateId": "aws",
  "schedule": {
    "expiration": null,
    "interval": "P10675199DT2H48M5.4775807S",
    "state": "Disabled"
  },
  "status": {
    "countSuccessiveCompleteFailures": 0,
    "escrowsPruned": false,
    "synchronizedEntryCountByType": [],
    "code": "Paused",
    "lastExecution": null,
    "lastSuccessfulExecution": null,
    "progress": [],
    "lastSuccessfulExecutionWithExports": null,
    "steadyStateFirstAchievedTime": "0001-01-01T00:00:00Z",
    "steadyStateLastAchievedTime": "0001-01-01T00:00:00Z",
    "quarantine": null,
    "troubleshootingUrl": null
  },
  "synchronizationJobSettings": [
    {
      "name": "QuarantineTooManyDeletesThreshold",
      "value": "500"
    }
  ]
}
```

Monitor provisioning events using the provisioning logs

In addition to monitoring the status of the provisioning job, you can use the [provisioning logs](#) to query for all the events that are occurring. For example, query for a particular user and determine if they were successfully provisioned.

Request

```
GET https://graph.microsoft.com/beta/auditLogs/provisioning
```

Response

```
HTTP/1.1 200 OK
Content-type: application/json

{
    "@odata.context": "https://graph.microsoft.com/beta/$metadata#auditLogs/provisioning",
    "value": [
        {
            "id": "gc532ff9-r265-ec76-861e-42e2970a8218",
            "activityDateTime": "2019-06-24T20:53:08Z",
            "tenantId": "7928d5b5-7442-4a97-ne2d-66f9j9972ecn",
            "cycleId": "44576n58-v14b-70fj-8404-3d22tt46ed93",
            "changeId": "eaad2f8b-e6e3-409b-83bd-e4e2e57177d5",
            "action": "Create",
            "durationInMilliseconds": 2785,
            "sourceSystem": {
                "id": "0404601d-a9c0-4ec7-bbcd-02660120d8c9",
                "displayName": "Azure Active Directory",
                "details": {}
            },
            "targetSystem": {
                "id": "cd22f60b-5f2d-1adg-adb4-76ef31db996b",
                "displayName": "AWS Contoso",
                "details": {
                    "ApplicationId": "f2764360-e0ec-5676-711e-cd6fc0d4dd61",
                    "ServicePrincipalId": "chc46a42-966b-47d7-9774-576b1c8bd0b8",
                    "ServicePrincipalDisplayName": "AWS Contoso"
                }
            },
            "initiatedBy": {
                "id": "",
                "displayName": "Azure AD Provisioning Service",
                "initiatorType": "system"
            }
        ]
    ]
}
```

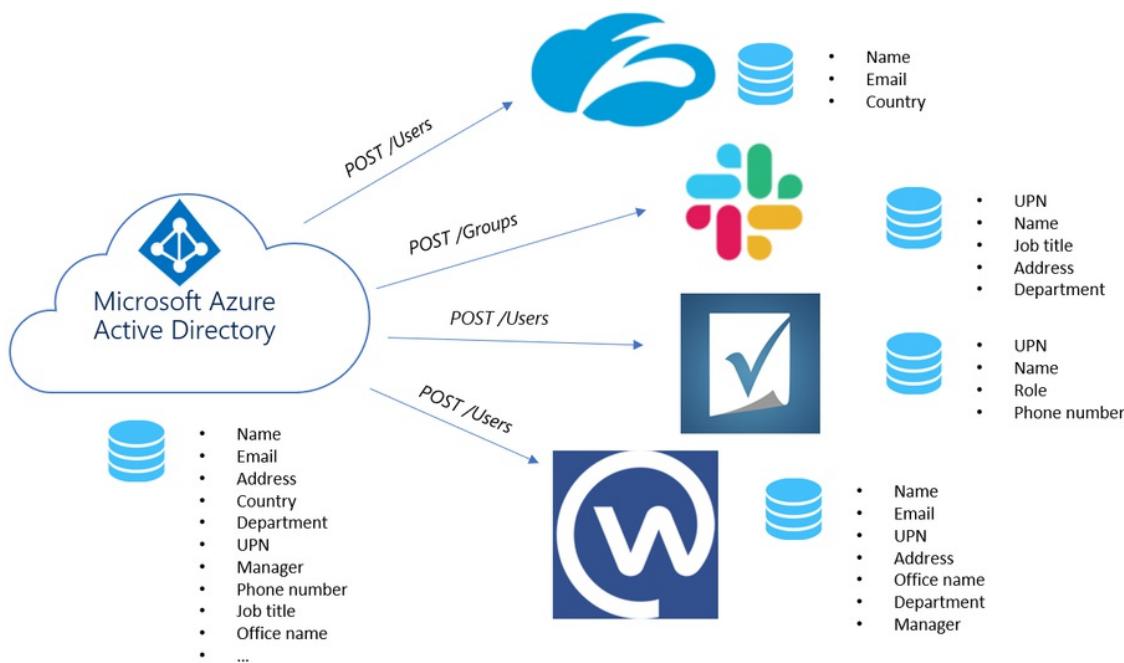
See also

- [Review the synchronization Microsoft Graph documentation](#)
- [Integrating a custom SCIM app with Azure AD](#)

Tutorial: Develop and plan provisioning for a SCIM endpoint in Azure Active Directory

4/12/2022 • 37 minutes to read • [Edit Online](#)

As an application developer, you can use the System for Cross-Domain Identity Management (SCIM) user management API to enable automatic provisioning of users and groups between your application and Azure AD (AAD). This article describes how to build a SCIM endpoint and integrate with the AAD provisioning service. The SCIM specification provides a common user schema for provisioning. When used in conjunction with federation standards like SAML or OpenID Connect, SCIM gives administrators an end-to-end, standards-based solution for access management.



SCIM is a standardized definition of two endpoints: a `/Users` endpoint and a `/Groups` endpoint. It uses common REST verbs to create, update, and delete objects, and a pre-defined schema for common attributes like group name, username, first name, last name and email. Apps that offer a SCIM 2.0 REST API can reduce or eliminate the pain of working with a proprietary user management API. For example, any compliant SCIM client knows how to make an HTTP POST of a JSON object to the `/Users` endpoint to create a new user entry. Instead of needing a slightly different API for the same basic actions, apps that conform to the SCIM standard can instantly take advantage of pre-existing clients, tools, and code.

The standard user object schema and rest APIs for management defined in SCIM 2.0 (RFC 7642, 7643, 7644) allow identity providers and apps to more easily integrate with each other. Application developers that build a SCIM endpoint can integrate with any SCIM-compliant client without having to do custom work.

To automate provisioning to an application will require building and integrating a SCIM endpoint with the Azure AD SCIM client. Use the following steps to start provisioning users and groups into your application.

1. Design your user and group schema

Identify the application's objects and attributes to determine how they map to the user and group schema supported by the AAD SCIM implementation.

2. Understand the AAD SCIM implementation

Understand how the AAD SCIM client is implemented to model your SCIM protocol request handling and responses.

3. Build a SCIM endpoint

An endpoint must be SCIM 2.0-compatible to integrate with the AAD provisioning service. As an option, use Microsoft Common Language Infrastructure (CLI) libraries and code samples to build your endpoint. These samples are for reference and testing only; we recommend against using them as dependencies in your production app.

4. Integrate your SCIM endpoint with the AAD SCIM client

If your organization uses a third-party application to implement a profile of SCIM 2.0 that AAD supports, you can quickly automate both provisioning and deprovisioning of users and groups.

5. Publish your application to the AAD application gallery

Make it easy for customers to discover your application and easily configure provisioning.



Design your user and group schema

Each application requires different attributes to create a user or group. Start your integration by identifying the required objects (users, groups) and attributes (name, manager, job title, etc.) that your application needs.

The SCIM standard defines a schema for managing users and groups.

The **core** user schema only requires three attributes (all other attributes are optional):

- `id`, service provider defined identifier
- `userName`, a unique identifier for the user (generally maps to the Azure AD user principal name)
- `meta`, *read-only* metadata maintained by the service provider

In addition to the **core** user schema, the SCIM standard defines an **enterprise** user extension with a model for extending the user schema to meet your application's needs.

For example, if your application requires both a user's email and user's manager, use the **core** schema to collect the user's email and the **enterprise** user schema to collect the user's manager.

To design your schema, follow these steps:

1. List the attributes your application requires, then categorize as attributes needed for authentication (e.g. `loginName` and `email`), attributes needed to manage the user lifecycle (e.g. `status / active`), and all other attributes needed for the application to work (e.g. `manager`, `tag`).
2. Check if the attributes are already defined in the **core** user schema or **enterprise** user schema. If not, you must define an extension to the user schema that covers the missing attributes. See example below for an extension to the user to allow provisioning a user `tag`.
3. Map SCIM attributes to the user attributes in Azure AD. If one of the attributes you have defined in your SCIM endpoint does not have a clear counterpart on the Azure AD user schema, guide the tenant administrator to extend their schema or use an extension attribute as shown below for the `tags`

property.

REQUIRED APP ATTRIBUTE	MAPPED SCIM ATTRIBUTE	MAPPED AZURE AD ATTRIBUTE
loginName	userNames	userPrincipalName
firstName	name.givenName	givenName
lastName	name.familyName	surName
workMail	emails[type eq "work"].value	Mail
manager	manager	manager
tag	urn:ietf:params:scim:schemas:extension:CustomExtensionName:2.0:User:tag	extensionAttribute1
status	active	isSoftDeleted (computed value not stored on user)

Example list of required attributes

```
{  
  "schemas": ["urn:ietf:params:scim:schemas:core:2.0:User",  
    "urn:ietf:params:scim:schemas:extension:enterprise:2.0:User",  
    "urn:ietf:params:scim:schemas:extension:CustomExtensionName:2.0:User"],  
  "userName": "bjensen@testuser.com",  
  "id": "48af03ac28ad4fb88478",  
  "externalId": "bjensen",  
  "name": {  
    "familyName": "Jensen",  
    "givenName": "Barbara"  
  },  
  "urn:ietf:params:scim:schemas:extension:enterprise:2.0:User": {  
    "Manager": "123456"  
  },  
  "urn:ietf:params:scim:schemas:extension:CustomExtensionName:2.0:User": {  
    "tag": "701984",  
  },  
  "meta": {  
    "resourceType": "User",  
    "created": "2010-01-23T04:56:22Z",  
    "lastModified": "2011-05-13T04:42:34Z",  
    "version": "W\\\"3694e05e9dff591\"",  
    "location":  
      "https://example.com/v2/Users/2819c223-7f76-453a-919d-413861904646"  
  }  
}
```

Example schema defined by a JSON payload

NOTE

In addition to the attributes required for the application, the JSON representation also includes the required `id`, `externalId`, and `meta` attributes.

It helps to categorize between `/User` and `/Group` to map any default user attributes in Azure AD to the SCIM RFC, see [how customize attributes are mapped between Azure AD and your SCIM endpoint](#).

AZURE ACTIVE DIRECTORY USER	"URN:IEFT:PARAMS:SCIM:SCHEMAS:EXTENSION:ENTERPRISE:2.0:USER"
IsSoftDeleted	active
department	urn:ietf:params:scim:schemas:extension:enterprise:2.0:User:department
displayName	displayName
employeeId	urn:ietf:params:scim:schemas:extension:enterprise:2.0:User:employeeNumber
Faxsimile-TelephoneNumber	phoneNumbers[type eq "fax"].value
givenName	name.givenName
jobTitle	title
mail	emails[type eq "work"].value
mailNickname	externalId
manager	urn:ietf:params:scim:schemas:extension:enterprise:2.0:User:manager
mobile	phoneNumbers[type eq "mobile"].value
postalCode	addresses[type eq "work"].postalCode
proxy-Addresses	emails[type eq "other"].Value
physical-Delivery-OfficeName	addresses[type eq "other"].Formatted
streetAddress	addresses[type eq "work"].streetAddress
surname	name.familyName
telephone-Number	phoneNumbers[type eq "work"].value
user-PrincipalName	userName

Example list of user and group attributes

AZURE ACTIVE DIRECTORY GROUP	URN:IEFT:PARAMS:SCIM:SCHEMAS:CORE:2.0:GROUP
displayName	displayName
members	members
objectId	externalId

Example list of group attributes

NOTE

You are not required to support both users and groups, or all the attributes shown here, it's only a reference on how attributes in Azure AD are often mapped to properties in the SCIM protocol.

There are several endpoints defined in the SCIM RFC. You can start with the `/User` endpoint and then expand from there.

ENDPOINT	DESCRIPTION
<code>/User</code>	Perform CRUD operations on a user object.
<code>/Group</code>	Perform CRUD operations on a group object.
<code>/Schemas</code>	The set of attributes supported by each client and service provider can vary. One service provider might include <code>name</code> , <code>title</code> , and <code>emails</code> , while another service provider uses <code>name</code> , <code>title</code> , and <code>phoneNumbers</code> . The schemas endpoint allows for discovery of the attributes supported.
<code>/Bulk</code>	Bulk operations allow you to perform operations on a large collection of resource objects in a single operation (e.g. update memberships for a large group).
<code>/ServiceProviderConfig</code>	Provides details about the features of the SCIM standard that are supported, for example the resources that are supported and the authentication method.
<code>/ResourceTypes</code>	Specifies metadata about each resource.

Example list of endpoints

NOTE

Use the `/Schemas` endpoint to support custom attributes or if your schema changes frequently as it enables a client to retrieve the most up-to-date schema automatically. Use the `/Bulk` endpoint to support groups.

Understand the AAD SCIM implementation

To support a SCIM 2.0 user management API, this section describes how the AAD SCIM client is implemented and shows how to model your SCIM protocol request handling and responses.

IMPORTANT

The behavior of the Azure AD SCIM implementation was last updated on December 18, 2018. For information on what changed, see [SCIM 2.0 protocol compliance of the Azure AD User Provisioning service](#).

Within the [SCIM 2.0 protocol specification](#), your application must support these requirements:

REQUIREMENT	REFERENCE NOTES (SCIM PROTOCOL)
Create users, and optionally also groups	section 3.3

REQUIREMENT	REFERENCE NOTES (SCIM PROTOCOL)
Modify users or groups with PATCH requests	section 3.5.2 . Supporting ensures that groups and users are provisioned in a performant manner.
Retrieve a known resource for a user or group created earlier	section 3.4.1
Query users or groups	section 3.4.2 . By default, users are retrieved by their <code>id</code> and queried by their <code>username</code> and <code>externalId</code> , and groups are queried by <code>displayName</code> .
The filter <code>excludedAttributes=members</code> when querying the group resource	section 3.4.2.5
Accept a single bearer token for authentication and authorization of AAD to your application.	
Soft-deleting a user <code>active=false</code> and restoring the user <code>active=true</code>	The user object should be returned in a request whether or not the user is active. The only time the user should not be returned is when it is hard deleted from the application.
Support the /Schemas endpoint	section 7 The schema discovery endpoint will be used to discover additional attributes.

Use the general guidelines when implementing a SCIM endpoint to ensure compatibility with AAD:

General:

- `id` is a required property for all resources. Every response that returns a resource should ensure each resource has this property, except for `ListResponse` with zero members.
- Values sent should be stored in the same format as what they were sent in. Invalid values should be rejected with a descriptive, actionable error message. Transformations of data should not happen between data being sent by Azure AD and data being stored in the SCIM application. (e.g. A phone number sent as 555555555555 should not be saved/returned as +5 (555) 555-5555)
- It isn't necessary to include the entire resource in the **PATCH** response.
- Don't require a case-sensitive match on structural elements in SCIM, in particular **PATCH op** operation values, as defined in [section 3.5.2](#). AAD emits the values of `op` as **Add**, **Replace**, and **Remove**.
- Microsoft AAD makes requests to fetch a random user and group to ensure that the endpoint and the credentials are valid. It's also done as a part of the **Test Connection** flow in the [Azure portal](#).
- Support HTTPS on your SCIM endpoint.
- Custom complex and multivalued attributes are supported but AAD does not have many complex data structures to pull data from in these cases. Simple paired name/value type complex attributes can be mapped to easily, but flowing data to complex attributes with three or more subattributes are not well supported at this time.
- The "type" sub-attribute values of multivalued complex attributes must be unique. For example, there can not be two different email addresses with the "work" sub-type.

Retrieving Resources:

- Response to a query/filter request should always be a `ListResponse`.
- Microsoft AAD only uses the following operators: `eq`, `and`
- The attribute that the resources can be queried on should be set as a matching attribute on the application in the [Azure portal](#), see [Customizing User Provisioning Attribute Mappings](#).

/Users:

- The `entitlements` attribute is not supported.

- Any attributes that are considered for user uniqueness must be usable as part of a filtered query. (e.g. if user uniqueness is evaluated for both `userName` and `emails[type eq "work"]`, a GET to `/Users` with a filter must allow for both `userName eq "user@contoso.com"` and `emails[type eq "work"].value eq "user@contoso.com"` queries.

/Groups:

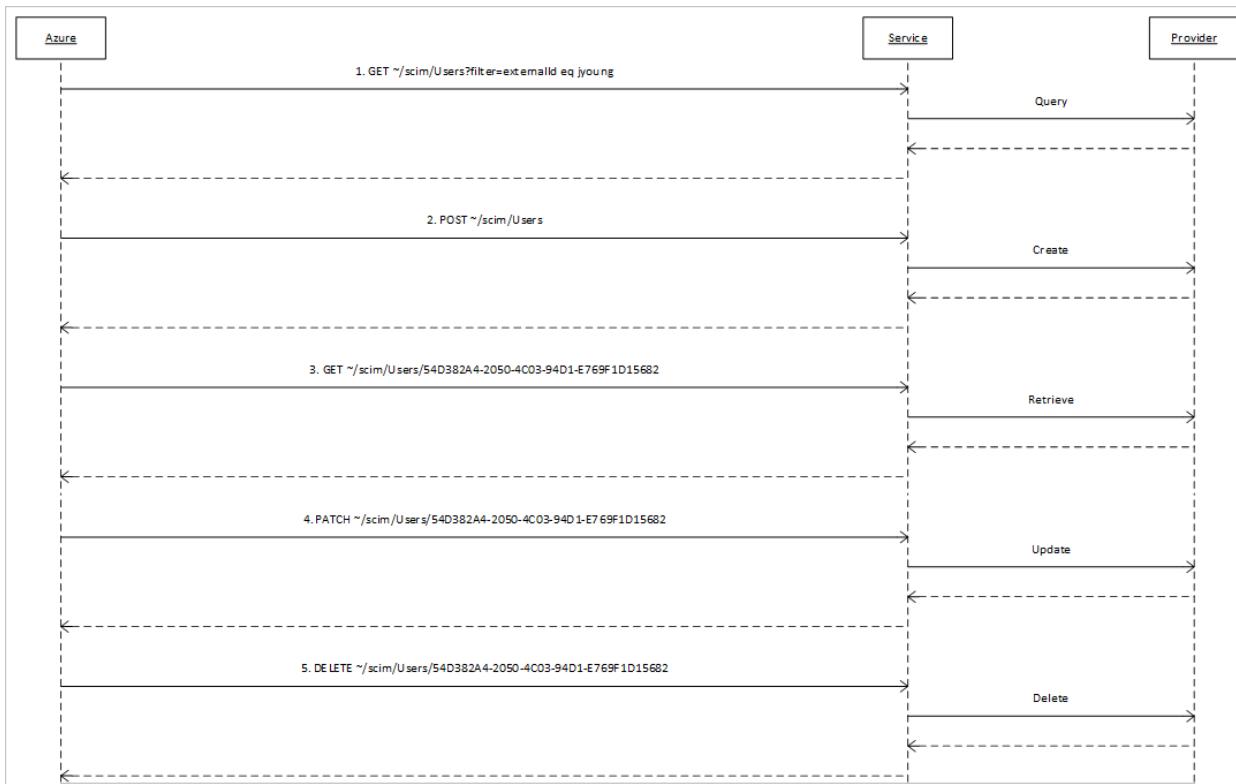
- Groups are optional, but only supported if the SCIM implementation supports **PATCH** requests.
- Groups must have uniqueness on the '`displayName`' value for the purpose of matching between Azure Active Directory and the SCIM application. This is not a requirement of the SCIM protocol, but is a requirement for integrating a SCIM service with Azure Active Directory.

/Schemas (Schema discovery):

- [Sample request/response](#)
- Schema discovery is not currently supported on the custom non-gallery SCIM application, but it is being used on certain gallery applications. Going forward, schema discovery will be used as the sole method to add additional attributes to the schema of an existing gallery SCIM application.
- If a value is not present, do not send null values.
- Property values should be camel cased (e.g. `readWrite`).
- Must return a list response.
- The `/schemas` request will be made by the Azure AD SCIM client every time someone saves the provisioning configuration in the Azure Portal or every time a user lands on the edit provisioning page in the Azure Portal. Any additional attributes discovered will be surfaced to customers in the attribute mappings under the target attribute list. Schema discovery only leads to additional target attributes being added. It will not result in attributes being removed.

User provisioning and deprovisioning

The following illustration shows the messages that AAD sends to a SCIM service to manage the lifecycle of a user in your application's identity store.



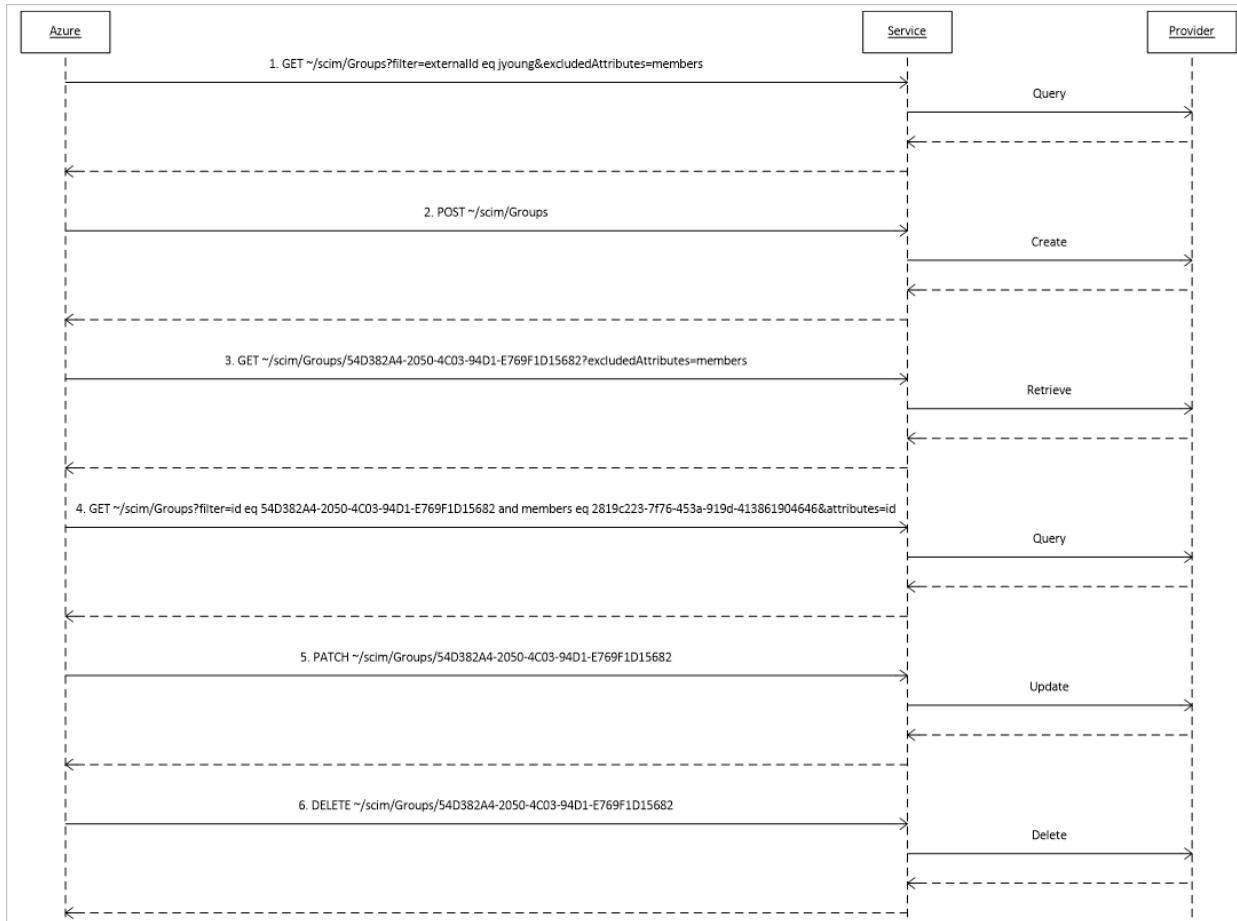
User provisioning and deprovisioning sequence

Group provisioning and deprovisioning

Group provisioning and deprovisioning are optional. When implemented and enabled, the following illustration shows the messages that AAD sends to a SCIM service to manage the lifecycle of a group in your application's

identity store. Those messages differ from the messages about users in two ways:

- Requests to retrieve groups specify that the members attribute is to be excluded from any resource provided in response to the request.
- Requests to determine whether a reference attribute has a certain value are requests about the members attribute.



Group provisioning and deprovisioning sequence

SCIM protocol requests and responses

This section provides example SCIM requests emitted by the AAD SCIM client and example expected responses. For best results, you should code your app to handle these requests in this format and emit the expected responses.

IMPORTANT

To understand how and when the AAD user provisioning service emits the operations described below, see the section [Provisioning cycles: Initial and incremental](#) in [How provisioning works](#).

User Operations

- [Create User \(Request / Response\)](#)
- [Get User \(Request / Response\)](#)
- [Get User by query \(Request / Response\)](#)
- [Get User by query - Zero results \(Request / Response\)](#)
- [Update User \[Multi-valued properties\] \(Request / Response\)](#)
- [Update User \[Single-valued properties\] \(Request / Response\)](#)
- [Disable User \(Request / Response\)](#)
- [Delete User \(Request / Response\)](#)

Group Operations

- [Create Group \(Request / Response\)](#)
- [Get Group \(Request / Response\)](#)
- [Get Group by displayName \(Request / Response\)](#)
- [Update Group \[Non-member attributes\] \(Request / Response\)](#)
- [Update Group \[Add Members\] \(Request / Response\)](#)
- [Update Group \[Remove Members\] \(Request / Response\)](#)
- [Delete Group \(Request / Response\)](#)

Schema discovery

- [Discover schema \(Request / Response\)](#)

User Operations

- Users can be queried by `userName` or `emails[type eq "work"]` attributes.

Create User

Request

POST /Users

```
{  
    "schemas": [  
        "urn:ietf:params:scim:schemas:core:2.0:User",  
        "urn:ietf:params:scim:schemas:extension:enterprise:2.0:User"],  
    "externalId": "0a21f0f2-8d2a-4f8e-bf98-7363c4aed4ef",  
    "userName": "Test_User_ab6490ee-1e48-479e-a20b-2d77186b5dd1",  
    "active": true,  
    "emails": [  
        {"primary": true,  
         "type": "work",  
         "value": "Test_User_fd0ea19b-0777-472c-9f96-4f70d2226f2e@testuser.com"}],  
    "meta": {  
        "resourceType": "User"  
    },  
    "name": {  
        "formatted": "givenName familyName",  
        "familyName": "familyName",  
        "givenName": "givenName"  
    },  
    "roles": []  
}
```

Response

HTTP/1.1 201 Created

```
{
  "schemas": ["urn:ietf:params:scim:schemas:core:2.0:User"],
  "id": "48af03ac28ad4fb88478",
  "externalId": "0a21f0f2-8d2a-4f8e-bf98-7363c4aed4ef",
  "meta": {
    "resourceType": "User",
    "created": "2018-03-27T19:59:26.000Z",
    "lastModified": "2018-03-27T19:59:26.000Z"
  },
  "userName": "Test_User_ab6490ee-1e48-479e-a20b-2d77186b5dd1",
  "name": {
    "formatted": "givenName familyName",
    "familyName": "familyName",
    "givenName": "givenName",
  },
  "active": true,
  "emails": [
    {
      "value": "Test_User_fd0ea19b-0777-472c-9f96-4f70d2226f2e@testuser.com",
      "type": "work",
      "primary": true
    }
  ]
}
```

Get User

Request

GET /Users/5d48a0a8e9f04aa38008

Response (User found)

HTTP/1.1 200 OK

```
{
  "schemas": ["urn:ietf:params:scim:schemas:core:2.0:User"],
  "id": "5d48a0a8e9f04aa38008",
  "externalId": "58342554-38d6-4ec8-948c-50044d0a33fd",
  "meta": {
    "resourceType": "User",
    "created": "2018-03-27T19:59:26.000Z",
    "lastModified": "2018-03-27T19:59:26.000Z"
  },
  "userName": "Test_User_feed3ace-693c-4e5a-82e2-694be1b39934",
  "name": {
    "formatted": "givenName familyName",
    "familyName": "familyName",
    "givenName": "givenName",
  },
  "active": true,
  "emails": [
    {
      "value": "Test_User_22370c1a-9012-42b2-bf64-86099c2a1c22@testuser.com",
      "type": "work",
      "primary": true
    }
  ]
}
```

Request

GET /Users/5171a35d82074e068ce2

Response (User not found. Note that the detail is not required, only status.)

```
{
  "schemas": [
    "urn:ietf:params:scim:api:messages:2.0:Error"
  ],
  "status": "404",
  "detail": "Resource 23B51B0E5D7AE9110A49411D@7cca31655d49f3640a494224 not found"
}
```

Get User by query

Request

```
GET /Users?filter=userName eq "Test_User_dfeef4c5-5681-4387-b016-bdf221e82081"
```

Response

HTTP/1.1 200 OK

```
{
  "schemas": ["urn:ietf:params:scim:api:messages:2.0>ListResponse"],
  "totalResults": 1,
  "Resources": [
    {
      "schemas": ["urn:ietf:params:scim:schemas:core:2.0:User"],
      "id": "2441309d85324e7793ae",
      "externalId": "7fce0092-d52e-4f76-b727-3955bd72c939",
      "meta": {
        "resourceType": "User",
        "created": "2018-03-27T19:59:26.000Z",
        "lastModified": "2018-03-27T19:59:26.000Z"
      },
      "userName": "Test_User_dfeef4c5-5681-4387-b016-bdf221e82081",
      "name": {
        "familyName": "familyName",
        "givenName": "givenName"
      },
      "active": true,
      "emails": [
        {
          "value": "Test_User_91b67701-697b-46de-b864-bd0bbe4f99c1@testuser.com",
          "type": "work",
          "primary": true
        }
      ],
      "startIndex": 1,
      "itemsPerPage": 20
    }
}
```

Get User by query - Zero results

Request

```
GET /Users?filter=userName eq "non-existent user"
```

Response

HTTP/1.1 200 OK

```
{
  "schemas": ["urn:ietf:params:scim:api:messages:2.0>ListResponse"],
  "totalResults": 0,
  "Resources": [],
  "startIndex": 1,
  "itemsPerPage": 20
}
```

Update User [Multi-valued properties]

Request

```
PATCH /Users/6764549bef60420686bc HTTP/1.1
```

```
{
  "schemas": ["urn:ietf:params:scim:api:messages:2.0:PatchOp"],
  "Operations": [
    {
      "op": "Replace",
      "path": "emails[type eq \"work\"].value",
      "value": "updatedEmail@microsoft.com"
    },
    {
      "op": "Replace",
      "path": "name.familyName",
      "value": "updatedFamilyName"
    }
  ]
}
```

Response

HTTP/1.1 200 OK

```
{
  "schemas": ["urn:ietf:params:scim:schemas:core:2.0:User"],
  "id": "6764549bef60420686bc",
  "externalId": "6c75de36-30fa-4d2d-a196-6bdedb6b6539",
  "meta": {
    "resourceType": "User",
    "created": "2018-03-27T19:59:26.000Z",
    "lastModified": "2018-03-27T19:59:26.000Z"
  },
  "userName": "Test_User_fbb9dda4-fcde-4f98-a68b-6c5599e17c27",
  "name": {
    "formatted": "givenName updatedFamilyName",
    "familyName": "updatedFamilyName",
    "givenName": "givenName"
  },
  "active": true,
  "emails": [
    {
      "value": "updatedEmail@microsoft.com",
      "type": "work",
      "primary": true
    }
  ]
}
```

Update User [Single-valued properties]

Request

PATCH /Users/5171a35d82074e068ce2 HTTP/1.1

```
{
  "schemas": ["urn:ietf:params:scim:api:messages:2.0:PatchOp"],
  "Operations": [
    {
      "op": "Replace",
      "path": "userName",
      "value": "5b50642d-79fc-4410-9e90-4c077cdd1a59@testuser.com"
    }
  ]
}
```

Response

HTTP/1.1 200 OK

```
{
  "schemas": ["urn:ietf:params:scim:schemas:core:2.0:User"],
  "id": "5171a35d82074e068ce2",
  "externalId": "aa1eca08-7179-4eeb-a0be-a519f7e5cd1a",
  "meta": {
    "resourceType": "User",
    "created": "2018-03-27T19:59:26.000Z",
    "lastModified": "2018-03-27T19:59:26.000Z"
  },
  "userName": "5b50642d-79fc-4410-9e90-4c077cdd1a59@testuser.com",
  "name": {
    "formatted": "givenName familyName",
    "familyName": "familyName",
    "givenName": "givenName",
  },
  "active": true,
  "emails": [
    {
      "value": "Test_User_49dc1090-aada-4657-8434-4995c25a00f7@testuser.com",
      "type": "work",
      "primary": true
    }
  ]
}
```

Disable User

Request

PATCH /Users/5171a35d82074e068ce2 HTTP/1.1

```
{
  "Operations": [
    {
      "op": "Replace",
      "path": "active",
      "value": false
    }
  ],
  "schemas": [
    "urn:ietf:params:scim:api:messages:2.0:PatchOp"
  ]
}
```

Response

```
{
  "schemas": [
    "urn:ietf:params:scim:schemas:core:2.0:User"
  ],
  "id": "CEC50F275D83C4530A495FCF@834d0e1e5d8235f90a495fda",
  "userName": "deanruiz@testuser.com",
  "name": {
    "familyName": "Harris",
    "givenName": "Larry"
  },
  "active": false,
  "emails": [
    {
      "value": "gloversuzanne@testuser.com",
      "type": "work",
      "primary": true
    }
  ],
  "addresses": [
    {
      "country": "ML",
      "type": "work",
      "primary": true
    }
  ],
  "meta": {
    "resourceType": "Users",
    "location": "/scim/5171a35d82074e068ce2/Users/CEC50F265D83B4530B495FCF@5171a35d82074e068ce2"
  }
}
```

Delete User

Request

DELETE /Users/5171a35d82074e068ce2 HTTP/1.1

Response

HTTP/1.1 204 No Content

Group Operations

- Groups shall always be created with an empty members list.
- Groups can be queried by the `displayName` attribute.
- Update to the group PATCH request should yield an *HTTP 204 No Content* in the response. Returning a body with a list of all the members isn't advisable.
- It isn't necessary to support returning all the members of the group.

Create Group

Request

POST /Groups HTTP/1.1

```
{
  "schemas": ["urn:ietf:params:scim:schemas:core:2.0:Group",
  "http://schemas.microsoft.com/2006/11/ResourceManagement/ADSCIM/2.0/Group"],
  "externalId": "8aa1a0c0-c4c3-4bc0-b4a5-2ef676900159",
  "displayName": "displayName",
  "meta": {
    "resourceType": "Group"
  }
}
```

Response

HTTP/1.1 201 Created

```
{
  "schemas": ["urn:ietf:params:scim:schemas:core:2.0:Group"],
  "id": "927fa2c08dcba4a7fae9e",
  "externalId": "8aa1a0c0-c4c3-4bc0-b4a5-2ef676900159",
  "meta": {
    "resourceType": "Group",
    "created": "2018-03-27T19:59:26.000Z",
    "lastModified": "2018-03-27T19:59:26.000Z"
  },
  "displayName": "displayName",
  "members": []
}
```

Get Group

Request

GET /Groups/40734ae655284ad3abcc?excludedAttributes=members HTTP/1.1

Response

HTTP/1.1 200 OK

```
{
  "schemas": ["urn:ietf:params:scim:schemas:core:2.0:Group"],
  "id": "40734ae655284ad3abcc",
  "externalId": "60f1bb27-2e1e-402d-bcc4-ec999564a194",
  "meta": {
    "resourceType": "Group",
    "created": "2018-03-27T19:59:26.000Z",
    "lastModified": "2018-03-27T19:59:26.000Z"
  },
  "displayName": "displayName",
}
```

Get Group by displayName

Request

GET /Groups?excludedAttributes=members&filter=displayName eq "displayName" HTTP/1.1

Response

HTTP/1.1 200 OK

```
{
  "schemas": ["urn:ietf:params:scim:api:messages:2.0>ListResponse"],
  "totalResults": 1,
  "Resources": [
    {
      "schemas": ["urn:ietf:params:scim:schemas:core:2.0:Group"],
      "id": "8c601452cc934a9ebef9",
      "externalId": "0db508eb-91e2-46e4-809c-30dcda0c685",
      "meta": {
        "resourceType": "Group",
        "created": "2018-03-27T22:02:32.000Z",
        "lastModified": "2018-03-27T22:02:32.000Z",
      },
      "displayName": "displayName",
    }
  ],
  "startIndex": 1,
  "itemsPerPage": 20
}
```

Update Group [Non-member attributes]

Request

PATCH /Groups/fa2ce26709934589afc5 HTTP/1.1

```
{  
    "schemas": ["urn:ietf:params:scim:api:messages:2.0:PatchOp"],  
    "Operations": [  
        {"op": "Replace",  
         "path": "displayName",  
         "value": "1879db59-3bdf-4490-ad68-ab880a269474updatedDisplayName"  
    ]  
}
```

Response

HTTP/1.1 204 No Content

Update Group [Add Members]

Request

PATCH /Groups/a99962b9f99d4c4fac67 HTTP/1.1

```
{  
    "schemas": ["urn:ietf:params:scim:api:messages:2.0:PatchOp"],  
    "Operations": [  
        {"op": "Add",  
         "path": "members",  
         "value": [{"  
             "$ref": null,  
             "value": "f648f8d5ea4e4cd38e9c"  
         }]  
    ]  
}
```

Response

HTTP/1.1 204 No Content

Update Group [Remove Members]

Request

PATCH /Groups/a99962b9f99d4c4fac67 HTTP/1.1

```
{  
    "schemas": ["urn:ietf:params:scim:api:messages:2.0:PatchOp"],  
    "Operations": [  
        {"op": "Remove",  
         "path": "members",  
         "value": [{"  
             "$ref": null,  
             "value": "f648f8d5ea4e4cd38e9c"  
         }]  
    ]  
}
```

Response

HTTP/1.1 204 No Content

Delete Group

Request

DELETE /Groups/cdb1ce18f65944079d37 HTTP/1.1

Response

HTTP/1.1 204 No Content

Schema discovery

Discover schema

Request

GET /Schemas

Response

```
{
  "schemas": [
    "urn:ietf:params:scim:api:messages:2.0>ListResponse"
  ],
  "itemsPerPage": 50,
  "startIndex": 1,
  "totalResults": 3,
  "Resources": [
    {
      "schemas": ["urn:ietf:params:scim:schemas:core:2.0:Schema"],
      "id": "urn:ietf:params:scim:schemas:core:2.0:User",
      "name": "User",
      "description": "User Account",
      "attributes": [
        {
          "name": "userName",
          "type": "string",
          "multiValued": false,
          "description": "Unique identifier for the User, typically used by the user to directly authenticate to the service provider. Each User MUST include a non-empty userName value. This identifier MUST be unique across the service provider's entire set of Users. REQUIRED.",
          "required": true,
          "caseExact": false,
          "mutability": "readWrite",
          "returned": "default",
          "uniqueness": "server"
        },
      ],
      "meta": {
        "resourceType": "Schema",
        "location": "/v2/Schemas/urn:ietf:params:scim:schemas:core:2.0:User"
      }
    },
    {
      "schemas": ["urn:ietf:params:scim:schemas:core:2.0:Schema"],
      "id": "urn:ietf:params:scim:schemas:core:2.0:Group",
      "name": "Group",
      "description": "Group",
      "attributes": [
        {
          "name": "displayName",
          "type": "string",
          "multiValued": false,
          "description": "A human-readable name for the Group. REQUIRED.",
          "required": false,
          "caseExact": false,
          "mutability": "readWrite",
          "returned": "default",
          "uniqueness": "none"
        },
      ],
      "meta": {
        "resourceType": "Schema",
        "location": "/v2/Schemas/urn:ietf:params:scim:schemas:core:2.0:Group"
      }
    },
    {
      "schemas": ["urn:ietf:params:scim:schemas:core:2.0:Schema"],
      "id": "urn:ietf:params:scim:schemas:extension:enterprise:2.0:User",
      "name": "EnterpriseUser",
      "description": "Enterprise User",
      "attributes": [
        {
          "name": "extensionAttribute1",
          "type": "string",
          "multiValued": false,
          "description": "A string attribute defined by the Enterprise extension. REQUIRED."
        }
      ],
      "meta": {
        "resourceType": "Schema",
        "location": "/v2/Schemas/urn:ietf:params:scim:schemas:extension:enterprise:2.0:User"
      }
    }
  ]
}
```

```

    "attributes" : [
        {
            "name" : "employeeNumber",
            "type" : "string",
            "multiValued" : false,
            "description" : "Numeric or alphanumeric identifier assigned to a person, typically based on order of hire or association with an organization.",
            "required" : false,
            "caseExact" : false,
            "mutability" : "readWrite",
            "returned" : "default",
            "uniqueness" : "none"
        },
    ],
    "meta" : {
        "resourceType" : "Schema",
        "location" :
        "/v2/Schemas/urn:ietf:params:scim:schemas:extension:enterprise:2.0:User"
    }
}
]
}

```

Security requirements

TLS Protocol Versions

The only acceptable TLS protocol versions are TLS 1.2 and TLS 1.3. No other versions of TLS are permitted. No version of SSL is permitted.

- RSA keys must be at least 2,048 bits.
- ECC keys must be at least 256 bits, generated using an approved elliptic curve

Key Lengths

All services must use X.509 certificates generated using cryptographic keys of sufficient length, meaning:

Cipher Suites

All services must be configured to use the following cipher suites, in the exact order specified below. Note that if you only have an RSA certificate, installed the ECDSA cipher suites do not have any effect.

TLS 1.2 Cipher Suites minimum bar:

- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384

IP Ranges

The Azure AD provisioning service currently operates under the IP Ranges for AzureActiveDirectory as listed [here](#). You can add the IP ranges listed under the AzureActiveDirectory tag to allow traffic from the Azure AD provisioning service into your application. Note that you will need to review the IP range list carefully for computed addresses. An address such as '40.126.25.32' could be represented in the IP range list as '40.126.0.0/18'. You can also programmatically retrieve the IP range list using the following [API](#).

Azure AD also supports an agent based solution to provide connectivity to applications in private networks (on-premises, hosted in Azure, hosted in AWS, etc.). Customers can deploy a lightweight agent, which provides connectivity to Azure AD without opening an inbound ports, on a server in their private network. Learn more [here](#).

Build a SCIM endpoint

Now that you have designed your schema and understood the Azure AD SCIM implementation, you can get started developing your SCIM endpoint. Rather than starting from scratch and building the implementation completely on your own, you can rely on a number of open source SCIM libraries published by the SCIM community.

For guidance on how to build a SCIM endpoint including examples, see [Develop a sample SCIM endpoint](#).

The open source .NET Core [reference code example](#) published by the Azure AD provisioning team is one such resource that can jump start your development. Once you have built your SCIM endpoint, you will want to test it out. You can use the collection of [postman tests](#) provided as part of the reference code or run through the sample requests / responses provided [above](#).

NOTE

The reference code is intended to help you get started building your SCIM endpoint and is provided "AS IS." Contributions from the community are welcome to help build and maintain the code.

The solution is composed of two projects, *Microsoft.SCIM* and *Microsoft.SCIM.WebHostSample*.

The *Microsoft.SCIM* project is the library that defines the components of the web service that conforms to the SCIM specification. It declares the interface *Microsoft.SCIM.IProvider*, requests are translated into calls to the provider's methods, which would be programmed to operate on an identity store.

```
«interface»
Microsoft.SystemForCrossDomainIdentityManagement::IProvider
+getStartupBehavior()
+Create()
+Delete()
+Query()
+Retrieve()
+Update()
```

The *Microsoft.SCIM.WebHostSample* project is a Visual Studio ASP.NET Core Web Application, based on the *Empty* template. This allows the sample code to be deployed as standalone, hosted in containers or within Internet Information Services. It also implements the *Microsoft.SCIM.IProvider* interface keeping classes in memory as a sample identity store.

```

public class Startup
{
    ...
    public IMonitor MonitoringBehavior { get; set; }
    public IProvider ProviderBehavior { get; set; }

    public Startup(IWebHostEnvironment env, IConfiguration configuration)
    {
        ...
        this.MonitoringBehavior = new ConsoleMonitor();
        this.ProviderBehavior = new InMemoryProvider();
    }
    ...
}

```

Building a custom SCIM endpoint

The SCIM service must have an HTTP address and server authentication certificate of which the root certification authority is one of the following names:

- CNNIC
- Comodo
- CyberTrust
- DigiCert
- GeoTrust
- GlobalSign
- Go Daddy
- VeriSign
- WoSign
- DST Root CA X3

The .NET Core SDK includes an HTTPS development certificate that can be used during development, the certificate is installed as part of the first-run experience. Depending on how you run the ASP.NET Core Web Application it will listen to a different port:

- Microsoft.SCIM.WebHostSample: <https://localhost:5001>
- IIS Express: <https://localhost:44359/>

For more information on HTTPS in ASP.NET Core use the following link: [Enforce HTTPS in ASP.NET Core](#)

Handling endpoint authentication

Requests from Azure Active Directory include an OAuth 2.0 bearer token. Any service receiving the request should authenticate the issuer as being Azure Active Directory for the expected Azure Active Directory tenant.

In the token, the issuer is identified by an iss claim, like

`"iss": "https://sts.windows.net/cbb1a5ac-f33b-45fa-9bf5-f37db0fed422/"`. In this example, the base address of the claim value, `https://sts.windows.net`, identifies Azure Active Directory as the issuer, while the relative address segment, `cbb1a5ac-f33b-45fa-9bf5-f37db0fed422`, is a unique identifier of the Azure Active Directory tenant for which the token was issued.

The audience for the token will be the application template ID for the application in the gallery, each of the applications registered in a single tenant may receive the same `iss` claim with SCIM requests. The application template ID for all custom apps is `8adf8e6e-67b2-4cf2-a259-e3dc5476c621`. The token generated by the Azure AD provisioning service should only be used for testing. It should not be used in production environments.

In the sample code, requests are authenticated using the `Microsoft.AspNetCore.Authentication.JwtBearer` package. The following code enforces that requests to any of the service's endpoints are authenticated using the bearer token issued by Azure Active Directory for a specified tenant:

```

public void ConfigureServices(IServiceCollection services)
{
    if (_env.IsDevelopment())
    {
        ...
    }
    else
    {
        services.AddAuthentication(options =>
        {
            options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
            options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
        })
        .AddJwtBearer(options =>
        {
            options.Authority = " https://sts.windows.net/cbb1a5ac-f33b-45fa-9bf5-
f37db0fed422/";
            options.Audience = "8adf8e6e-67b2-4cf2-a259-e3dc5476c621";
            ...
        });
    }
    ...
}

public void Configure(IApplicationBuilder app)
{
    ...
    app.UseAuthentication();
    app.UseAuthorization();
    ...
}

```

A bearer token is also required to use of the provided [postman tests](#) and perform local debugging using localhost. The sample code uses ASP.NET Core environments to change the authentication options during development stage and enable the use a self-signed token.

For more information on multiple environments in ASP.NET Core, see [Use multiple environments in ASP.NET Core](#).

The following code enforces that requests to any of the service's endpoints are authenticated using a bearer token signed with a custom key:

```

public void ConfigureServices(IServiceCollection services)
{
    if (_env.IsDevelopment())
    {
        services.AddAuthentication(options =>
        {
            options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
            options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
        })
        .AddJwtBearer(options =>
        {
            options.TokenValidationParameters =
                new TokenValidationParameters
                {
                    ValidateIssuer = false,
                    ValidateAudience = false,
                    ValidateLifetime = false,
                    ValidateIssuerSigningKey = false,
                    ValidIssuer = "Microsoft.Security.Bearer",
                    ValidAudience = "Microsoft.Security.Bearer",
                    IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("A1B2C3D4E5F6A1B2C3D4E5F6"))
                };
        });
    }
    ...
}

```

Send a GET request to the Token controller to get a valid bearer token, the method *GenerateJSONWebToken* is responsible to create a token matching the parameters configured for development:

```

private string GenerateJSONWebToken()
{
    // Create token key
    SymmetricSecurityKey securityKey =
        new SymmetricSecurityKey(Encoding.UTF8.GetBytes("A1B2C3D4E5F6A1B2C3D4E5F6"));
    SigningCredentials credentials =
        new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

    // Set token expiration
    DateTime startTime = DateTime.UtcNow;
    DateTime expiryTime = startTime.AddMinutes(120);

    // Generate the token
    JwtSecurityToken token =
        new JwtSecurityToken(
            "Microsoft.Security.Bearer",
            "Microsoft.Security.Bearer",
            null,
            notBefore: startTime,
            expires: expiryTime,
            signingCredentials: credentials);

    string result = new JwtSecurityTokenHandler().WriteToken(token);
    return result;
}

```

Handling provisioning and deprovisioning of users

Example 1. Query the service for a matching user

Azure Active Directory (AAD) queries the service for a user with an `externalId` attribute value matching the `mailNickname` attribute value of a user in AAD. The query is expressed as a Hypertext Transfer Protocol (HTTP) request such as this example, wherein `jyoung` is a sample of a `mailNickname` of a user in Azure Active Directory.

NOTE

This is an example only. Not all users will have a mailNickname attribute, and the value a user has may not be unique in the directory. Also, the attribute used for matching (which in this case is `externalId`) is configurable in the [AAD attribute mappings](#).

```
GET https://.../scim/Users?filter=externalId eq jyoung HTTP/1.1
Authorization: Bearer ...
```

In the sample code the request is translated into a call to the `QueryAsync` method of the service's provider. Here is the signature of that method:

```
// System.Threading.Tasks.Tasks is defined in mscorelib.dll.
// Microsoft.SCIM.IRequest is defined in
// Microsoft.SCIM.Service.
// Microsoft.SCIM.Resource is defined in
// Microsoft.SCIM.Schemas.
// Microsoft.SCIM.IQueryParameters is defined in
// Microsoft.SCIM.Protocol.

Task<Resource[]> QueryAsync(IRequest<IQueryParameters> request);
```

In the sample query, for a user with a given value for the `externalId` attribute, values of the arguments passed to the `QueryAsync` method are:

- `parameters.AlternateFilters.Count: 1`
- `parameters.AlternateFilters.ElementAt(0).AttributePath: "externalId"`
- `parameters.AlternateFilters.ElementAt(0).ComparisonOperator: ComparisonOperator.Equals`
- `parameters.AlternateFilter.ElementAt(0).ComparisonValue: "jyoung"`

Example 2. Provision a user

If the response to a query to the web service for a user with an `externalId` attribute value that matches the `mailNickname` attribute value of a user doesn't return any users, then AAD requests that the service provision a user corresponding to the one in AAD. Here is an example of such a request:

```

POST https://.../scim/Users HTTP/1.1
Authorization: Bearer ...
Content-type: application/scim+json
{
  "schemas": [
    "urn:ietf:params:scim:schemas:core:2.0:User",
    "urn:ietf:params:scim:schemas:extension:enterprise:2.0User"],
  "externalId": "jyoung",
  "userName": "jyoung@testuser.com",
  "active": true,
  "addresses": null,
  "displayName": "Joy Young",
  "emails": [
    {
      "type": "work",
      "value": "jyoung@Contoso.com",
      "primary": true}],
  "meta": {
    "resourceType": "User",
    "name": {
      "familyName": "Young",
      "givenName": "Joy"},
    "phoneNumbers": null,
    "preferredLanguage": null,
    "title": null,
    "department": null,
    "manager": null}
}

```

In the sample code the request is translated into a call to the CreateAsync method of the service's provider. Here is the signature of that method:

```

// System.Threading.Tasks.Tasks is defined in mscorlib.dll.
// Microsoft.SCIM.IRequest is defined in
// Microsoft.SCIM.Service.
// Microsoft.SCIM.Resource is defined in
// Microsoft.SCIM.Schemas.

Task<Resource> CreateAsync(IRequest<Resource> request);

```

In a request to provision a user, the value of the resource argument is an instance of the Microsoft.SCIM.Core2EnterpriseUser class, defined in the Microsoft.SCIM.Schemas library. If the request to provision the user succeeds, then the implementation of the method is expected to return an instance of the Microsoft.SCIM.Core2EnterpriseUser class, with the value of the Identifier property set to the unique identifier of the newly provisioned user.

Example 3. Query the current state of a user

To update a user known to exist in an identity store fronted by an SCIM, Azure Active Directory proceeds by requesting the current state of that user from the service with a request such as:

```

GET ~/scim/Users/54D382A4-2050-4C03-94D1-E769F1D15682 HTTP/1.1
Authorization: Bearer ...

```

In the sample code the request is translated into a call to the RetrieveAsync method of the service's provider. Here is the signature of that method:

```

// System.Threading.Tasks.Tasks is defined in mscorlib.dll.
// Microsoft.SCIM.IRequest is defined in
// Microsoft.SCIM.Service.
// Microsoft.SCIM.Resource and
// Microsoft.SCIM.IResourceRetrievalParameters
// are defined in Microsoft.SCIM.Schemas

Task<Resource> RetrieveAsync(IRequest<IResourceRetrievalParameters> request);

```

In the example of a request to retrieve the current state of a user, the values of the properties of the object provided as the value of the parameters argument are as follows:

- Identifier: "54D382A4-2050-4C03-94D1-E769F1D15682"
- SchemaIdentifier: "urn:ietf:params:scim:schemas:extension:enterprise:2.0:User"

Example 4. Query the value of a reference attribute to be updated

If a reference attribute is to be updated, then Azure Active Directory queries the service to determine whether the current value of the reference attribute in the identity store fronted by the service already matches the value of that attribute in Azure Active Directory. For users, the only attribute of which the current value is queried in this way is the manager attribute. Here is an example of a request to determine whether the manager attribute of a user object currently has a certain value: In the sample code the request is translated into a call to the QueryAsync method of the service's provider. The value of the properties of the object provided as the value of the parameters argument are as follows:

- parameters.AlternateFilters.Count: 2
- parameters.AlternateFilters.ElementAt(x).AttributePath: "ID"
- parameters.AlternateFilters.ElementAt(x).ComparisonOperator: ComparisonOperator.Equals
- parameters.AlternateFilter.ElementAt(x).ComparisonValue: "54D382A4-2050-4C03-94D1-E769F1D15682"
- parameters.AlternateFilters.ElementAt(y).AttributePath: "manager"
- parameters.AlternateFilters.ElementAt(y).ComparisonOperator: ComparisonOperator.Equals
- parameters.AlternateFilter.ElementAt(y).ComparisonValue: "2819c223-7f76-453a-919d-413861904646"
- parameters.RequestedAttributePaths.ElementAt(0): "ID"
- parameters.SchemaIdentifier: "urn:ietf:params:scim:schemas:extension:enterprise:2.0:User"

Here, the value of the index x can be 0 and the value of the index y can be 1, or the value of x can be 1 and the value of y can be 0, depending on the order of the expressions of the filter query parameter.

Example 5. Request from Azure AD to an SCIM service to update a user

Here is an example of a request from Azure Active Directory to an SCIM service to update a user:

```

PATCH ~/scim/Users/54D382A4-2050-4C03-94D1-E769F1D15682 HTTP/1.1
Authorization: Bearer ...
Content-type: application/scim+json
{
  "schemas": [
    [
      "urn:ietf:params:scim:api:messages:2.0:PatchOp"
    ],
    "Operations": [
      [
        {
          "op": "Add",
          "path": "manager",
          "value": [
            {
              "$ref": "http://.../scim/Users/2819c223-7f76-453a-919d-413861904646",
              "value": "2819c223-7f76-453a-919d-413861904646"
            }
          ]
        }
      ]
    ]
  ]
}

```

In the sample code the request is translated into a call to the `UpdateAsync` method of the service's provider. Here is the signature of that method:

```

// System.Threading.Tasks.Tasks and
// System.Collections.Generic.IReadOnlyCollection<T> // are defined in mscorlib.dll.
// Microsoft.SCIM.IRequest is defined in
// Microsoft.SCIM.Service.
// Microsoft.SCIM.IPatch,
// is defined in Microsoft.SCIM.Protocol.

Task UpdateAsync(IRequest<IPatch> request);

```

In the example of a request to update a user, the object provided as the value of the `patch` argument has these property values:

ARGUMENT	VALUE
ResourceIdentifier.Identifier	"54D382A4-2050-4C03-94D1-E769F1D15682"
ResourceIdentifier.SchemaIdentifier	"urn:ietf:params:scim:schemas:extension:enterprise:2.0:User"
(PatchRequest as PatchRequest2).Operations.Count	1
(PatchRequest as PatchRequest2).Operations.ElementAt(0).OperationName	OperationName.Add
(PatchRequest as PatchRequest2).Operations.ElementAt(0).Path.AttributePath	"manager"
(PatchRequest as PatchRequest2).Operations.ElementAt(0).Value.Count	1
(PatchRequest as PatchRequest2).Operations.ElementAt(0).Value.ElementAt(0).Reference	http://.../scim/Users/2819c223-7f76-453a-919d-413861904646
(PatchRequest as PatchRequest2).Operations.ElementAt(0).Value.ElementAt(0).Value	2819c223-7f76-453a-919d-413861904646

Example 6. Deprovision a user

To deprovision a user from an identity store fronted by an SCIM service, AAD sends a request such as:

```
DELETE ~/scim/Users/54D382A4-2050-4C03-94D1-E769F1D15682 HTTP/1.1  
Authorization: Bearer ...
```

In the sample code the request is translated into a call to the `DeleteAsync` method of the service's provider. Here is the signature of that method:

```
// System.Threading.Tasks.Tasks is defined in mscorlib.dll.  
// Microsoft.SCIM.IRequest is defined in  
// Microsoft.SCIM.Service.  
// Microsoft.SCIM.IResourceIdentifier,  
// is defined in Microsoft.SCIM.Protocol.  
  
Task DeleteAsync(IRequest<IResourceIdentifier> request);
```

The object provided as the value of the `resourceIdentifier` argument has these property values in the example of a request to deprovision a user:

- `ResourceIdentifier.Identifier`: "54D382A4-2050-4C03-94D1-E769F1D15682"
- `ResourceIdentifier.SchemaIdentifier`: "urn:ietf:params:scim:schemas:extension:enterprise:2.0:User"

Integrate your SCIM endpoint with the AAD SCIM client

Azure AD can be configured to automatically provision assigned users and groups to applications that implement a specific profile of the [SCIM 2.0 protocol](#). The specifics of the profile are documented in [Understand the Azure AD SCIM implementation](#).

Check with your application provider, or your application provider's documentation for statements of compatibility with these requirements.

IMPORTANT

The Azure AD SCIM implementation is built on top of the Azure AD user provisioning service, which is designed to constantly keep users in sync between Azure AD and the target application, and implements a very specific set of standard operations. It's important to understand these behaviors to understand the behavior of the Azure AD SCIM client. For more information, see the section [Provisioning cycles: Initial and incremental](#) in [How provisioning works](#).

Getting started

Applications that support the SCIM profile described in this article can be connected to Azure Active Directory using the "non-gallery application" feature in the Azure AD application gallery. Once connected, Azure AD runs a synchronization process every 40 minutes where it queries the application's SCIM endpoint for assigned users and groups, and creates or modifies them according to the assignment details.

To connect an application that supports SCIM:

1. Sign in to the [AAD portal](#). Note that you can get access a free trial for Azure Active Directory with P2 licenses by signing up for the [developer program](#)
2. Select **Enterprise applications** from the left pane. A list of all configured apps is shown, including apps that were added from the gallery.
3. Select **+ New application > + Create your own application**.

4. Enter a name for your application, choose the option "*integrate any other application you don't find in the gallery*" and select **Add** to create an app object. The new app is added to the list of enterprise applications and opens to its app management screen.

Browse Azure AD Gallery (Preview)

Create your own application

What's the name of your app? SCIM-01 App

What are you looking to do with your application?

- Configure Application Proxy for secure remote access to an on-premises application
- Register an application you're working on to integrate with Azure AD
- Integrate any other application you don't find in the gallery

Cloud platforms

On-premises applications

Create

Azure AD application gallery

NOTE

If you are using the old app gallery experience, follow the screen guide below.

Categories

All (3120)

Business management (377)

Collaboration (445)

Construction (8)

Consumer (43)

Content management (148)

CRM (151)

Data services (148)

Developer services (105)

E-commerce (75)

Education (141)

ERP (82)

Finance (256)

Health (62)

Human resources (281)

IT infrastructure (191)

Mail (34)

management (1)

Add an application

Add your own app

Application you're developing

On-premises application

Non-gallery application

Register an app you're working on to integrate it with Azure AD

Configure Azure AD Application Proxy to enable secure remote access

Integrate any other application that you don't find in the gallery

Add your own application

* Name: SCIM app

Once you decide on a name for your new application, click the "Add" button below and we'll walk you through some simple configuration steps to get the application working.

Supports:

- SAML-based single sign-on [Learn more](#)
- Automatic User Provisioning with SCIM [Learn more](#)
- Password-based single sign-on [Learn more](#)

Add

Azure AD old app gallery experience

5. In the app management screen, select **Provisioning** in the left panel.
6. In the **Provisioning Mode** menu, select **Automatic**.

The screenshot shows the Azure portal interface for managing an SCIM application. On the left, a sidebar lists various management options like Overview, Deployment Plan, Diagnose and solve problems, Properties, Owners, Users and groups, Single sign-on, Provisioning (which is selected), Application proxy, Self-service, Conditional Access, Permissions, Token encryption (Preview), Activity, and Sign-ins. The main content area is titled 'SCIM app - Provisioning - Provisioning' and includes sections for Admin Credentials (with fields for Tenant URL and Secret Token) and Mappings.

Configuring provisioning in the Azure portal

7. In the **Tenant URL** field, enter the URL of the application's SCIM endpoint. Example:
`https://api.contoso.com/scim/`
8. If the SCIM endpoint requires an OAuth bearer token from an issuer other than Azure AD, then copy the required OAuth bearer token into the optional **Secret Token** field. If this field is left blank, Azure AD includes an OAuth bearer token issued from Azure AD with each request. Apps that use Azure AD as an identity provider can validate this Azure AD-issued token.

NOTE

It's **not** recommended to leave this field blank and rely on a token generated by Azure AD. This option is primarily available for testing purposes.

9. Select **Test Connection** to have Azure Active Directory attempt to connect to the SCIM endpoint. If the attempt fails, error information is displayed.

NOTE

Test Connection queries the SCIM endpoint for a user that doesn't exist, using a random GUID as the matching property selected in the Azure AD configuration. The expected correct response is HTTP 200 OK with an empty SCIM ListResponse message.

10. If the attempts to connect to the application succeed, then select **Save** to save the admin credentials.
11. In the **Mappings** section, there are two selectable sets of **attribute mappings**: one for user objects and one for group objects. Select each one to review the attributes that are synchronized from Azure Active Directory to your app. The attributes selected as **Matching** properties are used to match the users and groups in your app for update operations. Select **Save** to commit any changes.

NOTE

You can optionally disable syncing of group objects by disabling the "groups" mapping.

12. Under **Settings**, the **Scope** field defines which users and groups are synchronized. Select **Sync only assigned users and groups** (recommended) to only sync users and groups assigned in the **Users and groups** tab.
13. Once your configuration is complete, set the **Provisioning Status** to **On**.
14. Select **Save** to start the Azure AD provisioning service.
15. If syncing only assigned users and groups (recommended), be sure to select the **Users and groups** tab and assign the users or groups you want to sync.

Once the initial cycle has started, you can select **Provisioning logs** in the left panel to monitor progress, which shows all actions done by the provisioning service on your app. For more information on how to read the Azure AD provisioning logs, see [Reporting on automatic user account provisioning](#).

NOTE

The initial cycle takes longer to perform than later syncs, which occur approximately every 40 minutes as long as the service is running.

Publish your application to the AAD application gallery

If you're building an application that will be used by more than one tenant, you can make it available in the Azure AD application gallery. This will make it easy for organizations to discover the application and configure provisioning. Publishing your app in the Azure AD gallery and making provisioning available to others is easy. Check out the steps [here](#). Microsoft will work with you to integrate your application into our gallery, test your endpoint, and release onboarding [documentation](#) for customers to use.

Gallery onboarding checklist

Use the checklist to onboard your application quickly and customers have a smooth deployment experience. The information will be gathered from you when onboarding to the gallery.

- Support a [SCIM 2.0](#) user and group endpoint (Only one is required but both are recommended)
- Support at least 25 requests per second per tenant to ensure that users and groups are provisioned and deprovisioned without delay (Required)
- Establish engineering and support contacts to guide customers post gallery onboarding (Required)
- 3 Non-expiring test credentials for your application (Required)
- Support the OAuth authorization code grant or a long lived token as described below (Required)
- Establish an engineering and support point of contact to support customers post gallery onboarding (Required)
- [Support schema discovery \(required\)](#)
- Support updating multiple group memberships with a single PATCH
- Document your SCIM endpoint publicly

Authorization to provisioning connectors in the application gallery

The SCIM spec doesn't define a SCIM-specific scheme for authentication and authorization and relies on the use of existing industry standards.

AUTHORIZATION METHOD	PROS	CONS	SUPPORT
Username and password (not recommended or supported by Azure AD)	Easy to implement	Insecure - Your Pa\$\$word doesn't matter	Not supported for new gallery or non-gallery apps.
Long-lived bearer token	Long-lived tokens do not require a user to be present. They are easy for admins to use when setting up provisioning.	Long-lived tokens can be hard to share with an admin without using insecure methods such as email.	Supported for gallery and non-gallery apps.
OAuth authorization code grant	Access tokens are much shorter-lived than passwords, and have an automated refresh mechanism that long-lived bearer tokens do not have. A real user must be present during initial authorization, adding a level of accountability.	Requires a user to be present. If the user leaves the organization, the token is invalid and authorization will need to be completed again.	Supported for gallery apps, but not non-gallery apps. However, you can provide an access token in the UI as the secret token for short term testing purposes. Support for OAuth code grant on non-gallery is in our backlog, in addition to support for configurable auth / token URLs on the gallery app.
OAuth client credentials grant	Access tokens are much shorter-lived than passwords, and have an automated refresh mechanism that long-lived bearer tokens do not have. Both the authorization code grant and the client credentials grant create the same type of access token, so moving between these methods is transparent to the API. Provisioning can be completely automated, and new tokens can be silently requested without user interaction.		Not supported for gallery and non-gallery apps. Support is in our backlog.

NOTE

It's not recommended to leave the token field blank in the AAD provisioning configuration custom app UI. The token generated is primarily available for testing purposes.

OAuth code grant flow

The provisioning service supports the [authorization code grant](#) and after submitting your request for publishing your app in the gallery, our team will work with you to collect the following information:

- **Authorization URL**, a URL by the client to obtain authorization from the resource owner via user-agent redirection. The user is redirected to this URL to authorize access.
- **Token exchange URL**, a URL by the client to exchange an authorization grant for an access token, typically with client authentication.
- **Client ID**, the authorization server issues the registered client a client identifier, which is a unique string

representing the registration information provided by the client. The client identifier is not a secret; it is exposed to the resource owner and **must not** be used alone for client authentication.

- **Client secret**, a secret generated by the authorization server that should be a unique value known only to the authorization server.

NOTE

The Authorization URL and Token exchange URL are currently not configurable per tenant.

NOTE

OAuth v1 is not supported due to exposure of the client secret. OAuth v2 is supported.

Best practices (recommended, but not required):

- Support multiple redirect URLs. Administrators can configure provisioning from both "portal.azure.com" and "aad.portal.azure.com". Supporting multiple redirect URLs will ensure that users can authorize access from either portal.
- Support multiple secrets for easy renewal, without downtime.

How to setup OAuth code grant flow

1. Sign in to the Azure portal, go to **Enterprise applications > Application > Provisioning** and select **Authorize**.
 - a. Azure portal redirects user to the Authorization URL (sign in page for the third party app).
 - b. Admin provides credentials to the third party application.
 - c. Third party app redirects user back to Azure portal and provides the grant code
 - d. Azure AD provisioning services calls the token URL and provides the grant code. The third party application responds with the access token, refresh token, and expiry date
2. When the provisioning cycle begins, the service checks if the current access token is valid and exchanges it for a new token if needed. The access token is provided in each request made to the app and the validity of the request is checked before each request.

NOTE

While it's not possible to setup OAuth on the non-gallery applications, you can manually generate an access token from your authorization server and input it as the secret token to a non-gallery application. This allows you to verify compatibility of your SCIM server with the AAD SCIM client before onboarding to the app gallery, which does support the OAuth code grant.

Long-lived OAuth bearer tokens: If your application doesn't support the OAuth authorization code grant flow, instead generate a long lived OAuth bearer token that an administrator can use to setup the provisioning integration. The token should be perpetual, or else the provisioning job will be **quarantined** when the token expires.

For additional authentication and authorization methods, let us know on [UserVoice](#).

Gallery go-to-market launch check list

To help drive awareness and demand of our joint integration, we recommend you update your existing documentation and amplify the integration in your marketing channels. The below is a set of checklist activities

we recommend you complete to support the launch

- Ensure your sales and customer support teams are aware, ready, and can speak to the integration capabilities. Brief your teams, provide them with FAQs and include the integration into your sales materials.
- Craft a blog post or press release that describes the joint integration, the benefits and how to get started.
[Example: Imprivata and Azure Active Directory Press Release](#)
- Leverage your social media like Twitter, Facebook or LinkedIn to promote the integration to your customers. Be sure to include @AzureAD so we can retweet your post. [Example: Imprivata Twitter Post](#)
- Create or update your marketing pages/website (e.g. integration page, partner page, pricing page, etc.) to include the availability of the joint integration. [Example: Pingboard integration Page](#), [Smartsheet integration page](#), [Monday.com pricing page](#)
- Create a help center article or technical documentation on how customers can get started. [Example: Envoy + Microsoft Azure Active Directory integration](#).
- Alert customers of the new integration through your customer communication (monthly newsletters, email campaigns, product release notes).

Next steps

[Develop a sample SCIM endpoint](#) [Automate user provisioning and deprovisioning to SaaS apps](#) [Customize attribute mappings for user provisioning](#) [Writing expressions for attribute mappings](#) [Scoping filters for user provisioning](#) [Account provisioning notifications](#) [List of tutorials on how to integrate SaaS apps](#)

Increase resilience of authentication and authorization applications you develop

4/12/2022 • 2 minutes to read • [Edit Online](#)

Microsoft Identity uses modern, token-based authentication and authorization. This means that a client application acquires tokens from an Identity provider to authenticate the user and to authorize the application to call protected APIs. A service will validate tokens.

A token is valid for a certain length of time before the app must acquire a new one. Rarely, a call to retrieve a token could fail due to an issue like network or infrastructure failure or authentication service outage. In this document, we outline steps a developer can take to increase resilience in their applications if a token acquisition failure occurs.

These articles provide guidance on increasing resiliency in apps using the Microsoft identity platform and Azure Active Directory. There is guidance for both for client and service applications that work on behalf of a signed in user as well as daemon applications that work on their own behalf. They contain best practices for using tokens as well as calling resources.

- [Build resilience into applications that sign-in users](#)
- [Build resilience into applications without users](#)
- [Build resilience in your identity and access management infrastructure](#)
- [Build resilience in your CIAM systems](#)
- [Build services that are resilient to metadata refresh](#)

Increase the resilience of authentication and authorization in client applications you develop

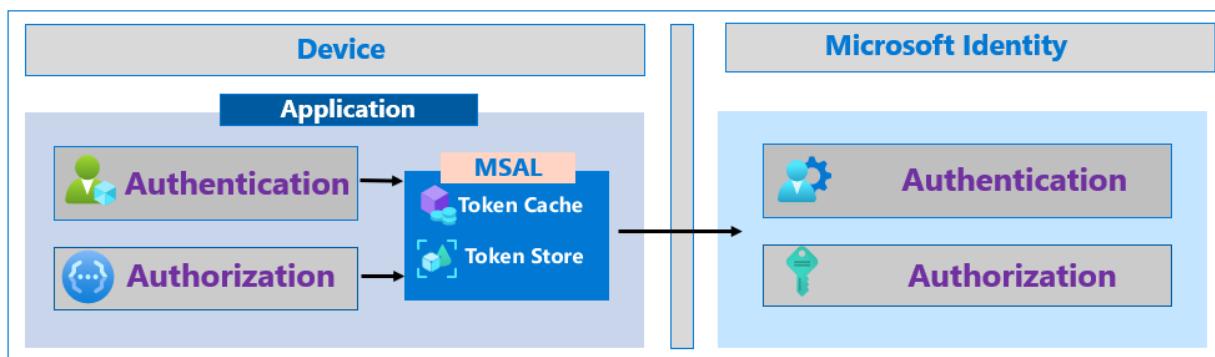
4/12/2022 • 12 minutes to read • [Edit Online](#)

This section provides guidance on building resilience into client applications that use the Microsoft identity platform and Azure Active Directory to sign in users and perform actions on behalf of those users.

Use the Microsoft Authentication Library (MSAL)

The [Microsoft Authentication Library \(MSAL\)](#) is a key part of the [Microsoft identity platform](#). It simplifies and manages acquiring, managing, caching, and refreshing tokens, and uses best practices for resilience. MSAL is designed to enable a secure solution without developers having to worry about the implementation details.

MSAL caches tokens and uses a silent token acquisition pattern. It also automatically serializes the token cache on platforms that natively provide secure storage like Windows UWP, iOS and Android. Developers can customize the serialization behavior when using [Microsoft.Identity.Web](#), [MSAL.NET](#), [MSAL for Java](#), and [MSAL for Python](#).



When using MSAL, token caching, refreshing, and silent acquisition is supported automatically. You can use simple patterns to acquire the tokens necessary for modern authentication. We support many languages, and you can find a sample that matches your language and scenario on our [Samples](#) page.

- [C#](#)
- [JavaScript](#)

```
try
{
    result = await app.AcquireTokenSilent(scopes, account).ExecuteAsync();
}
catch(MsalUiRequiredException ex)
{
    result = await app.AcquireToken(scopes).WithClaims(ex.Claims).ExecuteAsync()
}
```

MSAL can in some cases proactively refresh tokens. When Microsoft Identity issues a long-lived token, it can send information to the client for the optimal time to refresh the token ("refresh_in"). MSAL will proactively refresh the token based on this information. The app will continue to run while the old token is valid but will have a longer timeframe during which to make another successful token acquisition.

Stay up to date

Developers should have a process for updating to the latest MSAL release. Authentication is part of your app security and your app needs to stay current with the security improvements contained in new MSAL releases. This is generally good practice for libraries under continuous development and doing so will ensure you have the most up to date code with respect to app resilience. As Microsoft Identity continues to innovate on ways for applications to be more resilient, apps that use the latest MSAL will be the most prepared to build on these innovations.

[Check the latest MSAL.js version and release notes](#)

[Check the latest MSAL .NET version and release notes](#)

[Check the latest MSAL Python version and release notes](#)

[Check the latest MSAL Java version and release notes](#)

[Check the latest MSAL iOS and macOS version and release notes](#)

[Check the latest MSAL Android version and release notes](#)

[Check the latest MSAL Angular version and release notes](#)

[Check the latest Microsoft.Identity.Web version and release notes](#)

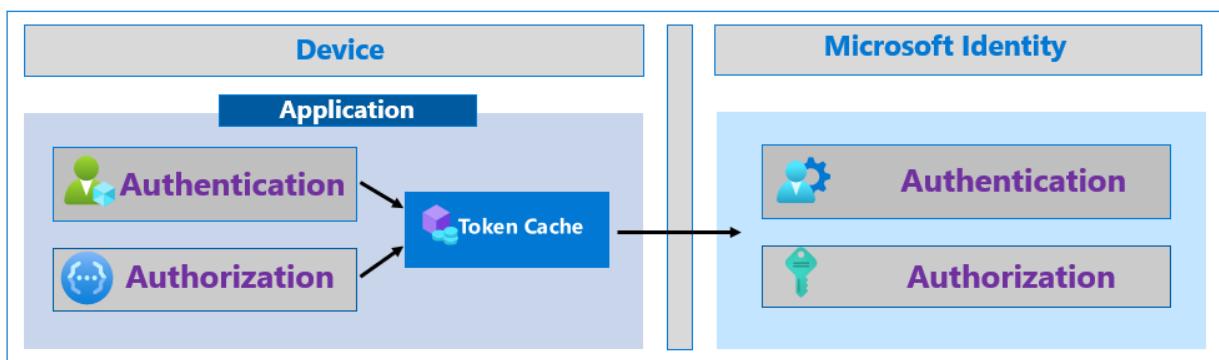
Use resilient patterns for token handling

If you are not using MSAL, you can use these resilient patterns for token handling. These best practices are implemented automatically by the MSAL library.

In general, an application that uses modern authentication will call an endpoint to retrieve tokens that authenticate the user or authorize the application to call protected APIs. MSAL is meant to handle the details of authentication and implements several patterns to improve resilience of this process. Use the guidance in this section to implement best practices if you choose to use a library other than MSAL. If you use MSAL, you get all of these best-practices for free, as MSAL implements them automatically.

Cache tokens

Apps should properly cache tokens received from Microsoft Identity. When your app receives tokens, the HTTP response that contains the tokens also contains an "expires_in" property that tells the application how long to cache, and reuse, the token. It is important that applications use the "expires_in" property to determine the lifespan of the token. Application must never attempt to decode an API access token.

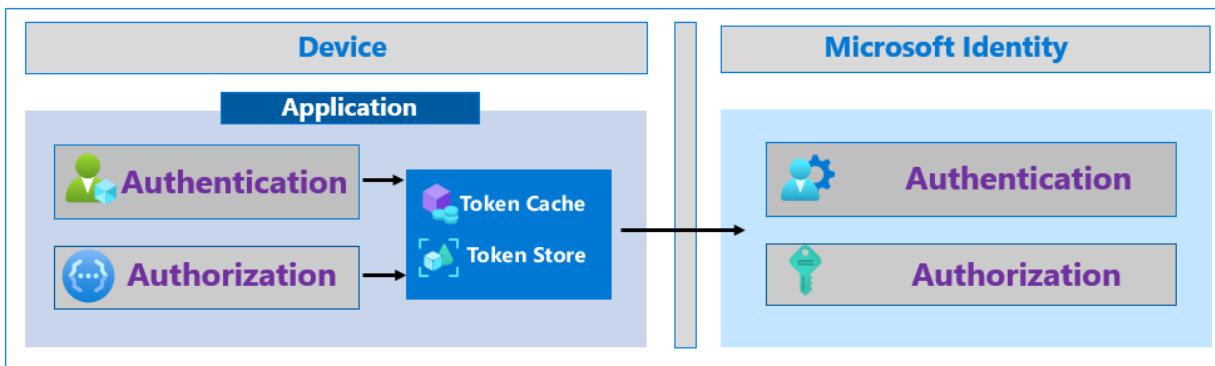


Using the cached token prevents unnecessary traffic between your app and Microsoft Identity, and makes your app less susceptible to token acquisition failures by reducing the number of token acquisition calls. Cached tokens also improve your application's performance as the app needs to block on acquiring tokens less. Your user can stay signed-in to your application for the length of that token's lifetime.

Serialize and persist tokens

Apps should securely serialize their token cache to persist the tokens between instances of the app. Tokens can

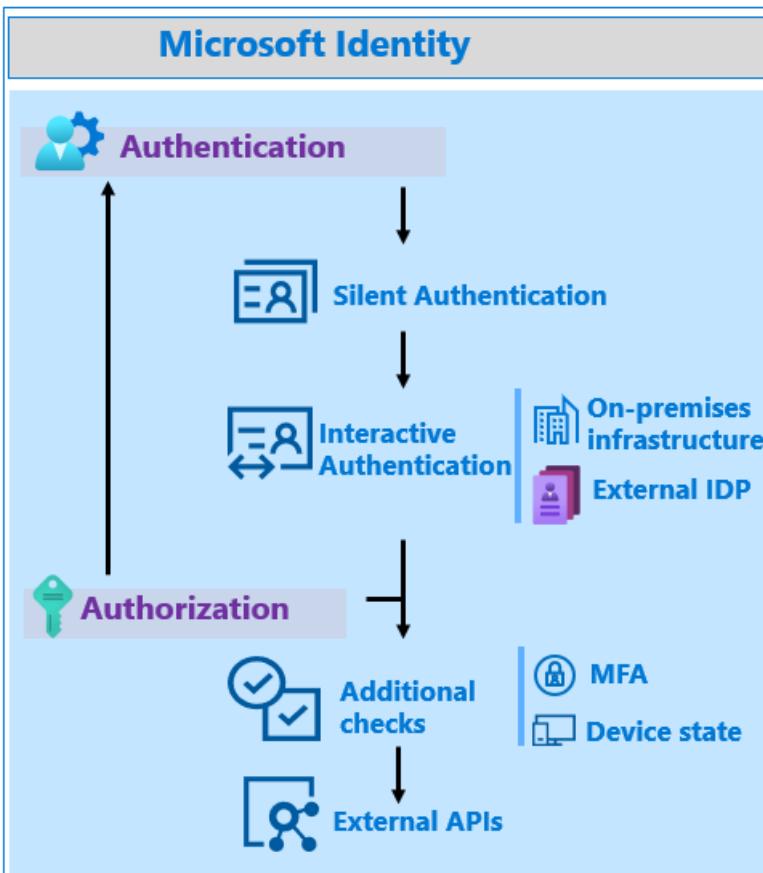
be reused as long as they are within their valid lifetime. [Refresh tokens](#), and, increasingly, [access tokens](#), are issued for many hours. This valid time can span a user starting your application many times. When your app starts, it should check to see if there is a valid access or refresh token that can be used. This will increase the app's resilience and performance as it avoids any unnecessary calls to Microsoft Identity.



The persistent token storage should be access controlled and encrypted to the owning user or process identity. On platforms like mobile, Windows and Mac, the developer should take advantage of built-in capabilities for storing credentials.

Acquire tokens silently

The process of authenticating a user or retrieving authorization to call an API can require multiple steps in Microsoft Identity. For example, when the user signs in for the first time they may need to enter credentials and perform a multi-factor authentication via a text message. Each step adds a dependency on the resource that provides that service. The best experience for the user, and the one with the least dependencies, is to attempt to acquire a token silently to avoid these extra steps before requesting user interaction.



Acquiring a token silently starts with using a valid token from the app's token cache. If there is no valid token available, the app should attempt to acquire a token using a refresh token, if available, and the token endpoint. If neither of these options is available, the app should acquire a token using the "prompt=none" parameter. This

will use the authorization endpoint, but not show any UI to the user. If the Microsoft Identity can provide a token to the app without interacting with the user, it will. If none of these methods result in a token, then a user will need to re-authenticate interactively.

NOTE

In general, apps should avoid using prompts like "login" and "consent" as they will force user interaction even when no interaction is required.

Handle service responses properly

While applications should handle all error responses, there are some responses that can impact resilience. If your application receives an HTTP 429 response code, Too Many Requests, Microsoft Identity is throttling your requests. If your app continues to make too many requests, it will continue to be throttled preventing your app from receiving tokens. Your application should not attempt to acquire a token again until after the time, in seconds, in the Retry-After response field has passed. Receiving a 429 response is often an indication that the application is not caching and reusing tokens correctly. Developers should review how tokens are cached and reused in the application.

When an application receives an HTTP 5xx response code the app must not enter a fast retry loop. When present, the application should honor the same Retry-After handling as it does for a 429 response. If no Retry-After header is provided by the response, we recommend implementing an exponential back-off retry with the first retry at least 5 seconds after the response.

When a request times out applications should not retry immediately. Implement an exponential back-off retry with the first retry at least 5 seconds after the response.

Evaluate options for retrieving authorization related information

Many applications and APIs need specific information about the user to make authorization decisions. There are a few ways for an application to get this information. Each method has its advantages and disadvantages. Developers should weigh these to determine which strategy is best for resilience in their app.

Tokens

Identity (ID) tokens and access tokens contain standard claims that provide information about the subject. These are documented in [Microsoft identity platform ID tokens](#) and [Microsoft identity platform access tokens](#). If the information your app needs is already in the token, then the most efficient technique for retrieving that data is to use token claims as that will save the overhead of an additional network call to retrieve information separately. Fewer network calls mean higher overall resilience for the application.

NOTE

Some applications call the UserInfo endpoint to retrieve claims about the user that authenticated. The information available in the ID token that your app can receive is a superset of the information it can get from the UserInfo endpoint. Your app should use the ID token to get information about the user instead of calling the UserInfo endpoint.

An app developer can augment standard token claims with [optional claims](#). One common optional claim is [groups](#). There are several ways to add group claims. The "Application Group" option only includes groups assigned to the application. The "All" or "Security groups" options include groups from all apps in the same tenant, which can add many groups to the token. It is important to evaluate the effect in your case, as it can potentially negate the efficiency gained by requesting groups in the token by causing token bloat and even requiring additional calls to get the full list of groups.

Instead of using groups in your token you can instead use and include app roles. Developers can define [app](#)

[roles](#) for their apps and APIs which the customer can manage from their directory using the portal or APIs. IT Pros can then assign roles to different users and groups to control who has access to what content and functionality. When a token is issued for the application or API, the roles assigned to the user will be available in the roles claim in the token. Getting this information directly in a token can save additional APIs calls.

Finally, IT Admins can also add claims based on specific information in a tenant. For example, an enterprise can have an extension to have an enterprise specific User ID.

In all cases, adding information from the directory directly to a token can be efficient and increase the apps resilience by reducing the number of dependencies the app has. On the other hand, it does not address any resilience issues from being unable to acquire a token. You should only add optional claims for the main scenarios of your application. If the app requires information only for the admin functionality, then it is best for the application to obtain that information only as needed.

Microsoft Graph

Microsoft Graph provides a unified API endpoint to access the Microsoft 365 data that describes the patterns of productivity, identity and security in an organization. Applications that use Microsoft Graph can potentially use any of the information across Microsoft 365 for authorization decisions.

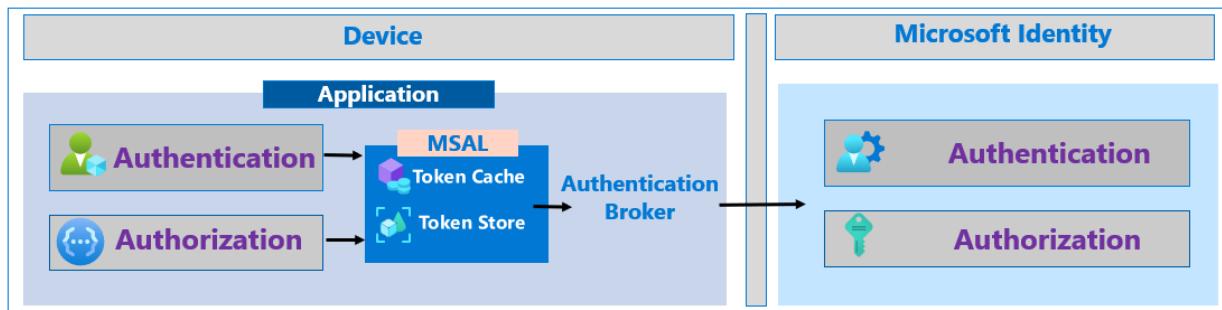
Apps require just a single token to access all of Microsoft 365. This is more resilient than using the older APIs that are specific to Microsoft 365 components like Microsoft Exchange or Microsoft SharePoint where multiple tokens are required.

When using Microsoft Graph APIs, we suggest you use a [Microsoft Graph SDK](#). The Microsoft Graph SDKs are designed to simplify building high-quality, efficient, and resilient applications that access Microsoft Graph.

For authorization decisions, developers should consider when to use the claims available in a token as an alternative to some Microsoft Graph calls. As mentioned above, developers could request groups, app roles, and optional claims in their tokens. In terms of resilience, using Microsoft Graph for authorization requires additional network calls that rely on Microsoft Identity (to get the token to access Microsoft Graph) as well as Microsoft Graph itself. However, if your application already relies on Microsoft Graph as its data layer, then relying on the Graph for authorization is not an additional risk to take.

Use broker authentication on mobile devices

On mobile devices, using an authentication broker like Microsoft Authenticator will improve resilience. The broker adds benefits above what is available with other options such as the system browser or an embedded WebView. The authentication broker can utilize a [primary refresh token](#) (PRT) that contains claims about the user and the device and can be used to get authentication tokens to access other applications from the device. When a PRT is used to request access to an application, its device and MFA claims are trusted by Azure AD. This increases resilience by avoiding additional steps to authenticate the device again. Users won't be challenged with multiple MFA prompts on the same device, therefore increasing resilience by reducing dependencies on external services and improving the user experience.



Broker authentication is automatically supported by MSAL. You can find more information on using brokered authentication on the following pages:

- [Configure SSO on macOS and iOS](#)
- [How to enable cross-app SSO on Android using MSAL](#)

Adopt Continuous Access Evaluation

[Continuous Access Evaluation \(CAE\)](#) is a recent development that can increase application security and resilience with long-lived tokens. CAE is an emerging industry standard being developed in the Shared Signals and Events Working Group of the OpenID Foundation. With CAE, an access token can be revoked based on [critical events](#) and [policy evaluation](#), rather than relying on a short token lifetime. For some resource APIs, because risk and policy are evaluated in real time, CAE can substantially increase token lifetime up to 28 hours. As resource APIs and applications adopt CAE, Microsoft Identity will be able to issue access tokens that are revocable and are valid for extended periods of time. These long-lived tokens will be proactively refreshed by MSAL.

While CAE is in early phases, it is possible to [develop client applications today that will benefit from CAE](#) when the resources (APIs) the application uses adopt CAE. As more resources adopt CAE, your application will be able to acquire CAE enabled tokens for those resources as well. The Microsoft Graph API, and [Microsoft Graph SDKs](#), will preview CAE capability early 2021. If you would like to participate in the public preview of Microsoft Graph with CAE, you can let us know you are interested here: <https://aka.ms/GraphCAEPreview>.

If you develop resource APIs, we encourage you to participate in the [Shared Signals and Events WG](#). We are working with this group to enable the sharing of security events between Microsoft Identity and resource providers.

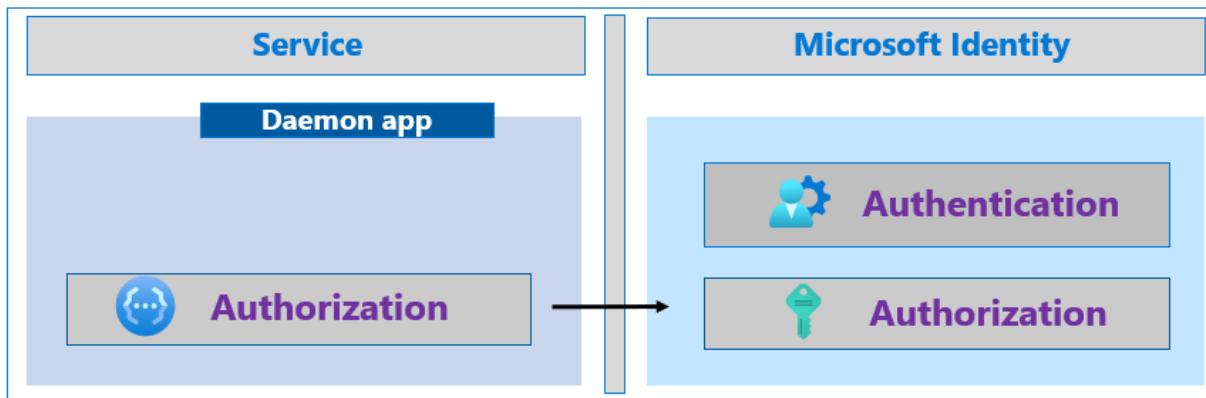
Next steps

- [How to use Continuous Access Evaluation enabled APIs in your applications](#)
- [Build resilience into daemon applications](#)
- [Build resilience in your identity and access management infrastructure](#)
- [Build resilience in your CIAM systems](#)

Increase the resilience of authentication and authorization in daemon applications you develop

4/12/2022 • 3 minutes to read • [Edit Online](#)

This article provides guidance on how developers can use the Microsoft identity platform and Azure Active Directory to increase the resilience of daemon applications. This includes background processes, services, server to server apps, and applications without users.



Use Managed Identities for Azure Resources

Developers building daemon apps on Microsoft Azure can use [Managed Identities for Azure Resources](#). Managed Identities eliminate the need for developers to manage secrets and credentials. The feature improves resilience by avoiding mistakes around certificate expiry, rotation errors, or trust. It also has several built-in features meant specifically to increase resilience.

Managed Identities use long lived access tokens and information from Microsoft Identity to proactively acquire new tokens within a large window of time before the existing token expires. Your app can continue to run while attempting to acquire a new token.

Managed Identities also use regional endpoints to improve performance and resilience against out-of-region failures. Using a regional endpoint helps to keep all traffic inside a geographical area. For example, if your Azure Resource is in WestUS2, all the traffic, including Microsoft Identity generated traffic, should stay in WestUS2. This eliminates possible points of failure by consolidating the dependencies of your service.

Use the Microsoft Authentication Library

Developers of daemon apps who do not use Managed Identities can use the [Microsoft Authentication Library \(MSAL\)](#), which makes implementing authentication and authorization simple, and automatically uses best practices for resilience. MSAL will make the process of providing the required Client Credentials easier. For example, your application does not need to implement creating and signing JSON Web Token assertions when using certificate-based credentials.

Use Microsoft.Identity.Web for .NET Developers

Developers building daemon apps on ASP.NET Core can use the [Microsoft.Identity.Web](#) library. This library is built on top of MSAL to make implementing authorization even easier for ASP.NET Core apps. It includes several [distributed token cache](#) strategies for distributed apps that can run in multiple regions.

Cache and store tokens

If you are not using MSAL to implement authentication and authorization, you can implement some best practices for caching and storing tokens. MSAL implements and follows these best practices automatically.

An application acquires tokens from an Identity provider to authorize the application to call protected APIs. When your app receives tokens, the response that contains the tokens also contains an "expires_in" property that tells the application how long to cache, and reuse, the token. It is important that applications use the "expires_in" property to determine the lifespan of the token. Application must never attempt to decode an API access token. Using the cached token prevents unnecessary traffic between your app and Microsoft Identity. Your user can stay signed-in to your application for the length of that token's lifetime.

Properly handle service responses

Finally, while applications should handle all error responses, there are some responses that can impact resilience. If your application receives an HTTP 429 response code, Too Many Requests, Microsoft Identity is throttling your requests. If your app continues to make too many requests, it will continue to be throttled preventing your app from receiving tokens. Your application should not attempt to acquire a token again until after the time, in seconds, in the "Retry-After" response field has passed. Receiving a 429 response is often an indication that the application is not caching and reusing tokens correctly. Developers should review how tokens are cached and reused in the application.

When an application receives an HTTP 5xx response code the app must not enter a fast retry loop. When present, the application should honor the same "Retry-After" handling as it does for a 429 response. If no "Retry-After" header is provided by the response, we recommend implementing an exponential back-off retry with the first retry at least 5 seconds after the response.

When a request times out applications should not retry immediately. Implement an exponential back-off retry with the first retry at least 5 seconds after the response.

Next steps

- [Build resilience into applications that sign-in users](#)
- [Build resilience in your identity and access management infrastructure](#)
- [Build resilience in your CIAM systems](#)

Build services that are resilient to Azure AD's OpenID Connect metadata refresh

4/12/2022 • 2 minutes to read • [Edit Online](#)

Protected web APIs need to validate access tokens. Web apps also validate the ID tokens. Token Validation has multiple parts, checking whether the token belongs to the application, has been issued by a trusted Identity Provider (IDP), has a lifetime that's still in range and hasn't been tampered with. There can also be special validations. For instance, the app needs to validate the signature and that signing keys (when embedded in a token) are trusted and that the token isn't being replayed. When the signing keys aren't embedded in the token, they need to be fetched from the identity provider (Discovery or Metadata). Sometimes it's also necessary to obtain keys dynamically at runtime.

Web apps and web APIs need to refresh stale OpenID Connect metadata for them to be resilient. This article helps guide on how to achieve resilient apps. It applies to ASP.NET Core, ASP.NET classic, and Microsoft.IdentityModel.

ASP.NET Core

Use latest version of [Microsoft.IdentityModel.*](#) and manually follow the guidelines below.

In the `ConfigureServices` method of the `Startup.cs`, ensure that `JwtBearerOptions.RefreshOnIssuerKeyNotFound` is set to true, and that you're using the latest [Microsoft.IdentityModel.*](#) library. This property should be enabled by default.

```
services.Configure<JwtBearerOptions>(AzureADDefaults.JwtBearerAuthenticationScheme, options =>
{
    ...
    // shouldn't be necessary as it's true by default
    options.RefreshOnIssuerKeyNotFound = true;
    ...
});
```

ASP.NET/ OWIN

Microsoft recommends that you move to ASP.NET Core, as development has stopped on ASP.NET.

If you're using ASP.NET classic, use the latest [Microsoft.IdentityModel.*](#).

OWIN has an automatic 24-hour refresh interval for the `OpenIdConnectConfiguration`. This refresh will only be triggered if a request is received after the 24-hour time span has passed. As far as we know, there's no way to change this value or trigger a refresh early, aside from restarting the application.

Microsoft.IdentityModel

If you validate your token yourself, for instance in an Azure Function, use the latest version of [Microsoft.IdentityModel.*](#) and follow the metadata guidance illustrated by the code snippets below.

```
var configManager =
    new ConfigurationManager<OpenIdConnectConfiguration>(
        "http://someaddress.com",
        new OpenIdConnectConfigurationRetriever());

var config = await configManager.GetConfigurationAsync().ConfigureAwait(false);
var validationParameters = new TokenValidationParameters()
{
    ...
    IssuerSigningKeys = config.SigningKeys;
    ...
};

var tokenHandler = new JsonWebTokenHandler();
result = Handler.ValidateToken(jwtToken, validationParameters);
if (result.Exception != null && result.Exception is SecurityTokenSignatureKeyNotFoundException)
{
    configManager.RequestRefresh();
    config = await configManager.GetConfigurationAsync().ConfigureAwait(false);
    validationParameters = new TokenValidationParameters()
    {
        ...
        IssuerSigningKeys = config.SigningKeys,
        ...
    };
}

// attempt to validate token again after refresh
result = Handler.ValidateToken(jwtToken, validationParameters);
}
```

Next steps

To learn more, see [token validation in a protected web API](#)

How to use Continuous Access Evaluation enabled APIs in your applications

4/12/2022 • 2 minutes to read • [Edit Online](#)

Continuous Access Evaluation (CAE) is an Azure AD feature that allows access tokens to be revoked based on critical events and policy evaluation rather than relying on token expiry based on lifetime. For some resource APIs, because risk and policy are evaluated in real time, this can increase token lifetime up to 28 hours. These long-lived tokens will be proactively refreshed by the Microsoft Authentication Library (MSAL), increasing the resiliency of your applications.

This article shows you how to use CAE-enabled APIs in your applications. Applications not using MSAL can add support for [claims challenges](#), [claims requests](#), and [client capabilities](#) to use CAE.

Implementation considerations

To use Continuous Access Evaluation, both your app and the resource API it's accessing must be CAE-enabled. However, preparing your code to use a CAE enabled resource will not prevent you from using APIs that are not CAE enabled.

If a resource API implements CAE and your application declares it can handle CAE, your app will get CAE tokens for that resource. For this reason, if you declare your app CAE ready, your application must handle the CAE claim challenge for all resource APIs that accept Microsoft Identity access tokens. If you do not handle CAE responses in these API calls, your app could end up in a loop retrying an API call with a token that is still in the returned lifespan of the token but has been revoked due to CAE.

The code

The first step is to add code to handle a response from the resource API rejecting the call due to CAE. With CAE APIs will return a 401 status and a WWW-Authenticate header when the access token has been revoked or the API detects a change in IP address used. The WWW-Authenticate header contains a Claims Challenge that the application can use to acquire a new access token.

For example:

Your app would check for:

- the API call returning the 401 status
 - the existence of a WWW-Authenticate header containing:
 - an "error" parameter with the value "insufficient_claims"
 - a "claims" parameter

When these conditions are met, the app can extract and decode the claims challenge using MSAL.NET `WwwAuthenticateParameters` class.

```

if (APIresponse.IsSuccessStatusCode)
{
    // ...
}
else
{
    if (APIresponse.StatusCode == System.Net.HttpStatusCode.Unauthorized
        && APIresponse.Headers.WwwAuthenticate.Any())
    {
        string claimChallenge =
WwwAuthenticateParameters.GetClaimChallengeFromResponseHeaders(APIresponse.Headers);
    }
}

```

Your app would then use the claims challenge to acquire a new access token for the resource.

```

try
{
    authResult = await _clientApp.AcquireTokenSilent(scopes, firstAccount)
        .WithClaims(claimChallenge)
        .ExecuteAsync()
        .ConfigureAwait(false);
}
catch (MsalUiRequiredException)
{
    try
    {
        authResult = await _clientApp.AcquireTokenInteractive(scopes)
            .WithClaims(claimChallenge)
            .WithAccount(firstAccount)
            .ExecuteAsync()
            .ConfigureAwait(false);
    }
    // ...
}

```

Once your application is ready to handle the claim challenge returned by a CAE enabled resource, you can tell Microsoft Identity your app is CAE ready. To do this in your MSAL application, build your Public Client using the Client Capabilities of "cp1".

```

_clientApp = PublicClientApplicationBuilder.Create(App.ClientId)
    .WithDefaultRedirectUri()
    .WithAuthority(authority)
    .WithClientCapabilities(new [] {"cp1"})
    .Build();

```

You can test your application by signing in a user to the application then using the Azure portal to Revoke the user's sessions. The next time the app calls the CAE enabled API, the user will be asked to reauthenticate.

Next steps

- [Continuous access evaluation](#) conceptual overview
- [Claims challenges, claims requests, and client capabilities](#)

Claims challenges, claims requests, and client capabilities

4/12/2022 • 6 minutes to read • [Edit Online](#)

A *claims challenge* is a response sent from an API indicating that an access token sent by a client application has insufficient claims. This can be because the token does not satisfy the conditional access policies set for that API, or the access token has been revoked.

A *claims request* is made by the client application to redirect the user back to the identity provider to retrieve a new token with claims that will satisfy the additional requirements that were not met.

Applications that use enhanced security features like [Continuous Access Evaluation \(CAE\)](#) and [Conditional Access authentication context](#) must be prepared to handle claims challenges.

Your application will receive claims challenges from popular services like [Microsoft Graph](#) only if it declares its [client capabilities](#) in its calls to the service.

Claims challenge header format

The claims challenge is a directive as a `www-authenticate` header returned by an API when an [access token](#) presented to it isn't authorized, and a new access token with the right capabilities is required instead. The claims challenge comprises multiple parts: the HTTP status code of the response and the `www-authenticate` header, which itself has multiple parts and must contain a claims directive.

Here's an example:

```
HTTP 401; Unauthorized

www-authenticate =Bearer realm="",
authorization_uri="https://login.microsoftonline.com/common/oauth2/authorize", error="insufficient_claims",
claims="eyJhbGciOiJSUzIiLCJnfdG9rZW4iOnsiYWNycyI6eyJlc3NlbnRpYWwiOnRydWUsInZhbHVlIjoiYzEifX19"
```

HTTP Status Code: Must be **401 Unauthorized**.

www-authenticate response header containing:

PARAMETER	REQUIRED/OPTIONAL	DESCRIPTION
Authentication type	Required	Must be Bearer .
Realm	Optional	The tenant ID or tenant domain name (for example, microsoft.com) being accessed. MUST be an empty string in the case where the authentication goes through the common endpoint .

PARAMETER	REQUIRED/OPTIONAL	DESCRIPTION
<code>authorization_uri</code>	Required	The URI of the authorize endpoint where an interactive authentication can be performed if necessary. If specified in realm, the tenant information MUST be included in the authorization_uri. If realm is an empty string, the authorization_uri MUST be against the common endpoint .
<code>error</code>	Required	Must be "insufficient_claims" when a claims challenge should be generated.
<code>claims</code>	Required when error is "insufficient_claims".	A quoted string containing a base 64 encoded claims request . The claims request should request claims for the "access_token" at the top level of the JSON object. The value (claims requested) will be context-dependent and specified later in this document. For size reasons, relying party applications SHOULD minify the JSON before base 64 encoding. The raw JSON of the example above is <pre>{"access_token":{"acr":{"essential":true,"value":"cp1"}}}</pre> .

The **401** response may contain more than one `www-authenticate` header. All fields in the preceding table must be contained within the same `www-authenticate` header. The `www-authenticate` header that contains the claims challenge *can* contain other fields. Fields in the header are unordered. According to RFC 7235, each parameter name must occur only once per authentication scheme challenge.

Claims request

When an application receives a claims challenge, it indicates that the prior access token is no longer considered valid. In this scenario, the application should clear the token from any local cache or user session. Then, it should redirect the signed-in user back to Azure Active Directory (Azure AD) to retrieve a new token by using the [OAuth 2.0 authorization code flow](#) with a *claims* parameter that will satisfy the additional requirements that were not met.

Here's an example:

```
GET https://login.microsoftonline.com/14c2f153-90a7-4689-9db7-9543bf084dad/oauth2/v2.0/authorize
?client_id=2810aca2-a927-4d26-8bca-5b32c1ef5ea9
&redirect_uri=https%3A%2F%contoso.com%3A44321%2Fsignin-oidc
&response_type=code
&scope=openid%20profile%20offline_access%20user.read%20Sites.Read.All
&response_mode=form_post
&login_hint=kalyan%40contoso.onmicrosoft.com
&domain_hint=organizations
&claims=%7B%22access_token%22%3A%7B%22acr%22%3A%7B%22essential%22%3Atrue%2C%22value%22%3A%22c1%22%7D%7D%7D
```

The claims challenge should be passed as a part of all calls to Azure AD's `/authorize` endpoint until a token is successfully retrieved, after which it is no longer needed.

To populate the claims parameter, the developer has to:

1. Decode the base64 string received earlier.

2. URL-encode the string and add again to the `claims` parameter.

Upon completion of this flow, the application will receive an Access Token that has the additional claims that prove that the user satisfied the conditions required.

Client capabilities

Client capabilities help a resources provider like a Web API detect whether the calling client application understands the claims challenge and can then customize its response accordingly. This capability might be useful when not all API clients are capable of handling claim challenges, and some earlier versions still expect a different response.

Some popular applications like [Microsoft Graph](#) send claims challenges only if the calling client app declares that it's capable of handling them by using *client capabilities*.

To avoid extra traffic or impacts to user experience, Azure AD does not assume that your app can handle claims challenged unless you explicitly opt in. An application will not receive claims challenges (and will not be able to use the related features such as CAE tokens) unless it declares it is ready to handle them with the "cp1" capability.

How to communicate client capabilities to Azure AD

The following example claims parameter shows how a client application communicates its capability to Azure AD in an [OAuth 2.0 authorization code flow](#).

```
Claims: {"access_token":{"xms_cc":{"values":["cp1"]}}}
```

Those using MSAL library will use the following code:

```
_clientApp = PublicClientApplicationBuilder.Create(App.ClientId)
    .WithDefaultRedirectUri()
    .WithAuthority(authority)
    .WithClientCapabilities(new [] {"cp1"})
    .Build();*
```

Those using Microsoft.Identity.Web can add the following code to the configuration file:

```
{
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    // the remaining settings
    // ...
    "ClientCapabilities": [ "cp1" ]
  },
}
```

An example of how the request to Azure AD will look like:

```
GET https://login.microsoftonline.com/14c2f153-90a7-4689-9db7-9543bf084dad/oauth2/v2.0/authorize
?client_id=2810aca2-a927-4d26-8bca-5b32c1ef5ea9
&redirect_uri=https%3A%2F%contoso.com%3A44321%2Fsignin-oidc
&response_type=code
&scope=openid%20profile%20offline_access%20user.read%20Sites.Read.All
&response_mode=form_post
&login_hint=kalyan%40contoso.onmicrosoft.com
&domain_hint=organizations
&claims=%7B%22access_token%22%3A%7B%22xms_cc%22%3A%7B%22values%22%3A%5B%22cp1%22%5D%7D%7D%7D
```

When you already have an existing payload for claims parameter, then you would add this to the existing set.

For example, if you already have the following response from a Condition Access authentication context operation

```
{"access_token": {"acrs": {"essential": true, "value": "c25"}}}
```

You would prepend the client capability in the existing **claims** payload.

```
{"access_token": {"xms_cc": {"values": ["cp1"]}, "acrs": {"essential": true, "value": "c25"}}}
```

Receiving xms_cc claim in an access token

To receive information about whether client applications can handle claims challenges, an API implementer must request **xms_cc** as an optional claim in its application manifest.

The **xms_cc** claim with a value of "cp1" in the access token is the authoritative way to identify a client application is capable of handling a claims challenge. **xms_cc** is an optional claim that will not always be issued in the access token, even if the client sends a claims request with "xms_cc". In order for an access token to contain the **xms_cc** claim, the resource application (that is, the API implementer) must request **xms_cc** as an [optional claim](#) in its application manifest. When requested as an optional claim, **xms_cc** will be added to the access token only if the client application sends **xms_cc** in the claims request. The value of the **xms_cc** claim request will be included as the value of the **xms_cc** claim in the access token, if it is a known value. The only currently known value is **cp1**.

The values are not case-sensitive and unordered. If more than one value is specified in the **xms_cc** claim request, those values will be a multi-valued collection as the value of the **xms_cc** claim.

A request of :

```
{ "access_token": { "xms_cc": {"values": ["cp1", "foo", "bar"] } }}
```

will result in a claim of

```
"xms_cc": [ "cp1", "foo", "bar" ]
```

in the access token, if **cp1**, **foo** and **bar** are known capabilities.

This is how the app's manifest looks like after the **xms_cc optional claim** has been requested

```
"optionalClaims":  
{  
    "accessToken": [  
        {  
            "additionalProperties": [],  
            "essential": false,  
            "name": "xms_cc",  
            "source": null  
        },  
        "idToken": [],  
        "saml2Token": []  
    ]  
}
```

The API can then customize their responses based on whether the client is capable of handling claims challenge

or not.

An example in C#

```
Claim ccClaim = context.User.FindAll(clientCapabilitiesClaim).FirstOrDefault(x => x.Type == "xms_cc");
if (ccClaim != null && ccClaim.Value == "cp1")
{
    // Return formatted claims challenge as this client understands this
}
else
{
    // Throw generic exception
    throw new UnauthorizedAccessException("The caller does not meet the authentication bar to carry out this
operation. The service cannot allow this operation");
}
```

Next steps

- [Microsoft identity platform and OAuth 2.0 authorization code flow](#)
- [How to use Continuous Access Evaluation enabled APIs in your applications](#)
- [Granular Conditional Access for sensitive data and actions](#)

Set up your application's Azure AD test environment

4/12/2022 • 8 minutes to read • [Edit Online](#)

As a developer, you use the software development lifecycle and move your app between development, test, and production environments. When this process is used for applications protected by the Microsoft identity platform, you should set up an Azure Active Directory (Azure AD) environment to be used for testing. This environment can be used in the early testing stages of the development lifecycle as well as a long-term, permanent test environment. Depending on your resource isolation requirements, you can use your organization's Azure AD production tenant or an entirely separate tenant for testing.

In this article, you learn how to set up an Azure AD test environment so you can test your application integrated with Microsoft identity platform. Evaluate the level of isolation needed and whether you need a separate tenant for testing or if you can use your production tenant.

Decide the level of isolation needed

In general, it is easier and less overhead to use your production tenant as a test environment. However, this is only a viable option if you can achieve the right level of isolation between test and production resources. Isolation is especially important for high privilege scenarios.

Set up your test environment in a separate tenant (not your organization's production tenant) if:

- You have a set of resources that requires unique, tenant-wide settings. For example, your app may need to access tenant resources as itself and not on behalf of a user (uses app-only permissions). App-only access requires admin consent that applies across the entire tenant and those permissions are hard to scope down safely within a tenant boundary.
- You have minimal risk tolerance for unauthorized access by tenant members to your test resources.
- Configuration changes could have critical impact on your production environment.
- You aren't able to create test users and associated test data in your tenant.
- You plan on performing automated sign-ins to your application for testing and your production tenant has configured authentication method policies that require some user interaction. For example, if multifactor authentication is required for all users you won't be able to perform automated sign-ins for integration testing.
- You must ensure that global administrators can't manage or access specific test resources. You'll need to isolate that resource in a separate tenant with separate global administrators.
- Adding non-production resources and/or workload to your production tenant would [exceed service or throttling limits](#) for that tenant.

If none of these conditions apply to you, then follow these steps to [set up your test environment in your production tenant](#). If any of them do apply, however, you should [set up a test environment in a separate tenant](#).

Set up a test environment in a separate tenant

If you can't safely constrain your test app in your production tenant, you can create a separate tenant for development and testing purposes.

Get a test tenant

If you don't already have a dedicated test tenant, you can create one for free using the Microsoft 365 Developer

Program or manually create one yourself.

Join the Microsoft 365 Developer Program (recommended)

The [Microsoft 365 Developer Program](#) is free and can have test user accounts and sample data packs automatically added to the tenant.

1. Click on the **Join now** button on the screen.
2. Sign in with a new Microsoft Account or use an existing (work) account you already have.
3. On the sign-up page select your region, enter a company name and accept the terms and conditions of the program before you click **Next**.
4. Click on **Set Up Subscription**. Specify the region where you want to create your new tenant, create a username, domain, and enter a password. This will create a new tenant and the first administrator of the tenant.
5. Enter the security information, which is needed to protect the administrator account of your new tenant. This will set up multifactor authentication for the account

Manually create a tenant

You can [manually create a tenant](#), which will be empty upon creation and will have to be configured with test data.

Populate your tenant with users

For convenience, you may want to invite yourself and other members of your development team to be guest users in the tenant. This will create separate guest objects in the test tenant, but means you only have to manage one set of credentials for your corporate account and your test account.

1. From the [Azure portal](#), click on **Azure Active Directory**.
2. Go to **Users**.
3. Click on **New guest user** and invite your work account email address.
4. Repeat for other members of the development and/or testing team for your application.

You can also create test users in your test tenant. If you used one of the Microsoft 365 sample packs, you may already have some test users in your tenant. If not, you should be able to create some yourself as the tenant administrator.

1. From the [Azure portal](#), click on **Azure Active Directory**.
2. Go to **Users**.
3. Click **New user** and create some new test users in your directory.

Get an Azure AD subscription (optional)

If you want to fully test Azure AD premium features on your application, you'll need to sign up your tenant for a [Premium P1 or Premium P2 license](#).

If you signed up using the Microsoft 365 Developer program, your test tenant will come with Azure AD P2 licenses. If not, you can still enable a one month [free trial of Azure AD premium](#).

Create and configure an app registration

You'll need to create an app registration to use in your test environment. This should be a separate registration from your eventual production app registration, to maintain security isolation between your test environment and your production environment. How you configure your application depends on the type of app you are building. For more information, check out the app registration steps for your app scenario in the left navigation pane, like this article for [web application registration](#).

Populate your tenant with policies

If your app will primarily be used by a single organization (commonly referred to as single tenant), and you have access to that production tenant, then you should try to replicate the settings of your production tenant that can

affect your app's behavior. That will lower the chances of unexpected errors when operating in production.

Conditional access policies

Replicating conditional access policies ensures you don't encounter unexpected blocked access when moving to production and your application can appropriately handle the errors it's likely to receive.

Viewing your production tenant conditional access policies may need to be performed by a company administrator.

1. Sign into the [Azure portal](#) using your production tenant account
2. Go to **Azure Active Directory > Enterprise applications > Conditional access**.
3. View the list of policies in your tenant. Click the first one.
4. Navigate to **Cloud apps or actions**.
5. If the policy only applies to a select group of apps, then move on to the next policy. If not, then it will likely apply to your app as well when you move to production. You should copy the policy over to your test tenant.

In a new tab or browser session, navigate to the [Azure portal](#), and sign into your test tenant.

1. Go to **Azure Active Directory > Enterprise applications > Conditional access**.
2. Click on **New policy**
3. Copy the settings from the production tenant policy, identified through the previous steps.

Permission grant policies

Replicating permission grant policies ensures you don't encounter unexpected prompts for admin consent when moving to production.

1. Sign into the [Azure portal](#) using your production tenant account
2. Click on **Azure Active Directory**.
3. Go to **Enterprise applications**.
4. From your production tenant, go to **Azure Active Directory > Enterprise applications > Consent and permissions > User consent settings**. Copy the settings there to your test tenant.

Token lifetime policies

Replicating token lifetime policies ensures tokens issued to your application don't expire unexpectedly in production.

Token lifetime policies can currently only be managed through PowerShell. Read about [configurable token lifetimes](#) to learn about identifying any token lifetime policies that apply to your whole production organization. Copy those policies to your test tenant.

Set up a test environment in your production tenant

If you can safely constrain your test app in your production tenant, go ahead and set up your tenant for testing purposes.

Create and configure an app registration

You'll need to create an app registration to use in your test environment. This should be a separate registration from your eventual production app registration, to maintain security isolation between your test environment and your production environment. How you configure your application depends on the type of app you are building. For more information, check out the [app registration steps for your app scenario](#) in the left navigation pane.

Create some test users

You'll need to create some test users with associated test data to use while testing your scenarios. This step might need to be performed by an admin

1. From the [Azure portal](#), click on Azure Active Directory.
2. Go to Users.
3. Click **New user** and create some new test users in your directory.

Add the test users to a group (optional)

For convenience, you can assign all these users to a group, which makes other assignment operations easier.

1. From the [Azure portal](#), click on Azure Active Directory.
2. Go to Groups.
3. Click **New group**.
4. Select either **Security** or **Microsoft 365** for group type.
5. Name your group.
6. Add the test users created in the previous step.

Restrict your test application to specific users

You can restrict the users in your tenant that are allowed to use your test application to specific users or groups, through user assignment. When you [created an app through App registrations](#), a representation of your app was created in **Enterprise applications** as well. Use the **Enterprise applications** settings to restrict who can use the application in your tenant.

IMPORTANT

If your app is a [multi-tenant app](#), this operation won't restrict users in other tenants from signing into and using your app. It will only restrict users in the tenant that user assignment is configured in.

For detailed instructions on restricting an app to specific users in a tenant, go to [restricting your app to a set of users](#).

Next steps

Learn about [throttling and service limits](#) you might hit while setting up a test environment.

For more detailed information about test environments, read [Securing Azure environments with Azure Active Directory](#).

Run automated integration tests

4/12/2022 • 9 minutes to read • [Edit Online](#)

As a developer, you want to run automated integration tests on the apps you develop. Calling your API protected by Microsoft identity platform (or other protected APIs such as [Microsoft Graph](#)) in automated integration tests is a challenge. Azure AD often requires an interactive user sign-in prompt, which is difficult to automate. This article describes how you can use a non-interactive flow, called [Resource Owner Password Credential Grant \(ROPC\)](#), to automatically sign in users for testing.

To prepare for your automated integration tests, create some test users, create and configure an app registration, and potentially make some configuration changes to your tenant. Some of these steps require admin privileges. Also, Microsoft recommends that you *do not* use the ROPC flow in a production environment. [Create a separate test tenant](#) that you are an administrator of so you can safely and effectively run your automated integration tests.

WARNING

Microsoft recommends you *do not* use the ROPC flow in a production environment. In most production scenarios, more secure alternatives are available and recommended. The ROPC flow requires a very high degree of trust in the application, and carries risks which are not present in other authentication flows. You should only use this flow for testing purposes in a [separate test tenant](#), and only with test users.

IMPORTANT

- The Microsoft identity platform only supports ROPC within 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 with ROPC, which means features like SMS sign-in, FIDO, and the Authenticator app won't work with that flow.
- 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 Active Directory Federation Services (AD FS) used to authenticate on-premises accounts). If users are full-page redirected to an on-premises identity provider, Azure AD is not able to test the username and password against that identity provider. [Pass-through authentication](#) is supported with ROPC, however.
- An exception to a hybrid identity federation scenario would be the following: Home Realm Discovery policy with [AllowCloudPasswordValidation](#) set to TRUE will enable ROPC flow to work for federated users when on-premises password is synced to cloud. For more information, see [Enable direct ROPC authentication of federated users for legacy applications](#).

Create a separate test tenant

Using the ROPC authentication flow is risky in a production environment, so [create a separate tenant](#) to test your applications. You can use an existing test tenant, but you need to be an admin in the tenant since some of the following steps require admin privileges.

Create and configure a key vault

We recommend you securely store the test usernames and passwords as [secrets](#) in Azure Key Vault. When you

run the tests later, the tests run in the context of a security principal. The security principal is an Azure AD user if you're running tests locally (for example, in Visual Studio or Visual Studio Code), or a service principal or managed identity if you're running tests in Azure Pipelines or another Azure resource. The security principal must have **Read** and **List** secrets permissions so the test runner can get the test usernames and passwords from your key vault. For more information, read [Authentication in Azure Key Vault](#).

1. [Create a new key vault](#) if you don't have one already.
2. Take note of the **Vault URI** property value (similar to `https://<your-unique-keyvault-name>.vault.azure.net/`) which is used in the example test later in this article.
3. [Assign an access policy](#) for the security principal running the tests. Grant the user, service principal, or managed identity **Get** and **List** secrets permissions in the key vault.

Create test users

Create some test users in your tenant for testing. Since the test users are not actual humans, we recommend you assign complex passwords and securely store these passwords as **secrets** in Azure Key Vault.

1. In the [Azure portal](#), select **Azure Active Directory**.
2. Go to **Users**.
3. Select **New user** and create one or more test user accounts in your directory.
4. The example test later in this article uses a single test user. [Add the test username and password as secrets](#) in the key vault you created previously. Add the username as a secret named "TestUserName" and the password as a secret named "TestPassword".

Create and configure an app registration

Register an application that acts as your client app when calling APIs during testing. This should *not* be the same application you may already have in production. You should have a separate app to use only for testing purposes.

Register an application

Create an app registration. You can follow the steps in the [app registration quickstart](#). You don't need to add a redirect URI or add credentials, so you can skip those sections.

Take note of the **Application (client) ID**, which is used in the example test later in this article.

Enable your app for public client flows

ROPC is a public client flow, so you need to enable your app for public client flows. From your app registration in the [Azure portal](#), go to **Authentication > Advanced settings > Allow public client flows**. Set the toggle to **Yes**.

Consent to the permissions you want to use while testing

Since ROPC is not an interactive flow, you won't be prompted with a consent screen to consent to these at runtime. Pre-consent to the permissions to avoid errors when acquiring tokens.

Add the permissions to your app. Do not add any sensitive or high-privilege permissions to the app, we recommend you scope your testing scenarios to basic integration scenarios around integrating with Azure AD.

From your app registration in the [Azure portal](#), go to **API Permissions > Add a permission**. Add the permissions you need to call the APIs you'll be using. A test example further in this article uses the `https://graph.microsoft.com/User.Read` and `https://graph.microsoft.com/User.ReadBasic.All` permissions.

Once the permissions are added, you'll need to consent to them. The way you consent to the permissions depends on if your test app is in the same tenant as the app registration and whether you're an admin in the tenant.

App and app registration are in the same tenant and you're an admin

If you plan on testing your app in the same tenant you registered it in and you are an administrator in that tenant, you can consent to the permissions from the [Azure portal](#). In your app registration in the Azure portal, go to **API Permissions** and select the **Grant admin consent for <your_tenant_name>** button next to the **Add a permission** button and then **Yes** to confirm.

App and app registration are in different tenants, or you're not an admin

If you do not plan on testing your app in the same tenant you registered it in, or you are not an administrator in your tenant, you cannot consent to the permissions from the [Azure portal](#). You can still consent to some permissions, however, by triggering a sign-in prompt in a web browser.

In your app registration in the [Azure portal](#), go to **Authentication > Platform configurations > Add a platform > Web**. Add the redirect URI "https://localhost" and select **Configure**.

There is no way for non-admin users to pre-consent through the Azure portal, so send the following request in a browser. When you are prompted with the login screen, sign in with a test account you created in a previous step. Consent to the permissions you are prompted with. You may need to repeat this step for each API you want to call and test user you want to use.

```
// Line breaks for legibility only

https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize?
client_id={your_client_ID}
&response_type=code
&redirect_uri=https://localhost
&response_mode=query
&scope={resource_you_want_to_call}/.default
&state=12345
```

Replace *{tenant}* with your tenant ID, *{your_client_ID}* with the client ID of your application, and *{resource_you_want_to_call}* with the identifier URI (for example, "https://graph.microsoft.com") or app ID of the API you are trying to access.

Exclude test apps and users from your MFA policy

Your tenant likely has a conditional access policy that [requires multifactor authentication \(MFA\) for all users](#), as recommended by Microsoft. MFA won't work with ROPC, so you'll need to exempt your test applications and test users from this requirement.

To exclude user accounts:

1. Navigate to the [Azure portal](#) and sign in to your tenant. Select **Azure Active Directory**. Select **Security** in the left navigation pane and then select **Conditional access**.
2. In **Policies**, select the conditional access policy that requires MFA.
3. Select **Users or workload identities**.
4. Select the **Exclude** tab and then the **Users and groups** checkbox.
5. Select the user account(s) to exclude in **Select excluded users**.
6. Select the **Select** button and then **Save**.

To exclude a test application:

1. In **Policies**, select the conditional access policy that requires MFA.
2. Select **Cloud apps or actions**.
3. Select the **Exclude** tab and then **Select excluded cloud apps**.
4. Select the app(s) you want to exclude in **Select excluded cloud apps**.
5. Select the **Select** button and then **Save**.

Write your application tests

Now that you're set up, you can write your automated tests. The following .NET example code uses [Microsoft Authentication Library \(MSAL\)](#) and [xUnit](#), a common testing framework.

Set up your appsettings.json file

Add the client ID of the test app you previously created, the necessary scopes, and the key vault URI to the *appsettings.json* file of your test project.

```
{  
    "Authentication": {  
        "AzureCloudInstance": "AzurePublic", //Will be different for different Azure clouds, like US Gov  
        "AadAuthorityAudience": "AzureAdMultipleOrgs",  
        "ClientId": <your_client_ID>  
    },  
  
    "WebAPI": {  
        "Scopes": [  
            //For this Microsoft Graph example. Your value(s) will be different depending on the API you're  
            calling  
            "https://graph.microsoft.com/User.Read",  
            //For this Microsoft Graph example. Your value(s) will be different depending on the API you're  
            calling  
            "https://graph.microsoft.com/User.ReadBasic.All"  
        ]  
    },  
  
    "KeyVault": {  
        "KeyVaultUri": "https://<your-unique-keyvault-name>.vault.azure.net//"  
    }  
}
```

Set up your client for use across all your test classes

Use [SecretClient\(\)](#) to get the test username and password secrets from Azure Key Vault. The code uses exponential back-off for retries in case Key Vault is being throttled.

[DefaultAzureCredential\(\)](#) authenticates with Azure Key Vault by getting an access token from a service principal configured by environment variables or a managed identity (if the code is running on an Azure resource with a managed identity). If the code is running locally, `DefaultAzureCredential` uses the local user's credentials. Read more in the [Azure Identity client library](#) content.

Use Microsoft Authentication Library (MSAL) to authenticate using the ROPC flow and get an access token. The access token is passed along as a bearer token in the HTTP request.

```
using Xunit;  
using System.Threading.Tasks;  
using Microsoft.Identity.Client;  
using System.Security;  
using System.Net;  
using System.Net.Http;  
using System.Net.Http.Headers;  
using Microsoft.Extensions.Configuration;  
using Azure.Identity;  
using Azure.Security.KeyVault.Secrets;  
using Azure.Core;  
using System;  
  
public class ClientFixture : IAsyncLifetime  
{  
    public HttpClient httpClient;  
  
    public async Task InitializeAsync()  
    {  
        var configuration = new ConfigurationBuilder()  
            .AddJsonFile("appsettings.json")  
            .Build();  
  
        var clientSecretCredential = new ClientSecretCredential(  
            configuration["AzureCloudInstance"],  
            configuration["ClientId"],  
            configuration["ClientSecret"]);  
  
        var authenticationProvider = new AuthenticationProvider(clientSecretCredential);  
        httpClient = new HttpClient(authenticationProvider);  
    }  
}
```

```

{
    var builder = new ConfigurationBuilder().AddJsonFile("<path-to-json-file>");

    IConfigurationRoot Configuration = builder.Build();

    var PublicClientApplicationOptions = new PublicClientApplicationOptions();
    Configuration.Bind("Authentication", PublicClientApplicationOptions);
    var app =
PublicClientApplicationBuilder.CreateWithApplicationOptions(PublicClientApplicationOptions)
    .Build();

    SecretClientOptions options = new SecretClientOptions()
    {
        Retry =
        {
            Delay= TimeSpan.FromSeconds(2),
            MaxDelay = TimeSpan.FromSeconds(16),
            MaxRetries = 5,
            Mode = RetryMode.Exponential
        }
    };

    string keyVaultUri = Configuration.GetValue<string>("KeyVault:KeyVaultUri");
    var client = new SecretClient(new Uri(keyVaultUri), new DefaultAzureCredential(), options);

    KeyVaultSecret userNameSecret = client.GetSecret("TestUserName");
    KeyVaultSecret passwordSecret = client.GetSecret("TestPassword");

    string password = passwordSecret.Value;
    string username = userNameSecret.Value;
    string[] scopes = Configuration.GetSection( "WebAPI:Scopes").Get<string[]>();
    SecureString securePassword = new NetworkCredential("", password).SecurePassword;

    AuthenticationResult result = null;
    httpClient = new HttpClient();

    try
    {
        result = await app.AcquireTokenByUsernamePassword(scopes, username, securePassword)
            .ExecuteAsync();
    }
    catch (MsalException) { }

    string accessToken = result.AccessToken;
    httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("bearer", accessToken);
}

public Task DisposeAsync() => Task.CompletedTask;
}

```

Use in your test classes

The following example is a test that calls Microsoft Graph. Replace this test with whatever you'd like to test on your own application or API.

```
public class ApiTests : IClassFixture<ClientFixture>
{
    ClientFixture clientFixture;

    public ApiTests(ClientFixture clientFixture)
    {
        this.clientFixture = clientFixture;
    }

    [Fact]
    public async Task GetRequestTest()
    {
        var testClient = clientFixture.httpClient;
        HttpResponseMessage response = await testClient.GetAsync("https://graph.microsoft.com/v1.0/me");
        var responseCode = response.StatusCode.ToString();
        Assert.Equal("OK", responseCode);
    }
}
```

Throttling and service limits to consider for testing

4/12/2022 • 6 minutes to read • [Edit Online](#)

As a developer, you want to test your application before releasing it to production. When testing applications protected by the Microsoft identity platform, you should set up an Azure Active Directory (Azure AD) environment and tenant to be used for testing.

Applications that integrate with Microsoft identity platform require directory objects (such as app registrations, service principals, groups, and users) to be created and managed in an Azure AD tenant. Any production tenant settings that affect your app's behavior should be replicated in the test tenant. Populate your test tenant with the needed conditional access, permission grant, claims mapping, token lifetime, and token issuance policies. Your application may also use Azure resources such as compute or storage, which need to be added to the test environment. Your test environment may require a lot of resources, depending on the app to be tested.

In order to ensure reliable usage of services by all customers, Azure AD and other services limit the number of resources that can be created per customer and per tenant. When setting up a test environment and deploying directory objects and Azure resources, you may hit some of these service limits and quotas.

Azure AD, Microsoft Graph, and other Azure services also limit the number of concurrent calls to a service or limit the amount of compute load per customer in order to prevent overuse of resources. This is a practice known as throttling and ensures that Azure services can handle usage and incoming requests without service outages. Throttling can occur at the application, tenant, or entire service level. Throttling commonly occurs when an application has a large number of requests within or across tenants. At runtime, your application can read or update Azure AD directory objects through Microsoft Graph as part of its business logic. For example, read or set user attributes, update a user's calendar, or send emails on behalf of the user. While running, your application could also deploy, access, update, and delete Azure resources as well. During testing, your application could hit these runtime throttling limits and the previously mentioned service limits while deploying resources or directory objects.

Azure AD service limits relevant to testing

General Azure AD usage constraints and service limits can be found [here](#). General Azure subscription and service limits, quotas, and constraints can be found [here](#).

The following table lists Azure AD service limits to consider when setting up a test environment or running tests.

CATEGORY	LIMIT
Tenants	A single user can create a maximum of 200 directories.

CATEGORY	LIMIT
Resources	<ul style="list-style-type: none"> A maximum of 50,000 Azure AD resources can be created in a single tenant by users of the Free edition of Azure Active Directory by default. If you have at least one verified domain, the default Azure AD service quota for your organization is extended to 300,000 Azure AD resources. Azure AD service quota for organizations created by self-service sign-up remains 50,000 Azure AD resources even after you performed an internal admin takeover and the organization is converted to a managed tenant with at least one verified domain. This service limit is unrelated to the pricing tier limit of 500,000 resources on the Azure AD pricing page. To go beyond the default quota, you must contact Microsoft Support. A non-admin user can create no more than 250 Azure AD resources. Both active resources and deleted resources that are available to restore count toward this quota. Only deleted Azure AD resources that were deleted fewer than 30 days ago are available to restore. Deleted Azure AD resources that are no longer available to restore count toward this quota at a value of one-quarter for 30 days. If you have developers who are likely to repeatedly exceed this quota in the course of their regular duties, you can create and assign a custom role with permission to create a limitless number of app registrations.
Applications	<ul style="list-style-type: none"> A user, group, or service principal can have a maximum of 1,500 app role assignments. A user can only have a maximum of 48 apps where they have username and password credentials configured.
Application manifest	A maximum of 1200 entries can be added in the Application Manifest.
Groups	<ul style="list-style-type: none"> A non-admin user can create a maximum of 250 groups in an Azure AD organization. Any Azure AD admin who can manage groups in the organization can also create unlimited number of groups (up to the Azure AD object limit). If you assign a role to remove the limit for a user, assign them to a less privileged built-in role such as User Administrator or Groups Administrator. An Azure AD organization can have a maximum of 5000 dynamic groups. A maximum of 300 role-assignable groups can be created in a single Azure AD organization (tenant). Any number of Azure AD resources can be members of a single group. A user can be a member of any number of groups.

CATEGORY	LIMIT
Azure AD roles and permissions	<ul style="list-style-type: none"> • A maximum of 30 Azure AD custom roles can be created in an Azure AD organization. • A maximum of 100 Azure AD custom role assignments for a single principal at tenant scope. • A maximum of 100 Azure AD built-in role assignments for a single principal at non-tenant scope (such as administrative unit or Azure AD object). There is no limit for Azure AD built-in role assignments at tenant scope.

Throttling limits relevant to testing

The following global Microsoft Graph throttling limits apply:

REQUEST TYPE	PER APP ACROSS ALL TENANTS
Request type	Per app across all tenants
Any	2000 requests per second

The following table lists Azure AD throttling limits to consider when running tests. Throttling is based on a token bucket algorithm, which works by adding individual costs of requests. The sum of request costs is then compared against pre-determined limits. Only the requests exceeding the limits will be throttled. For more detailed information on request costs, see [Identity and access service limits](#). Other service-specific limits on Microsoft Graph can be found [here](#).

LIMIT TYPE	RESOURCE UNIT QUOTA	WRITE QUOTA
application+tenant pair	S: 3500, M:5000, L:8000 per 10 seconds	3000 per 2 minutes and 30 seconds
application	150,000 per 20 seconds	70,000 per 5 minutes
tenant	Not Applicable	18,000 per 5 minutes

The application + tenant pair limit varies based on the number of users in the tenant requests are run against. The tenant sizes are defined as follows: S - under 50 users, M - between 50 and 500 users, and L - above 500 users.

What happens when a throttling limit is exceeded?

Throttling behavior can depend on the type and number of requests. For example, if you have a high volume of requests, all requests types are throttled. Threshold limits vary based on the request type. Therefore, you could encounter a scenario where writes are throttled but reads are still permitted.

When you exceed a throttling limit, you receive the HTTP status code `429 Too many requests` and your request fails. The response includes a `Retry-After` header value, which specifies the number of seconds your application should wait (or sleep) before sending the next request. Retry the request. If you send a request before the retry value has elapsed, your request isn't processed and a new retry value is returned. If the request fails again with a 429 error code, you are still being throttled. Continue to use the recommended `Retry-After` delay and retry the request until it succeeds.

Next steps

Learn how to [setup a test environment](#).

Use the portal to create an Azure AD application and service principal that can access resources

4/12/2022 • 8 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 applications, hosted services, or automated tools that need to access or modify resources, you can create an identity for the app. This identity is known as a service principal. Access to resources is restricted by the roles assigned to the service principal, giving you control over which resources can be accessed and at which level. For security reasons, it's always recommended to use service principals with automated tools rather than allowing them to log in with a user identity.

This article shows you how to use the portal to create the service principal in the Azure portal. 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. You can also [use Azure PowerShell to create a service principal](#).

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?](#).

App registration, app objects, and service principals

There is no way to directly create a service principal using the Azure portal. When you register an application through the Azure portal, an application object and service principal are automatically created in your home directory or tenant. For more information on the relationship between app registration, application objects, and service principals, read [Application and service principal objects in Azure Active Directory](#).

Permissions required for registering an app

You must have sufficient permissions to register an application with your Azure AD tenant, and assign to the application a role in your Azure subscription.

Check Azure AD permissions

1. Select **Azure Active Directory**.
2. Find your role under **Overview->My feed**. If you have the **User** role, you must make sure that non-administrators can register applications.

Overview

Manage tenants | What's new | Preview features | Got feedback? ▾

Overview Monitoring Tutorials

Search your tenant

Basic information

Name	Fourth Coffee	Users
Tenant ID		Groups
Primary domain	fourthcoffeetest.onmicrosoft.com	Applications
License	Azure AD Free	Devices

My feed

Test User
f85e06fe-...
User
[More info](#)

3. In the left pane, select **Users** and then **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 [Azure AD built-in roles](#) to learn about available administrator roles and the specific permissions in Azure AD that are given to each role. If your account is assigned the User role, but the app registration setting is limited to admin users, ask your administrator to either assign you 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 a role to an AD app. This action is granted through the **Owner** role or **User Access Administrator** role. If your account is assigned the **Contributor** role, you don't have adequate permission. You will receive an error when attempting to assign the service principal a role.

To check your subscription permissions:

1. 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' and contains a list of items: 'Event Grid Subscriptions', 'Resource groups', 'Manage subscriptions in the Billing/Account Center', and 'APEX C+L - Aquent Vendor Subscriptions'. Below this is a section for 'Resource Groups' which says 'No results were found.' At the bottom of the main content area, there's a 'Documentation' section with links to scaling with multiple subscriptions, Azure subscription limits and quotas, and creating an additional Azure subscription. The 'Subscriptions' icon in the bottom navigation bar is also highlighted with a red box.

2. Select the subscription you want to create the service principal in.

The screenshot shows the 'Subscriptions' page in the Microsoft Azure portal. It lists one subscription: 'Internal testing subscription'. The 'Internal testing subscription' row is highlighted with a red box. The page includes a search bar, a dropdown for 'My role', and an 'Apply' button. There are also filters for 'Show only subscriptions selected in the global subscriptions filter' and a 'Search to filter items...' field.

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 **My permissions**. Then, select [Click here to view complete access details for this subscription](#).

The screenshot shows the Azure portal interface for managing subscriptions. On the left, there's a sidebar titled 'Subscriptions' under 'Microsoft' with a '+ Add' button. Below it, a message says 'Showing subscriptions in Microsoft. Don't see a subscription? Switch directories'. A filter section shows '8 selected' and '3 selected' with an 'Apply' button. A search bar and sorting options ('Subscription na... ↑', 'Subscription ID ↑') are also present. The main area is titled 'Internal testing subscription - My permissions' and contains a 'Resource provider status' section with a message: 'You are an administrator on the subscription Internal testing subscription Click here to view complete access details for this subscription.' A red box highlights the 'My permissions' link at the bottom of the main content area.

4. Select **Role assignments** to view your assigned roles, and determine if you have adequate permissions to assign a role to an AD app. If not, ask your subscription administrator to add you to User Access Administrator role. In the following image, the user is assigned the Owner role, which means that user has adequate permissions.

The screenshot shows the 'Role assignments' page. At the top, there are buttons for '+ Add', 'Download role assignments', 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?'. Below that, tabs include 'Check access' (disabled), 'Role assignments' (highlighted with a red box), 'Roles', 'Deny assignments', and 'Classic administrators'. A heading 'Number of role assignments for this subscription' shows 1 assignment. A search bar and filters for 'Type : All', 'Role : All', 'Scope : All scopes', and 'Group by : Role' are present. The main table lists one item: '1 items (1 Users)'. The table columns are 'Name', 'Type', 'Role', and 'Scope'. One row shows 'Owner' as the name, 'User' as the type, 'Owner' as the role (highlighted with a red box), and 'This resource' as the scope. The user 'Test Admin' has a green profile icon with 'TA' initials.

Register an application with Azure AD and create a service principal

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

* 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 (Fourth Coffee only - Single tenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)
- Personal Microsoft accounts only

[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.



Register an app you're working on here. Integrate gallery apps and other apps from outside your organization by adding from [Enterprise applications](#).

By proceeding, you agree to the [Microsoft Platform Policies](#) ↗

[Register](#)

You've created your Azure AD application and service principal.

NOTE

You can register multiple applications with the same name in Azure AD, but the applications must have different Application (client) IDs.

Assign a role to the application

To access resources in your subscription, you must assign a role to the application. Decide which role offers the right permissions for the application. To learn about the available roles, see [Azure 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. The main content area is titled 'Subscriptions'. It shows a list of subscriptions, with the first one, 'cephalin320170403020701', selected. Below the list are links for documentation and a search bar. At the bottom, there are navigation links 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 Azure portal. It lists a single subscription named 'Internal testing subscription'. There are filters for 'My role' (set to '8 selected') and 'Show only subscriptions selected in the global subscriptions filter' (unchecked). A search bar is also present. The table has columns for 'SUBSCRIPTION' and 'SUBSCRIPTION ID'.

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 **Select Add > Add role assignment** to open the **Add role assignment** page.
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.

Assign the Contributor role to the application at the subscription scope. For detailed steps, see [Assign Azure roles using the Azure portal](#).

Your service principal is set up. You can start using it to run your scripts or apps. To manage your service principal (permissions, user consented permissions, see which users have consented, review permissions, see sign in information, and more), go to [Enterprise applications](#).

The next section shows how to get values that are needed when signing in programmatically.

Get tenant and app ID values for signing in

When programmatically signing in, pass the tenant ID with your authentication request and the application ID. You also need a certificate or an authentication key (described in the following section). 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	:	10233211-1234-4567-89ab-000000000000
Directory (tenant) ID	:	77f41007-0e11-42d2-84d9-000000000000
Object ID	:	12345678-9abc-4def-5678-000000000000

Copy to clipboard

The directory (tenant) ID can also be found in the default directory overview page.

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	:	10233211-1234-4567-89ab-000000000000
Directory (tenant) ID	:	77f41007-0e11-42d2-84d9-000000000000
Object ID	:	12345678-9abc-4def-5678-000000000000

Copy to clipboard

Authentication: Two options

There are two types of authentication available for service principals: password-based authentication (application secret) and certificate-based authentication. *We recommend using a certificate*, but you can also create an application secret.

Option 1: Upload a certificate

You can use an existing certificate if you have one. Optionally, you can create a self-signed certificate for *testing purposes only*. To create a self-signed certificate, open PowerShell and run [New-SelfSignedCertificate](#) with the following parameters to create the cert 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.

1. Select **Run** from the **Start** menu, and then enter **certmgr.msc**.

The Certificate Manager tool for the current user appears.

2. To view your certificates, under **Certificates - Current User** in the left pane, expand the **Personal** directory.

3. Right-click on the cert you created, select **All tasks->Export**.

4. Follow the Certificate Export wizard. Do not export the private key, and export to a .CER file.

To upload the certificate:

1. Select **Azure Active Directory**.
2. From **App registrations** in Azure AD, select your application.
3. Select **Certificates & secrets**.
4. Select **Certificates > 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.		

5. Select **Add**.

After registering the certificate with your application in the application registration portal, enable the client application code to use the certificate.

Option 2: Create a new application secret

If you choose not to use a certificate, you can create a new application secret.

1. Select **Azure Active Directory**.
2. From **App registrations** in Azure AD, select your application.
3. Select **Certificates & secrets**.
4. Select **Client secrets -> New client secret**.
5. Provide a description of the secret, and a duration. When done, select **Add**.

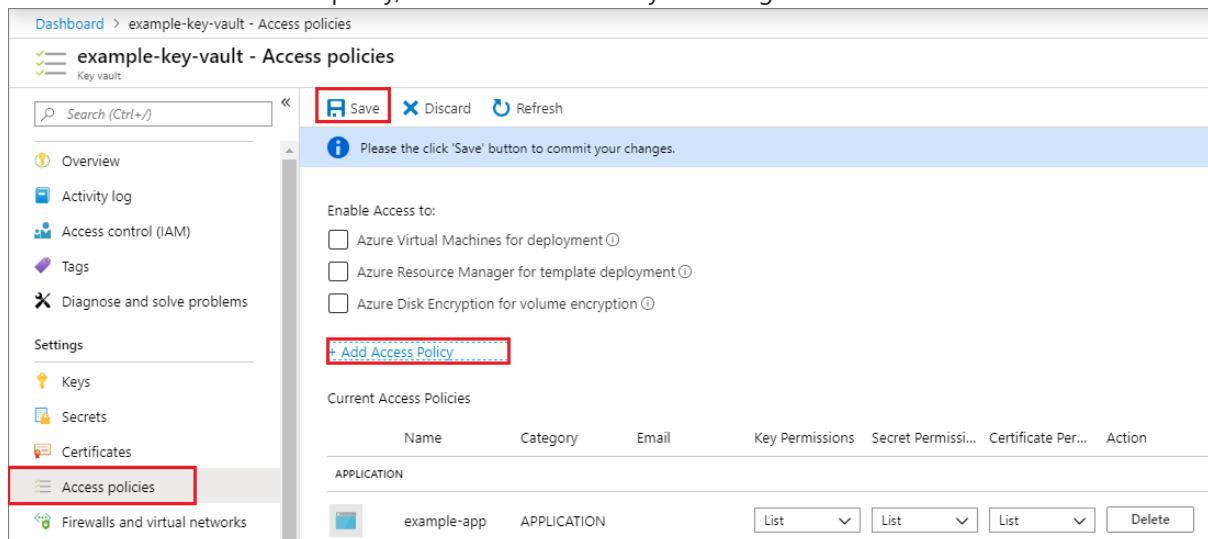
After saving the client secret, the value of the client secret is displayed. Copy this value because you won't be able to retrieve the key later. You will provide the key value with the application ID to sign in as the application. Store the key value where your application can retrieve it.

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 
		

Configure access policies on resources

Keep in mind, you might need to configure additional 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. On the left, there's a sidebar with 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Settings' (with 'Keys', 'Secrets', 'Certificates'), and 'Access policies' (which is selected and highlighted with a red box). The main area has a 'Save' button at the top right, a message 'Please click 'Save' button to commit your changes.', and a section titled 'Enable Access to:' with three checkboxes for 'Azure Virtual Machines for deployment', 'Azure Resource Manager for template deployment', and 'Azure Disk Encryption for volume encryption'. Below this is a 'Current Access Policies' table with one row for 'example-app' (Category: APPLICATION). At the bottom are 'List' dropdowns and a 'Delete' button.

Next steps

- Learn how to [use Azure PowerShell](#) to create a service principal.
- To learn about specifying security policies, see [Azure role-based access control \(Azure RBAC\)](#).
- For a list of available actions that can be granted or denied to users, see [Azure Resource Manager Resource Provider operations](#).
- For information about working with app registrations by using [Microsoft Graph](#), see the [Applications API](#) reference.

Use Azure PowerShell to create a service principal with a certificate

4/12/2022 • 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 uses the Azure Az PowerShell module, which is the recommended PowerShell module for interacting with Azure. To get started with the Az PowerShell module, see [Install Azure PowerShell](#). To learn how to migrate to the Az PowerShell module, see [Migrate Azure PowerShell from AzureRM to Az](#).

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 [Azure 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-AzRoleAssignment](#) 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, download the [New-SelfSignedCertificateEx cmdlet](#) from PKI Solutions. 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, 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, 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](#).
- For information about working with app registrations by using **Microsoft Graph**, see the [Applications API](#) reference.

Sign in any Azure Active Directory user using the multi-tenant application pattern

4/12/2022 • 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 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 the sample [Build a multi-tenant SaaS web application that calls Microsoft Graph using Azure AD and OpenID Connect](#).

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.

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 the 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 the Microsoft identity platform.

NOTE

While native client applications request and receive tokens from the Microsoft identity platform, they do so to send them to APIs, where they are validated. Native applications do not validate access tokens and must treat them as opaque.

Let's look at how an application validates tokens it receives from the 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](https://sts.windows.net/31537af4-6d77-4bb9-a681-d2394888ea26/), 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.

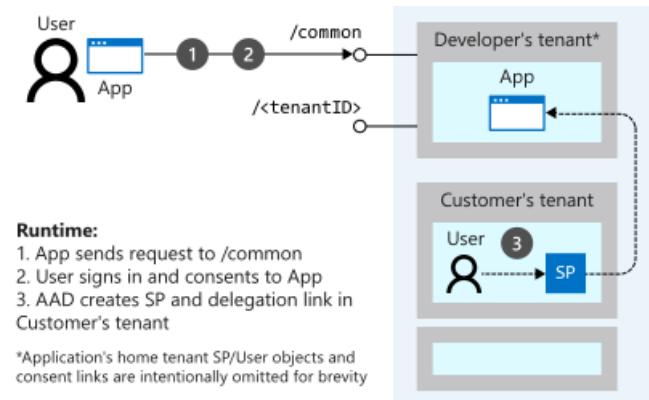
For example, if a multi-tenant application only allows sign-in from specific tenants who have signed up for their service, then it must check either the issuer value or the `tid` claim value in the token to make sure that tenant is in their list of subscribers. If a multi-tenant application only deals 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 easier; 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. The 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.

To learn more about user and admin consent, see [Configure the admin consent workflow](#).

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 [Microsoft Graph API](#) indicate which permissions require admin consent.

If your application uses permissions that require admin consent, 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.

Consent and multi-tier applications

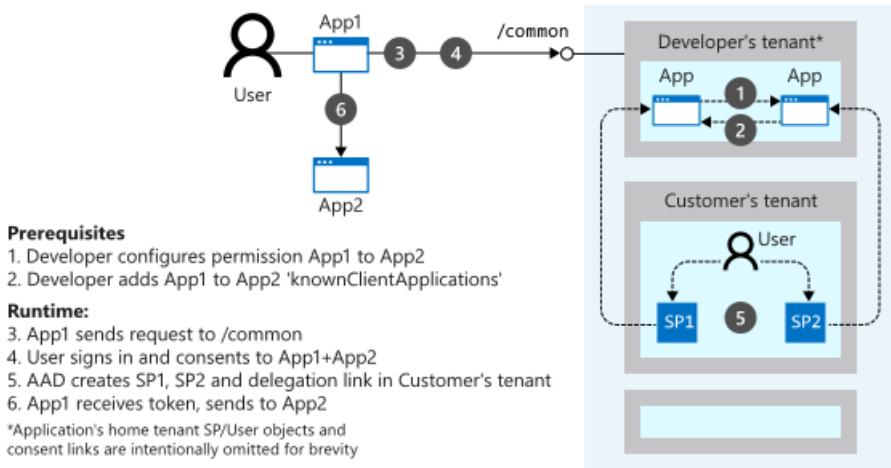
Your application may have multiple tiers, each represented by its own registration in Azure AD. For example, a native application that calls a web API, or a web application that calls a web API. In both of these cases, the client (native app or web app) requests permissions to call the resource (web 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.



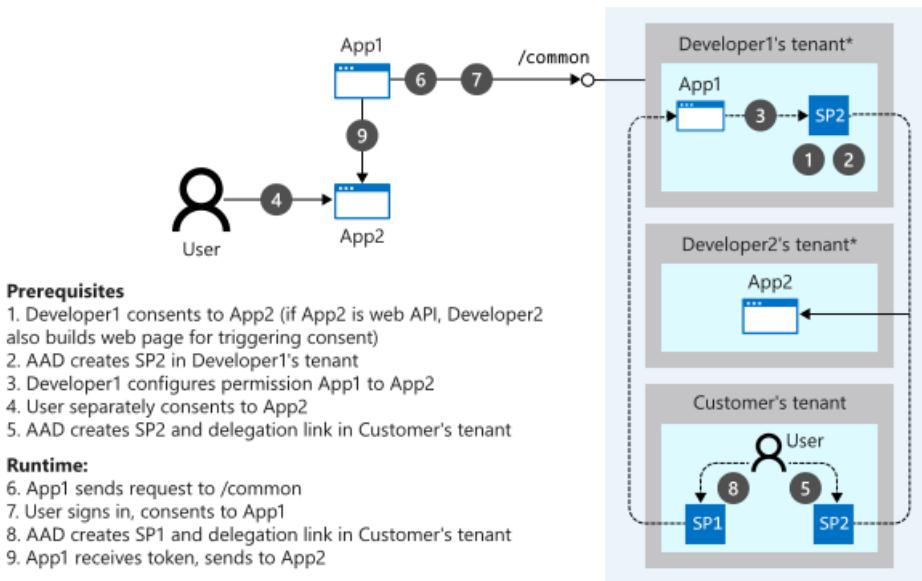
Multiple tiers in multiple tenants

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 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 Microsoft Authentication Library (MSAL) 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, MSAL 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.

Related content

- [Multi-tenant application sample](#)
- [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](#)

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 Microsoft 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 Microsoft 365 services like Exchange, SharePoint, OneDrive, OneNote, and more, visit [Microsoft Graph API](#).

Modify the accounts supported by an application

4/12/2022 • 2 minutes to read • [Edit Online](#)

When you registered your application with the Microsoft identity platform, you specified who--which account types--can access it. For example, you might've specified accounts only in your organization, which is a *single-tenant* app. Or, you might've specified accounts in any organization (including yours), which is a *multi-tenant* app.

In the following sections, you learn how to modify your app's registration in the Azure portal to change who, or what types of accounts, can access the application.

Prerequisites

- An application registered in your Azure AD tenant

Change the application registration to support different accounts

To specify a different setting for the account types supported by an existing app registration:

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which the app is registered.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations**, then select your application.
5. Now, specify who can use the application, sometimes referred to as the *sign-in audience*.

SUPPORTED ACCOUNT TYPES	DESCRIPTION
Accounts in this organizational directory only	Select this option if you're building an application for use only by users (or guests) in <i>your</i> tenant. Often called a <i>line-of-business</i> (LOB) application, this is a single-tenant application in the Microsoft identity platform.
Accounts in any organizational directory	Select this option if you'd like users in <i>any</i> Azure AD tenant to be able to use your application. This option is appropriate if, for example, you're building a software-as-a-service (SaaS) application that you intend to provide to multiple organizations. This is known as a multi-tenant application in the Microsoft identity platform.

6. Select **Save**.

Why changing to multi-tenant can fail

Switching an app registration from single- to multi-tenant can sometimes fail due to Application ID URI (App ID URI) name collisions. An example App ID URI is `https://contoso.onmicrosoft.com/myapp`.

The App ID URI is one of the ways an application is identified in protocol messages. For a single-tenant

application, the App ID URI need only be unique within that tenant. For a multi-tenant application, it must be globally unique so Azure AD can find the app across all tenants. Global uniqueness is enforced by requiring that the App ID URI's host name matches one of the Azure AD tenant's [verified publisher domains](#).

For example, if the name of your tenant is *contoso.onmicrosoft.com*, then

`https://contoso.onmicrosoft.com/myapp` is a valid App ID URI. 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 the second pattern, `https://contoso.com/myapp`, converting the app registration to multi-tenant fails.

For more information about configuring a verified publisher domain, see [Configure a verified domain](#).

Next steps

Learn more about the requirements for [converting an app from single- to multi-tenant](#).

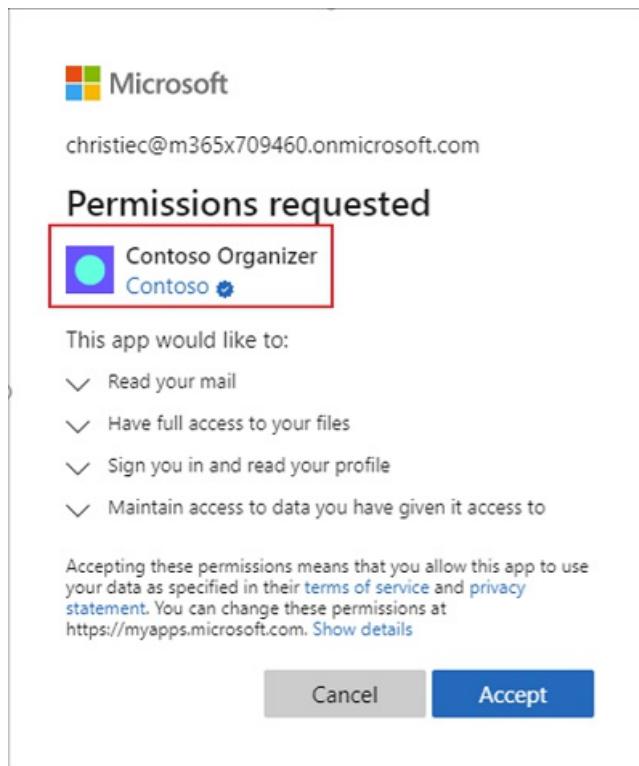
Publisher verification

4/12/2022 • 4 minutes to read • [Edit Online](#)

Publisher verification helps admins and end users understand the authenticity of application developers integrating with the Microsoft identity platform.

When an application is marked as publisher verified, it means that the publisher has verified their identity using a [Microsoft Partner Network](#) account that has completed the [verification](#) process and has associated this MPN account with their application registration.

A blue "verified" badge appears on the Azure AD consent prompt and other screens:



NOTE

We recently changed the color of the "verified" badge from blue to gray. We will revert that change sometime in the last half of February 2022, so the "verified" badge will be blue.

This feature is primarily for developers building multi-tenant apps that leverage [OAuth 2.0 and OpenID Connect](#) with the [Microsoft identity platform](#). These apps can sign users in using OpenID Connect, or they may use OAuth 2.0 to request access to data using APIs like [Microsoft Graph](#).

Benefits

Publisher verification provides the following benefits:

- **Increased transparency and risk reduction for customers**- this capability helps customers understand which apps being used in their organizations are published by developers they trust.
- **Improved branding**- a "verified" badge appears on the Azure AD [consent prompt](#), Enterprise Apps

page, and additional UX surfaces used by end users and admins.

- **Smoother enterprise adoption**- admins can configure [user consent policies](#), with publisher verification status as one of the primary policy criteria.

NOTE

- Starting in November 2020, end users will no longer be able to grant consent to most newly registered multi-tenant apps without verified publishers if [risk-based step-up consent](#) is enabled. This will apply to apps that are registered after November 8, 2020, use OAuth2.0 to request permissions beyond basic sign-in and read user profile, and request consent from users in different tenants than the one the app is registered in. A warning will be displayed on the consent screen informing users that these apps are risky and are from unverified publishers.
- Publisher verification is not supported in national clouds. Applications registered in national cloud tenants can't be publisher-verified at this time.

Requirements

There are a few pre-requisites for publisher verification, some of which will have already been completed by many Microsoft partners. They are:

- An MPN ID for a valid [Microsoft Partner Network](#) account that has completed the [verification](#) process. This MPN account must be the [Partner global account \(PGA\)](#) for your organization.
- An app registered in an Azure AD tenant, with a [Publisher Domain](#) configured.
- The domain of the email address used during MPN account verification must either match the publisher domain configured on the app or a DNS-verified [custom domain](#) added to the Azure AD tenant.
- The user performing verification must be authorized to make changes to both the app registration in Azure AD and the MPN account in Partner Center.
 - In Azure AD this user must be a member of one of the following [roles](#): Application Admin, Cloud Application Admin, or Global Admin.
 - In Partner Center this user must have one of the following [roles](#): MPN Admin, Accounts Admin, or a Global Admin (this is a shared role mastered in Azure AD).
- The user performing verification must sign in using [multi-factor authentication](#).
- The publisher agrees to the [Microsoft identity platform for developers Terms of Use](#).

Developers who have already met these pre-requisites can get verified in a matter of minutes. If the requirements have not been met, getting set up is free.

Frequently asked questions

Below are some frequently asked questions regarding the publisher verification program. For FAQs related to the requirements and the process, see [mark an app as publisher verified](#).

- **What information does publisher verification not provide?** When an application is marked publisher verified this does not indicate whether the application or its publisher has achieved any specific certifications, complies with industry standards, adheres to best practices, etc. Other Microsoft programs do provide this information, including [Microsoft 365 App Certification](#).
- **How much does this cost? Does it require any license?** Microsoft does not charge developers for publisher verification and it does not require any specific license.
- **How does this relate to Microsoft 365 Publisher Attestation? What about Microsoft 365 App**

Certification? These are complementary programs that developers can use to create trustworthy apps that can be confidently adopted by customers. Publisher verification is the first step in this process, and should be completed by all developers creating apps that meet the above criteria.

Developers who are also integrating with Microsoft 365 can receive additional benefits from these programs. For more information, refer to [Microsoft 365 Publisher Attestation](#) and [Microsoft 365 App Certification](#).

- **Is this the same thing as the Azure AD Application Gallery?** No- publisher verification is a complementary but separate program to the [Azure Active Directory application gallery](#). Developers who fit the above criteria should complete the publisher verification process independently of participation in that program.

Next steps

- Learn how to [mark an app as publisher verified](#).
- [Troubleshoot](#) publisher verification.

Configure an application's publisher domain

4/12/2022 • 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](#).
2. If you have access to multiple tenants, use the **Directory + subscription** filter  in the top menu to select the tenant in which the app is registered.
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. Under **Manage**, select the **Branding**.
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 domain isn't registered in the 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}"  
        },  
        {  
            "applicationId": "{YOUR-OTHER-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.

You're not required to maintain the resources that are used for verification after a domain has been verified. When the verification is finished, you can remove the hosted file.

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 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 Azure AD 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.

Mark your app as publisher verified

4/12/2022 • 2 minutes to read • [Edit Online](#)

When an app registration has a verified publisher, it means that the publisher of the app has [verified](#) their identity using their Microsoft Partner Network (MPN) account and has associated this MPN account with their app registration. This article describes how to complete the [publisher verification](#) process.

Quickstart

If you are already enrolled in the Microsoft Partner Network (MPN) and have met the [pre-requisites](#), you can get started right away:

1. Sign into the [App Registration portal](#) using [multi-factor authentication](#)
2. Choose an app and click **Branding**.
3. Click **Add MPN ID to verify publisher** and review the listed requirements.
4. Enter your MPN ID and click **Verify and save**.

For more details on specific benefits, requirements, and frequently asked questions see the [overview](#).

Mark your app as publisher verified

Make sure you have met the [pre-requisites](#), then follow these steps to mark your app(s) as Publisher Verified.

1. Ensure you are signed in using [multi-factor authentication](#) to an organizational (Azure AD) account that is authorized to make changes to the app(s) you want to mark as Publisher Verified and on the MPN Account in Partner Center.
 - In Azure AD this user must be a member of one of the following [roles](#): Application Admin, Cloud Application Admin, or Global Admin.
 - In Partner Center this user must have of the following [roles](#): MPN Admin, Accounts Admin, or a Global Admin (this is a shared role mastered in Azure AD).
2. Navigate to the **App registrations** blade:
3. Click on an app you would like to mark as Publisher Verified and open the **Branding** blade.
4. Ensure the app's [publisher domain](#) is set.
5. Ensure that either the publisher domain or a DNS-verified [custom domain](#) on the tenant matches the domain of the email address used during the verification process for your MPN account.
6. Click **Add MPN ID to verify publisher** near the bottom of the page.
7. Enter your **MPN ID**. This MPN ID must be for:
 - A valid Microsoft Partner Network account that has completed the verification process.
 - The Partner global account (PGA) for your organization.
8. Click **Verify and save**.
9. Wait for the request to process, this may take a few minutes.

10. If the verification was successful, the publisher verification window will close, returning you to the Branding blade. You will see a blue verified badge next to your verified **Publisher display name**.
11. Users who get prompted to consent to your app will start seeing the badge soon after you have gone through the process successfully, although it may take some time for this to replicate throughout the system.
12. Test this functionality by signing into your application and ensuring the verified badge shows up on the consent screen. If you are signed in as a user who has already granted consent to the app, you can use the *prompt=consent* query parameter to force a consent prompt. This parameter should be used for testing only, and never hard-coded into your app's requests.
13. Repeat this process as needed for any additional apps you would like the badge to be displayed for. You can use Microsoft Graph to do this more quickly in bulk, and PowerShell cmdlets will be available soon. See [Making Microsoft API Graph calls](#) for more info.

That's it! Let us know if you have any feedback about the process, the results, or the feature in general.

Next steps

If you run into problems, read the [troubleshooting information](#).

Troubleshoot publisher verification

4/12/2022 • 8 minutes to read • [Edit Online](#)

If you're unable to complete the process or are experiencing unexpected behavior with [publisher verification](#), you should start by doing the following if you're receiving errors or seeing unexpected behavior:

1. Review the [requirements](#) and ensure they've all been met.
2. Review the instructions to [mark an app as publisher verified](#) and ensure all steps have been performed successfully.
3. Review the list of [common issues](#).
4. Reproduce the request using [Graph Explorer](#) to gather more info and rule out any issues in the UI.

Common Issues

Below are some common issues that may occur during the process.

- **I don't know my Microsoft Partner Network ID (MPN ID) or I don't know who the primary contact for the account is**
 1. Navigate to the [MPN enrollment page](#)
 2. Sign in with a user account in the org's primary Azure AD tenant
 3. If an MPN account already exists, this will be recognized and you'll be added to the account
 4. Navigate to the [partner profile page](#) where the MPN ID and primary account contact will be listed
- **I don't know who my Azure AD Global Administrator (also known as company admin or tenant admin) is, how do I find them? What about the Application Administrator or Cloud Application Administrator?**
 1. Sign in to the [Azure AD Portal](#) using a user account in your organization's primary tenant
 2. Navigate to [Role Management](#)
 3. Select the desired admin role
 4. The list of users assigned that role will be displayed
- **I don't know who the admin(s) for my MPN account are** Go to the [MPN User Management page](#) and filter the user list to see what users are in various admin roles.
- **I am getting an error saying that my MPN ID is invalid or that I do not have access to it.**
 1. Go to your [partner profile](#) and verify that:
 - The MPN ID is correct.
 - There are no errors or "pending actions" shown, and the verification status under Legal business profile and Partner info both say "authorized" or "success".
 2. Go to the [MPN tenant management page](#) and confirm that the tenant the app is registered in and that you're signing with a user account from is on the list of associated tenants. To add another tenant, follow the instructions [here](#). Be aware that all Global Admins of any tenant you add will be granted Global Admin privileges on your Partner Center account.
 3. Go to the [MPN User Management page](#) and confirm the user you're signing in as is either a Global Admin, MPN Admin, or Accounts Admin. To add a user to a role in Partner Center, follow the instructions [here](#).
- **When I sign into the Azure AD portal, I do not see any apps registered. Why?** Your app

registrations may have been created using a different user account in this tenant, a personal/consumer account, or in a different tenant. Ensure you're signed in with the correct account in the tenant where your app registrations were created.

- I'm getting an error related to multi-factor authentication. What should I do? Ensure [multi-factor authentication](#) is enabled and [required](#) for the user you're signing in with and for this scenario. For example, MFA could be:

- Always required for the user you're signing in with
- [Required for Azure management](#).
- [Required for the type of administrator](#) you're signing in with.

Making Microsoft Graph API calls

If you're having an issue but unable to understand why based on what you are seeing in the UI, it may be helpful to perform further troubleshooting by using Microsoft Graph calls to perform the same operations you can perform in the App Registration portal.

The easiest way to make these requests is to use [Graph Explorer](#). You may also consider other options like using [Postman](#), or using PowerShell to [invoke a web request](#).

You can use Microsoft Graph to both set and unset your app's verified publisher and check the result after performing one of these operations. The result can be seen on both the [application](#) object corresponding to your app registration and any [service principals](#) that have been instantiated from that app. For more information on the relationship between those objects, see: [Application and service principal objects in Azure Active Directory](#).

Here are examples of some useful requests:

Set Verified Publisher

Request

```
POST /applications/0cd04273-0d11-4e62-9eb3-5c3971a7cbec/setVerifiedPublisher
{
    "verifiedPublisherId": "12345678"
}
```

Response

```
204 No Content
```

NOTE

verifiedPublisherID is your MPN ID.

Unset Verified Publisher

Request:

```
POST /applications/0cd04273-0d11-4e62-9eb3-5c3971a7cbec/unsetVerifiedPublisher
```

Response

204 No Content

Get Verified Publisher info from Application

```
GET https://graph.microsoft.com/v1.0/applications/0cd04273-0d11-4e62-9eb3-5c3971a7cbec

HTTP/1.1 200 OK

{
  "id": "0cd04273-0d11-4e62-9eb3-5c3971a7cbec",
  ...
  "verifiedPublisher" : {
    "displayName": "myexamplePublisher",
    "verifiedPublisherId": "12345678",
    "addedDateTime": "2019-12-10T00:00:00"
  }
}
```

Get Verified Publisher info from Service Principal

```
GET https://graph.microsoft.com/v1.0/servicePrincipals/010422a7-4d77-4f40-9335-b81ef5c23dd4

HTTP/1.1 200 OK

{
  "id": "010422a7-4d77-4f40-9335-b81ef5c22dd4",
  ...
  "verifiedPublisher" : {
    "displayName": "myexamplePublisher",
    "verifiedPublisherId": "12345678",
    "addedDateTime": "2019-12-10T00:00:00"
  }
}
```

Error Reference

The following is a list of the potential error codes you may receive, either when troubleshooting with Microsoft Graph or going through the process in the app registration portal.

MPNAccountNotFoundOrNoAccess

The MPN ID you provided (`MPNID`) doesn't exist, or you don't have access to it. Provide a valid MPN ID and try again.

Most commonly caused by the signed-in user not being a member of the proper role for the MPN account in Partner Center- see [requirements](#) for a list of eligible roles and see [common issues](#) for more information. Can also be caused by the tenant the app is registered in not being added to the MPN account, or an invalid MPN ID.

MPNGlobalAccountNotFound

The MPN ID you provided (`MPNID`) isn't valid. Provide a valid MPN ID and try again.

Most commonly caused when an MPN ID is provided which corresponds to a Partner Location Account (PLA). Only Partner Global Accounts are supported. See [Partner Center account structure](#) for more details.

MPNAccountInvalid

The MPN ID you provided (`MPNID`) isn't valid. Provide a valid MPN ID and try again.

Most commonly caused by the wrong MPN ID being provided.

MPNAccountNotVetted

The MPN ID (`MPNID`) you provided hasn't completed the vetting process. Complete this process in Partner Center and try again.

Most commonly caused by when the MPN account hasn't completed the [verification](#) process.

NoPublisherIdOnAssociatedMPNAccount

The MPN ID you provided (`MPNID`) isn't valid. Provide a valid MPN ID and try again.

Most commonly caused by the wrong MPN ID being provided.

MPNIdDoesNotMatchAssociatedMPNAccount

The MPN ID you provided (`MPNID`) isn't valid. Provide a valid MPN ID and try again.

Most commonly caused by the wrong MPN ID being provided.

ApplicationNotFound

The target application (`AppId`) can't be found. Provide a valid application ID and try again.

Most commonly caused when verification is being performed via Graph API, and the ID of the application provided is incorrect. Note- the ID of the application must be provided, not the AppId/ClientId.

B2CTenantNotAllowed

This capability isn't supported in an Azure AD B2C tenant.

EmailVerifiedTenantNotAllowed

This capability isn't supported in an email verified tenant.

NoPublisherDomainOnApplication

The target application (`AppId`) must have a Publisher Domain set. Set a Publisher Domain and try again.

Occurs when a [Publisher Domain](#) isn't configured on the app.

PublisherDomainMismatch

The target application's Publisher Domain (`publisherDomain`) doesn't match the domain used to perform email verification in Partner Center (`pcDomain`). Ensure these domains match and try again.

Occurs when neither the app's [Publisher Domain](#) nor one of the [custom domains](#) added to the Azure AD tenant match the domain used to perform email verification in Partner Center.

NotAuthorizedToVerifyPublisher

You aren't authorized to set the verified publisher property on application (< `AppId` >)

Most commonly caused by the signed-in user not being a member of the proper role for the MPN account in Azure AD- see [requirements](#) for a list of eligible roles and see [common issues](#) for more information.

MPNIdWasNotProvided

The MPN ID wasn't provided in the request body or the request content type wasn't "application/json".

MSANotSupported

This feature isn't supported for Microsoft consumer accounts. Only applications registered in Azure AD by an Azure AD user are supported.

InteractionRequired

Occurs when multi-factor authentication hasn't been performed before attempting to add a verified publisher to the app. See [common issues](#) for more information. Note: MFA must be performed in the same session when attempting to add a verified publisher. If MFA is enabled but not required to be performed in the session, the request will fail.

The error message displayed will be: "Due to a configuration change made by your administrator, or because you moved to a new location, you must use multi-factor authentication to proceed."

UnableToAddPublisher

One of these error messages are displayed: "A verified publisher can't be added to this application. Contact your administrator for assistance.", or "You're unable to add a verified publisher to this application. Contact your administrator for assistance."

First, verify you've met the [publisher verification requirements](#).

NOTE

If you've met the publisher verification requirements and are still having issues, try using an existing or newly created user with similar permissions.

When a request to add a verified publisher is made, many signals are used to make a security risk assessment. If the request is determined to be risky an error will be returned. For security reasons, Microsoft doesn't disclose the specific criteria used to determine whether a request is risky or not. If you received this error and believe the "risky" assessment is incorrect, try waiting and resubmitting the verification request. Some customers have reported success after multiple attempts.

Next steps

If you've reviewed all of the previous information and are still receiving an error from Microsoft Graph, gather as much of the following information as possible related to the failing request and [contact Microsoft support](#).

- Timestamp
- CorrelationId
- ObjectID or UserPrincipalName of signed in user
- ObjectId of target application
- AppId of target application
- TenantId where app is registered
- MPN ID
- REST request being made
- Error code and message being returned

Sign in with Microsoft: Branding guidelines for applications

4/12/2022 • 5 minutes to read • [Edit Online](#)

When developing applications with the Microsoft identity platform, 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
- Learn the requirements for using the Microsoft logo 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 Microsoft 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 are mastered 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 Microsoft 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.

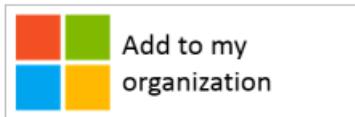
If your app supports end-user sign-up (for example, free to trial or freemium model): You can show a **sign-in** button that allows users to access your app with their work account or their personal account. Azure AD will show a consent prompt the first time they access your app.

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’ Microsoft 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 Microsoft 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

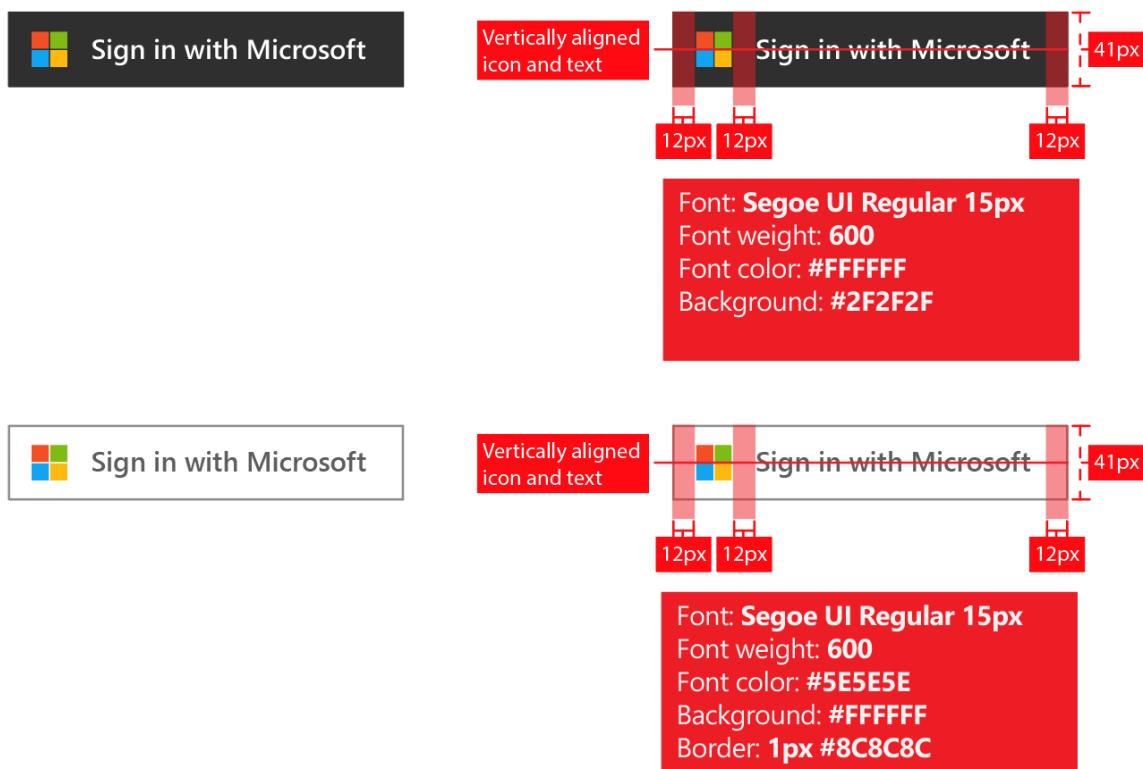
Your app should display a sign-in button that redirects users to the sign-in endpoint that corresponds to the protocol you use to integrate with Azure AD. The following section provides details on what that button should look like.

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.



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 "Microsoft 365 ID" or "Azure ID." Microsoft 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.

Configure terms of service and privacy statement for an app

4/12/2022 • 2 minutes to read • [Edit Online](#)

Developers who build and manage multi-tenant 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

- The terms of service and privacy statement links are not applicable to single-tenant apps
- If one or both of the two links are missing, your app will show an alert.

User consent experience

The following example shows the user consent experience for a multi-tenant app when the terms of service and privacy statement are configured and when these links are not configured.



testuser2@fourthcoffeetest.onmicrosoft.com

Permissions requested

SignInUserTest20210914164947
[App info](#)

This application is not published by Microsoft.

This app would like to:

- ✓ Maintain access to data you have given it access to
- ✓ 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>

Does this app look suspicious? [Report it here](#)

[Cancel](#) [Accept](#)



testuser2@fourthcoffeetest.onmicrosoft.com

Permissions requested

SignInUserTest20210914164947
[unverified](#)

This application is not published by Microsoft.

This app would like to:

- ✓ Maintain access to data you have given it access to
- ✓ 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>

Does this app look suspicious? [Report it here](#)

[Cancel](#) [Accept](#)

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/privacystatement>

Adding links to the terms of service and privacy statement

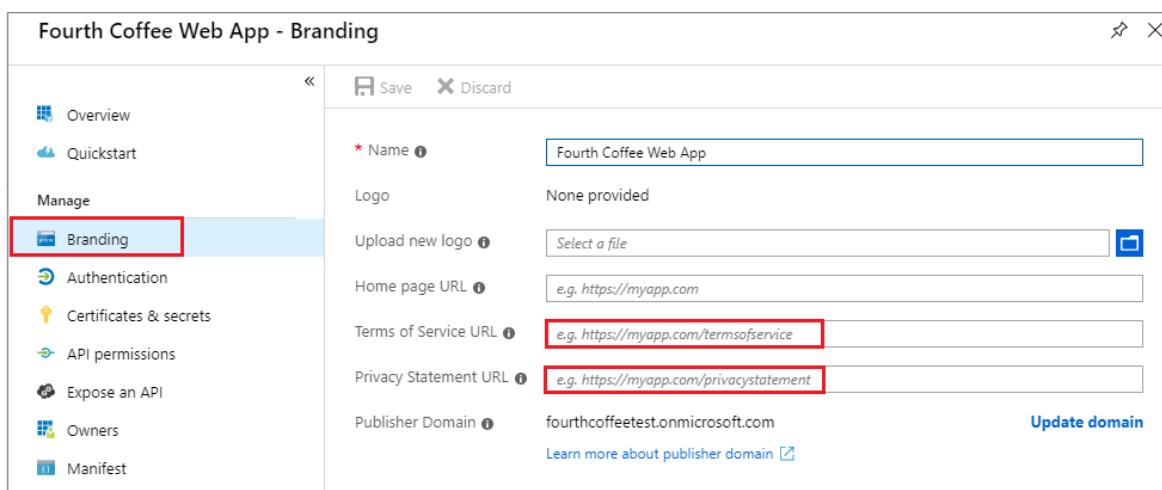
When the terms of service and privacy statement are ready, you can add links to these documents in your app using one of these methods:

- [Through the Azure portal](#)
- [Using the app object JSON](#)
- [Using the Microsoft Graph API](#)

Using the Azure portal

Follow these steps in the Azure portal.

1. Sign in to the [Azure portal](#) and select the correct Azure AD tenant(not B2C).
2. Navigate to the **App registrations** section and select your app.
3. Under **Manage**, select **Branding**.
4. Fill out the **Terms of service URL** and **Privacy statement URL** fields.
5. Select **Save**.



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.

1. Navigating to the **App Registrations** section and select your app.

2. Open the **Manifest** pane.
3. Ctrl+F, Search for "informationalUrls". Fill in the information.
4. Save your changes.

```
"informationalUrls": {  
    "termsOfService": "<your_terms_of_service_url>",  
    "privacy": "<your_privacy_statement_url>"  
}
```

Using the Microsoft Graph API

To programmatically update all your apps, you can use the Microsoft Graph API to update all your apps to include links to the terms of service and privacy statement documents.

```
PATCH https://graph.microsoft.com/v1.0/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 Microsoft Graph API only works when you sign in with an Azure AD account. Personal Microsoft accounts are not supported.

Get AppSource certified for Azure Active Directory

4/12/2022 • 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.

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](#).

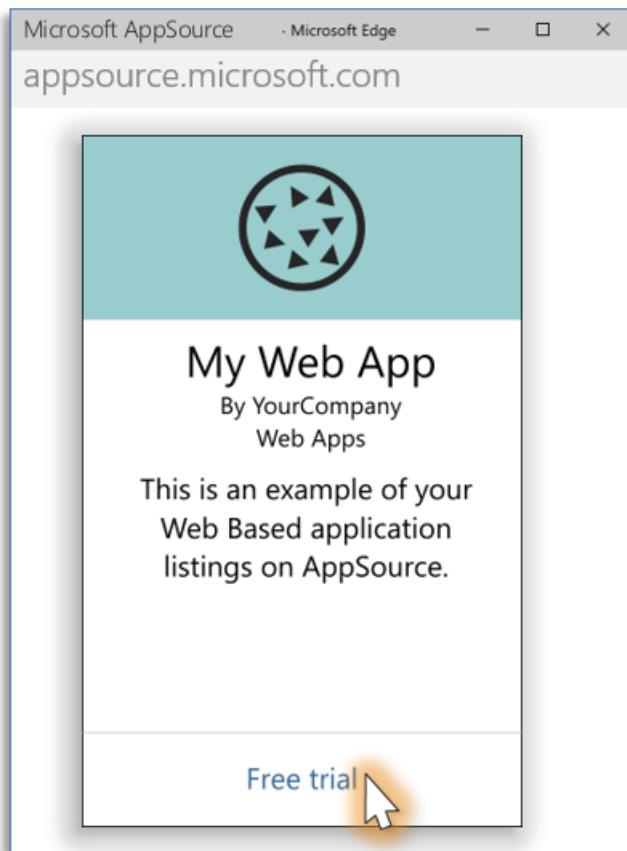
Single-tenant applications can enable the *Contact Me* experience, but if you want to enable the single-click/free trial experience that AppSource recommends, enable multi-tenancy on your application instead.

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:

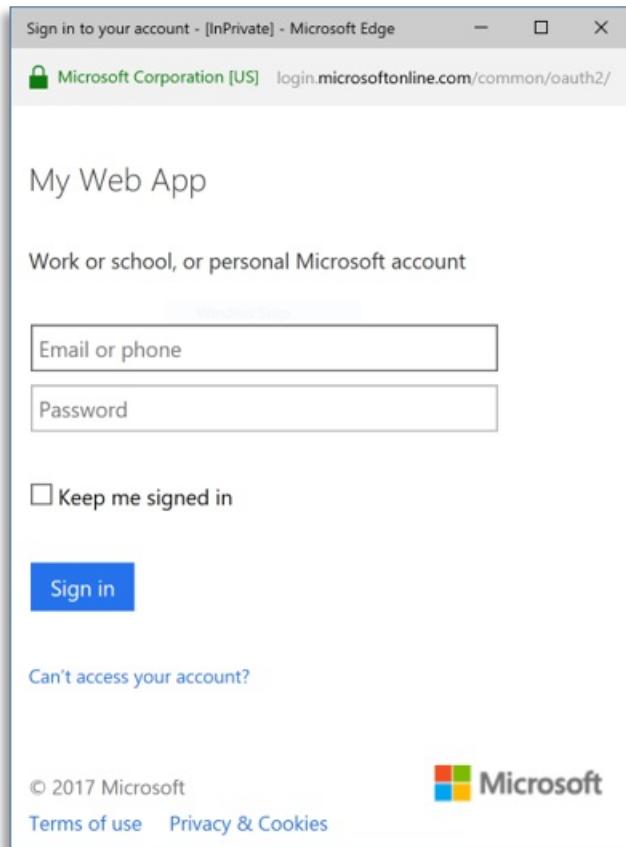
1. A user finds your application in the AppSource web site, then selects **Free trial** option.



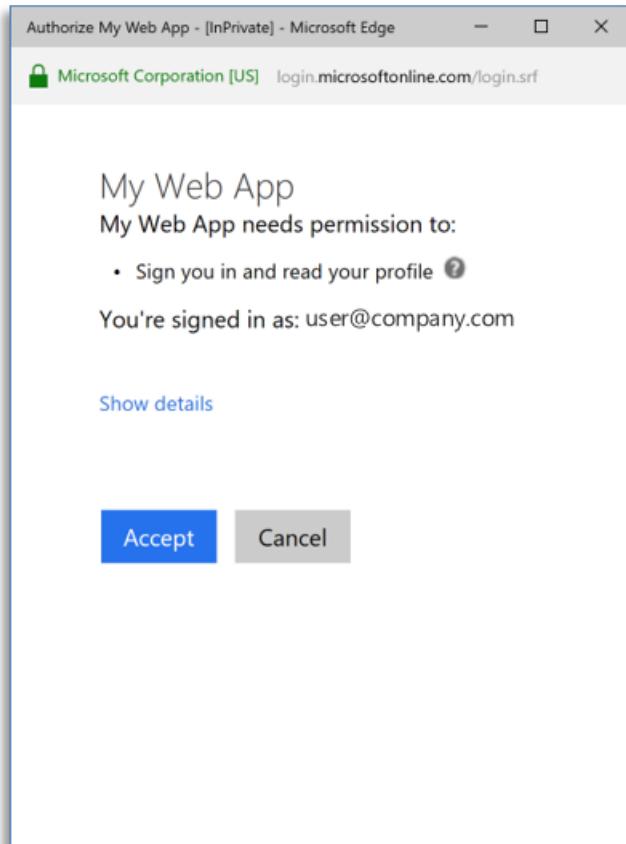
2. AppSource redirects the user to a URL in your web site. Your web site starts the *single-sign-on* process automatically (on page load).



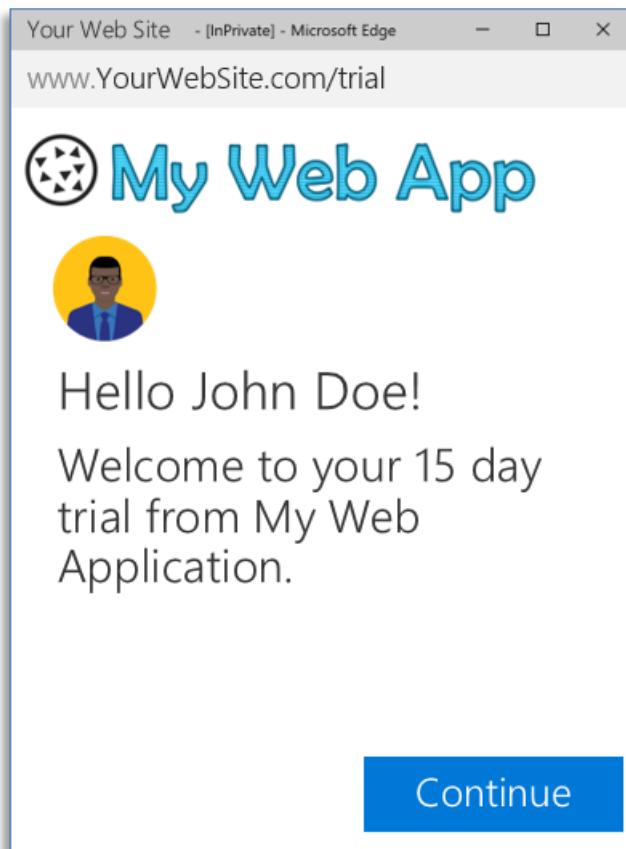
3. The user is redirected to the Microsoft sign-in page and the user provides credentials to sign in.



4. The user gives consent for your application.



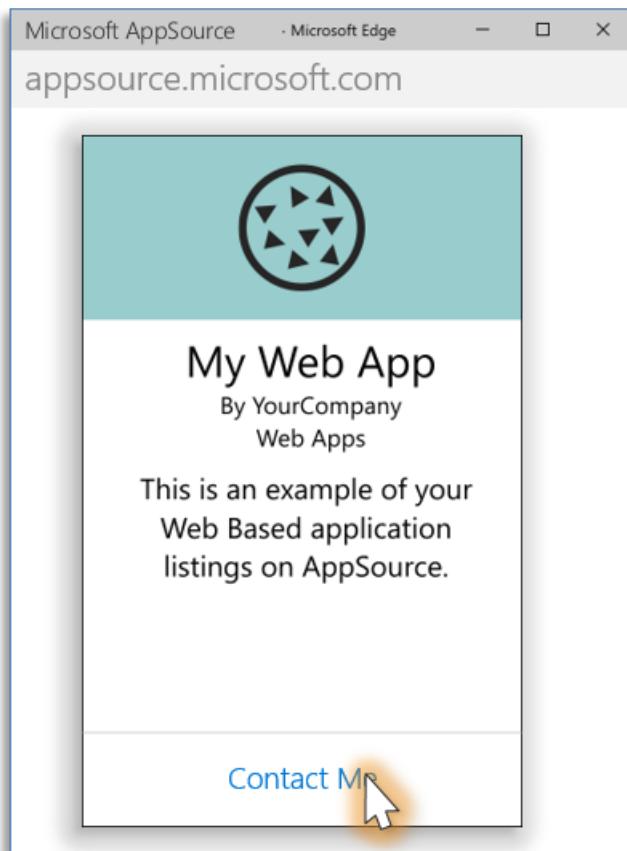
5. Sign-in completes and the user is redirected back to your web site. The user starts the free trial.



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:

1. A user finds your application in AppSource web site, then selects **Contact Me**.



2. The user fills out a form with contact information.

A screenshot of a Microsoft Edge browser window showing a contact form titled "Share your contact information" for "My Web App" by "YourCompany". The form includes fields for First name (John), Last name (Doe), Work email (john@contoso.com), Job title (empty), Company (Contoso), Country / region (empty), and Phone number (212-321-1122). Below the form is a note about sharing contact information with Microsoft and a "Send" button.

3. You receive the user's information, set up a trial instance, and send the hyperlink to access your application to the user.



4. The user accesses your application and completes the single sign-on process.

Sign in to your account - [InPrivate] - Microsoft Edge

Microsoft Corporation [US] login.microsoftonline.com/common/oauth2/

My Web App

Work or school, or personal Microsoft account

Email or phone

Password

Keep me signed in

Sign in

Can't access your account?

© 2017 Microsoft

Microsoft

Terms of use Privacy & Cookies

5. The user gives consent for your application.

Authorize My Web App - [InPrivate] - Microsoft Edge

Microsoft Corporation [US] login.microsoftonline.com/login.srf

My Web App

My Web App needs permission to:

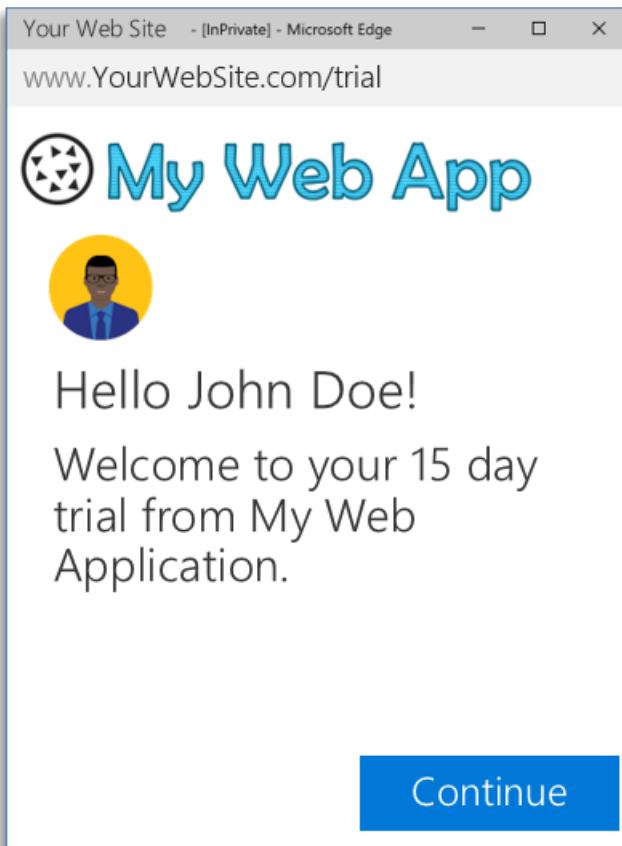
- Sign you in and read your profile ?

You're signed in as: user@company.com

Show details

Accept **Cancel**

6. Sign-in completes and the user is redirected back to your web site. The user starts the free trial.



More information

For more information about the AppSource trial experience, see [this video](#).

Get support

For Azure AD integration, we use [Microsoft Q&A](#) with the community to provide support.

We highly recommend you ask your questions on [Microsoft Q&A](#) first and browse existing issues to see if someone has asked your question before. Make sure that your questions or comments are tagged with [\[azure-active-directory\]](#).

Use the following comments section to provide feedback and help us refine and shape our content.

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](#)

Publish your application in the Azure Active Directory application gallery

4/12/2022 • 5 minutes to read • [Edit Online](#)

You can publish your application in the Azure Active Directory (Azure AD) application gallery. When your application is published, it's made available as an option for users when they add applications to their tenant. For more information, see [Overview of the Azure Active Directory application gallery](#).

To publish your application in the gallery, you need to complete the following tasks:

- Make sure that you complete the prerequisites.
- Create and publish documentation.
- Submit your application.
- Join the Microsoft partner network.

Prerequisites

- To publish your application in the gallery, you must first read and agree to specific [terms and conditions](#).
- Every application in the gallery must implement one of the supported single sign-on (SSO) options. To learn more about the supported options, see [Plan a single sign-on deployment](#). To learn more about authentication, see [Authentication vs. authorization](#) and [Azure active Directory code samples](#). For password SSO, make sure that your application supports form authentication so that password vaulting can be used. For a quick introduction about single sign-on configuration in the portal, see [Enable single sign-on for an enterprise application](#).
- For federated applications (OpenID and SAML/WS-Fed), the application must support the [software-as-a-service \(SaaS\) model](#) to be listed in the gallery. The enterprise gallery applications must support multiple user configurations and not any specific user.
- 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 user 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.
- Supporting provisioning is optional, but highly recommended. Provisioning must be done using the System for Cross-domain Identity Management (SCIM) protocol, which is easy to implement. Using SCIM allows users to automatically create and update accounts in your application without relying on manual processes such as uploading CSV files. To learn more about the Azure AD SCIM implementation, see [build a SCIM endpoint and configure user provisioning with Azure AD](#).

You can get a free test account with all the premium Azure AD features - 90 days free and can get extended as long as you do dev work with it: [Join the Microsoft 365 Developer Program](#).

Create and publish documentation

Documentation on your site

Ease of adoption is a significant factor in enterprise software decisions. Clear easy-to-follow documentation supports your users in their adoption journey and reduces support costs.

Your documentation should at a minimum include the following items:

- Introduction to your SSO functionality

- Protocols supported
- Version and SKU
- Supported identity providers list with documentation links
- Licensing information for your application
- Role-based access control for configuring SSO
- SSO Configuration Steps
 - UI configuration elements for SAML with expected values from the provider
 - Service provider information to be passed to identity providers
- If OIDC/OAuth, list of permissions required for consent with business justifications
- Testing steps for pilot users
- Troubleshooting information, including error codes and messages
- Support mechanisms for users
- Details about your SCIM endpoint, including the resources and attributes supported

Documentation on the Microsoft site

When your application is added to the gallery, documentation is created that explains the step-by-step process. For an example, see [Tutorials for integrating SaaS applications with Azure Active Directory](#). This documentation is created based on your submission to the gallery, and you can easily update it if you make changes to your application using your GitHub account.

Submit your application

After you've tested that your application integration works with Azure AD, submit your application request in the [Microsoft Application Network portal](#). The first time you try to sign into the portal you are presented with one of two screens.

- If you receive the message "That didn't work", then you need to contact the [Azure AD SSO Integration Team](#). Provide the email account that you want to use for submitting the request. A business email address such as `name@yourbusiness.com` is preferred. The Azure AD team will add the account in the Microsoft Application Network portal.
- If you see a "Request Access" page, then fill in the business justification and select **Request Access**.

After the account is added, you can sign in to the Microsoft Application Network portal and submit the request by selecting the **Submit Request (ISV)** tile on the home page. If you see the **Your sign-in was blocked** error while logging in, see [Troubleshoot sign-in to the Microsoft Application Network portal](#).

Implementation-specific options

On the Application Registration Form, select the feature that you want to enable. Select **OpenID Connect & OAuth 2.0, SAML 2.0/WS-Fed, or Password SSO(UserName & Password)** depending on the feature that your application supports.

If you're implementing a **SCIM** 2.0 endpoint for user provisioning, select **User Provisioning (SCIM 2.0)**. Download the schema to provide in the onboarding request. For more information, see [Export provisioning configuration and roll back to a known good state](#). The schema that you configured is used when testing the non-gallery application to build the gallery application.

You can track application requests by customer name at the Microsoft Application Network portal. For more information, see [Application requests by Customers](#).

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

The timeline for the process of listing a SCIM provisioning application in the gallery is variable and depends on numerous factors.

Not all applications can be onboarded. Per the terms and conditions, the choice may be made to not list an application. Onboarding applications is at the sole discretion of the onboarding team. If your application is declined, you should use the non-gallery provisioning application to satisfy your provisioning needs.

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

For any escalations, send email to the [Azure AD SSO Integration Team](#), and a response is sent as soon as possible.

Join the Microsoft partner network

The Microsoft Partner Network provides instant access to exclusive resources, programs, tools, and connections. To join the network and create your go to market plan, see [Reach commercial customers](#).

Next steps

- Learn more about managing enterprise applications in [What is application management in Azure Active Directory?](#)

Migrate applications to the Microsoft Authentication Library (MSAL)

4/12/2022 • 2 minutes to read • [Edit Online](#)

If any of your applications use the Azure Active Directory Authentication Library (ADAL) for authentication and authorization functionality, it's time to migrate them to the [Microsoft Authentication Library \(MSAL\)](#).

- All Microsoft support and development for ADAL, including security fixes, ends in December, 2022.
- There are no ADAL feature releases or new platform version releases planned prior to December, 2022.
- No new features have been added to ADAL since June 30, 2020.

WARNING

If you choose not to migrate to MSAL before ADAL support ends in December, 2022, you put your app's security at risk. Existing apps that use ADAL will continue to work after the end-of-support date, but Microsoft will no longer release security fixes on ADAL.

Why switch to MSAL?

MSAL provides multiple benefits over ADAL, including the following features:

FEATURES	MSAL	ADAL
Security		
Security fixes beyond December, 2022	✓	✗
Proactively refresh and revoke tokens based on policy or critical events for Microsoft Graph and other APIs that support Continuous Access Evaluation (CAE) .	✓	✗
Standards compliant with OAuth v2.0 and OpenID Connect (OIDC)	✓	✗
User accounts and experiences		
Azure Active Directory (Azure AD) accounts	✓	✓
Microsoft account (MSA)	✓	✗
Azure AD B2C accounts	✓	✗
Best single sign-on experience	✓	✗
Resilience		

FEATURES	MSAL	ADAL
Proactive token renewal	✓	✗
Throttling	✓	✗

AD FS support in MSAL.NET

You can use MSAL.NET, MSAL Java, and MSAL Python to get tokens from Active Directory Federation Services (AD FS) 2019 or later. Earlier versions of AD FS, including AD FS 2016, are unsupported by MSAL.

If you need to continue using AD FS, you should upgrade to AD FS 2019 or later before you update your applications from ADAL to MSAL.

How to migrate to MSAL

Before you start the migration, you need to identify which of your apps are using ADAL for authentication. Follow the steps in this article to get a list by using the Azure portal:

- [How to: Get a complete list of apps using ADAL in your tenant](#)

After identifying your apps that use ADAL, migrate them to MSAL depending on your application type as illustrated below.

Single-page app (SPA)

- [ADAL.js to MSAL.js](#)

Web app

- [ADAL Node to MSAL Node](#)
- [ADAL.NET to MSAL.NET](#)

Web API

- [ADAL Java to MSAL Java](#)
- [ADAL Python to MSAL Python](#)
- [ADAL.NET to MSAL.NET](#)

Desktop app

- [ADAL Java to MSAL Java](#)
- [ADAL Python to MSAL Python](#)
- [ADAL.NET to MSAL.NET](#)

Mobile app

- [ADAL.Android to MSAL.Android](#)
- [ADAL.iOS to MSAL.iOS](#)
- [Xamarin Android using brokers to MSAL.NET](#)
- [Xamarin iOS using brokers to MSAL.NET](#)

Service / daemon app

- [ADAL Python to MSAL Python](#)
- [ADAL.NET to MSAL.NET](#)

- [ADAL Node to MSAL Node](#)
- [ADAL Java to MSAL Java](#)

Migration help

If you have questions about migrating your app from ADAL to MSAL, here are some options:

- Post your question on [Microsoft Q&A](#) and tag it with `[azure-ad-adal-deprecation]`.
- Open an issue in the library's GitHub repository. See the [Languages and frameworks](#) section of the MSAL overview article for links to each library's repo.

If you partnered with an Independent Software Vendor (ISV) in the development of your application, we recommend that you contact them directly to understand their migration journey to MSAL.

Next steps

For more information about MSAL, including usage information and which libraries are available for different programming languages and application types, see:

- [Acquire and cache tokens using MSAL](#)
- [Application configuration options](#)
- [MSAL authentication libraries](#)

Get a complete list of apps using ADAL in your tenant

4/12/2022 • 2 minutes to read • [Edit Online](#)

Support for Active Directory Authentication Library (ADAL) will end in December, 2022. Apps using ADAL on existing OS versions will continue to work, but technical support and security updates will end. Without continued security updates, apps using ADAL will become increasingly vulnerable to the latest security attack patterns. For more information, see [Migrate apps to MSAL](#). This article provides guidance on how to use Azure Monitor workbooks to obtain a list of all apps that use ADAL in your tenant.

Sign-ins workbook

Workbooks are a set of queries that collect and visualize information that is available in Azure Active Directory (Azure AD) logs. [Learn more about the sign-in logs schema here](#). The Sign-ins workbook in the Azure AD admin portal now has a table to assist you in determining which applications use ADAL and how often they are used. First, we'll detail how to access the workbook before showing the visualization for the list of applications.

Step 1: Send Azure AD sign-in events to Azure Monitor

Azure AD doesn't send sign-in events to Azure Monitor by default, which the Sign-ins workbook in Azure Monitor requires.

Configure AD to send sign-in events to Azure Monitor by following the steps in [Integrate your Azure AD sign-in and audit logs with Azure Monitor](#). In the **Diagnostic settings** configuration step, select the **SignInLogs** check box.

No sign-in event that occurred *before* you configure Azure AD to send the events to Azure Monitor will appear in the Sign-ins workbook.

Step 2: Access sign-ins workbook in Azure portal

Once you've integrated your Azure AD sign-in and audit logs with Azure Monitor as specified in the Azure Monitor integration, access the sign-ins workbook:

1. Sign into the Azure portal
2. Navigate to **Azure Active Directory > Monitoring > Workbooks**
3. In the **Usage** section, open the **Sign-ins** workbook

The screenshot shows the Microsoft Azure Workbooks Gallery. On the left, a sidebar lists various Azure services: Password reset, Company branding, User settings, Properties, Security, Monitoring, Sign-ins, Audit logs, Provisioning logs (Preview), Logs, Diagnostic settings, and Workbooks (which is selected and highlighted with a red border). The main area displays two sections: 'Public Templates (16)' and 'Workbooks (24)'. In the 'Public Templates' section, the 'Sign-ins' template is selected and highlighted with a red border. The 'Workbooks' section lists four entries: 'Risky IP Workbooks', 'Device Insights', 'Auth Methods Activi...', and 'RiskyIPReport2'. At the top right, there are filters for 'Subscript...', 'Resource Group : All', and 'Reset filters'.

Step 3: Identify apps that use ADAL

The table at the bottom of the Sign-ins workbook page lists apps that recently used ADAL. You can also export a list of the apps. Update these apps to use MSAL.

The screenshot shows the 'Sign-ins' workbook page. The sidebar on the left is identical to the previous one, with 'Workbooks' selected. The main content area has a title 'Sign-ins by Device' and a chart showing sign-in counts for Windows 10, MacOs, Android, Windows, and Android 8. To the right, there is a 'Device Sign-in details' pane with a 'User' list containing Colleen Gihon, Brian Huntley, Aisha Wang, Adam Steenwyk, and Adrian Drumea. Below the chart, a section titled 'Sign-ins from apps that use Active Directory Authentication Libraries (ADAL)' provides information about the retirement of ADAL and a table of affected apps. The table has columns: App Name, App ID, ADAL Version, and Sign-in Count. It lists three apps: Sample App 1, Sample App 2, and Sample App 3, all using ADAL.Js 1.0.17. The row for Sample App 3 is highlighted with a red border.

If there are no apps using ADAL, the workbook will display a view as shown below.

The screenshot shows the Azure Active Directory Sign-ins page for the 'Woodgrove' tenant. The left sidebar includes options like Password reset, Company branding, User settings, Properties, Security, Monitoring, Sign-ins, Audit logs, Provisioning logs (Preview), Logs, Diagnostic settings, Workbooks (selected), Usage & insights, Troubleshooting + Support, Virtual assistant (Preview), and New support request.

The main area displays 'Sign-ins by Device' with a table and a chart. The table shows sign-in counts for various devices:

Name	Sign-in Count	Trend
Windows 10	9,607K	
MacOs	231	
Android	114	
Windows	43	
Android 8	32	

The 'Device Sign-in details' section lists users with their sign-in counts:

User
Colleen Ghion
Brian Huntley
Aisha Wang
Aisha Wang
Adam Steenwyk
Adrian Drumea

A note at the bottom states: "We've released the newer Microsoft Authentication Library (MSAL), thus the older Active Directory Authentication Library (ADAL) is being retired on June 30th, 2022. We are no longer adding new features to ADAL and will no longer add any security patches starting June 30th, 2022. We recommend updating the following apps to MSAL by making changes to the app source code. [Learn more about the migration steps here.](#)" A red box highlights this note.

A message box indicates: "The query returned no results."

Step 4: Update your code

After identifying your apps that use ADAL, migrate them to MSAL depending on your application type as illustrated below.

Single-page app (SPA)

- [ADAL.js to MSAL.js](#)

Web app

- [ADAL Node to MSAL Node](#)
- [ADAL.NET to MSAL.NET](#)

Web API

- [ADAL Java to MSAL Java](#)
- [ADAL Python to MSAL Python](#)
- [ADAL.NET to MSAL.NET](#)

Desktop app

- [ADAL Java to MSAL Java](#)
- [ADAL Python to MSAL Python](#)
- [ADAL.NET to MSAL.NET](#)

Mobile app

- [ADAL.Android to MSAL.Android](#)
- [ADAL.iOS to MSAL.iOS](#)
- [Xamarin Android using brokers to MSAL.NET](#)
- [Xamarin iOS using brokers to MSAL.NET](#)

Service / daemon app

- [ADAL Python to MSAL Python](#)
- [ADAL.NET to MSAL.NET](#)
- [ADAL Node to MSAL Node](#)

- [ADAL Java to MSAL Java](#)

Next steps

For more information about MSAL, including usage information and which libraries are available for different programming languages and application types, see:

- [Acquire and cache tokens using MSAL](#)
- [Application configuration options](#)
- [List of MSAL authentication libraries](#)

Remove an application registered with the Microsoft identity platform

4/12/2022 • 2 minutes to read • [Edit Online](#)

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

In the following sections, you learn how to:

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

Prerequisites

- An [application registered in your Azure AD tenant](#)

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](#).

NOTE

Deleting an application will also delete its service principal object in the application's home directory. For multi-tenant applications, service principal objects in other directories will not be deleted.

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

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directory + subscription** filter  in the top menu to select the tenant in which the app is registered.
3. Search and select the [Azure Active Directory](#).
4. Under **Manage**, select **App registrations** and select the application that you want to configure. Once you've selected the app, you'll see the application's [Overview](#) page.
5. From the [Overview](#) page, select **Delete**.
6. Read the deletion consequences. Check the box if one appears at the bottom of the pane.
7. Select **Delete** to confirm that you want to delete the app.

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 Administrator access, and

can remove the application through the Azure portal or use the [Azure AD PowerShell Cmdlets](#) to remove access.

Next steps

Learn more about [application and service principal objects](#) in the Microsoft identity platform.

Restore or remove a recently deleted application with the Microsoft identity platform

4/12/2022 • 2 minutes to read • [Edit Online](#)

After you delete an app registration, the app remains in a suspended state for 30 days. During that 30-day window, the app registration can be restored, along with all its properties. After that 30-day window passes, app registrations cannot be restored and the permanent deletion process may be automatically started. This functionality only applies to applications associated to a directory. It is not available for applications from a personal Microsoft account, which cannot be restored.

You can view your deleted applications, restore a deleted application, or permanently delete an application using the App registrations experience under Azure Active Directory (Azure AD) in the Azure portal.

Note that neither you nor Microsoft customer support can restore a permanently deleted application or an application deleted more than 30 days ago.

IMPORTANT

The deleted applications portal UI feature is in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Some features might be unsupported or have constrained capabilities. For more information, see [Supplemental terms of use for Microsoft Azure previews](#).

Required permissions

You must have one of the following roles to permanently delete applications.

- Global administrator
- Application administrator
- Cloud application administrator
- Hybrid identity administrator
- Application owner

You must have one of the following roles to restore applications.

- Global administrator
- Application owner

View your deleted applications

You can see all the applications in a soft deleted state. Only applications deleted less than 30 days ago can be restored.

To view your restorable applications

1. Sign in to the [Azure portal](#).
2. Search and select **Azure Active Directory**, select **App registrations**, and then select the **Deleted applications (Preview)** tab.

Review the list of applications. Only applications that have been deleted in the past 30 days are available to restore. If using the App registrations search preview, you can filter by the 'Deleted date' column to see only these applications.

Restore a recently deleted application

When an app registration is deleted from the organization, the app is in a suspended state and its configurations are preserved. When you restore an app registration, its configurations are also restored. However, if there were any organization-specific settings in **Enterprise applications** for the application's home tenant, those will not be restored.

This is because organization-specific settings are stored on a separate object, called the service principal. Settings held on the service principal include permission consents and user and group assignments for a certain organization; these configurations will not be restored when the app is restored. For more information, see [Application and service principal objects](#).

To restore an application

1. On the **Deleted applications (Preview)** tab, search for and select one of the applications deleted less than 30 days ago.
2. Select **Restore app registration**.

Permanently delete an application

You can manually permanently delete an application from your organization. A permanently deleted application can't be restored by you, another administrator, or by Microsoft customer support.

To permanently delete an application

1. On the **Deleted applications (Preview)** tab, search for and select one of the available applications.
2. Select **Delete permanently**.
3. Read the warning text and select **Yes**.

Next steps

After you've restored or permanently deleted your app, you can:

- [Add an application](#).
- Learn more about [application and service principal objects](#) in the Microsoft identity platform.

Quickstart: Sign in users in single-page apps (SPA) via the auth code flow

4/12/2022 • 13 minutes to read • [Edit Online](#)

In this quickstart, you download and run a code sample that demonstrates how a JavaScript Angular single-page application (SPA) can sign in users and call Microsoft Graph using the authorization code flow. The code sample demonstrates how to get an access token to call the Microsoft Graph API or any web API.

See [How the sample works](#) for an illustration.

This quickstart uses MSAL Angular v2 with the authorization code flow.

Prerequisites

- Azure subscription - [Create an Azure subscription for free](#)
- [Node.js](#)
- [Visual Studio Code](#) or another code editor

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. Go to the [Azure portal - App registrations](#) quickstart experience.
2. Enter a name for your application.
3. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
4. Select **Register**.
5. Go to the quickstart pane and 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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application. Users of your app might see this name, and you can change it later.
6. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
7. Select **Register**. On the app **Overview** page, note the **Application (client) ID** value for later use.
8. Under **Manage**, select **Authentication**.
9. Under **Platform configurations**, select **Add a platform**. In the pane that opens select **Single-page application**.
10. Set the **Redirect URIs** value to `http://localhost:4200/`. This is the default port NodeJS will listen on your local machine. We'll return the authentication response to this URI after successfully authenticating the user.

11. Select **Configure** to apply the changes.
12. Under **Platform Configurations** expand **Single-page application**.
13. Confirm that under **Grant types** Your Redirect URI is eligible for the Authorization Code Flow with PKCE.

Step 2: Download the project

To run the project with a web server by using Node.js, [download the core project files](#).

Step 3: Configure your JavaScript app

In the `src` folder, open the `app` folder then open the `app.module.ts` file and update the `clientId`, `authority`, and `redirectUri` values in the `auth` object.

```
// MSAL instance to be passed to msal-angular
export function MSALInstanceFactory(): IPublicClientApplication {
    return new PublicClientApplication({
        auth: {
            clientId: 'Enter_the_Application_Id_Here',
            authority: 'Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Info_Here',
            redirectUri: 'Enter_the_Redirect_Uri_Here'
        },
        cache: {
            cacheLocation: BrowserCacheLocation.LocalStorage,
            storeAuthStateInCookie: isIE, // set to true for IE 11
        }
    })
}
```

Modify the values in the `auth` section as described here:

- `Enter_the_Application_Id_Here` is the **Application (client) ID** for the application you registered.

To find the value of **Application (client) ID**, go to the app registration's **Overview** page in the Azure portal.

- `Enter_the_Cloud_Instance_Id_Here` is the instance of the Azure cloud. For the main or global Azure cloud, enter <https://login.microsoftonline.com>. For **national** clouds (for example, China), see [National clouds](#).
- `Enter_the_Tenant_info_here` is set to one of the following:

- If your application supports *accounts in this organizational directory*, replace this value with the **Tenant ID** or **Tenant name**. For example, `contoso.microsoft.com`.

To find the value of the **Directory (tenant) ID**, go to the app registration's **Overview** page in the Azure portal.

- 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`. For this quickstart, use `common`.
 - To restrict support to *personal Microsoft accounts only*, replace this value with `consumers`.

To find the value of **Supported account types**, go to the app registration's **Overview** page in the Azure portal.

- `Enter_the_Redirect_Uri_Here` is `http://localhost:4200/`.

The `authority` value in your `app.module.ts` should be similar to the following if you're using the main (global) Azure cloud:

```
authority: "https://login.microsoftonline.com/common",
```

Scroll down in the same file and update the `graphMeEndpoint`.

- Replace the string `Enter_the_Graph_Endpoint_Herev1.0/me` with `https://graph.microsoft.com/v1.0/me`
- `Enter_the_Graph_Endpoint_Herev1.0/me` is the endpoint that API calls will be made against. For the main (global) Microsoft Graph API service, enter `https://graph.microsoft.com/` (include the trailing forward-slash). For more information, see the [documentation](#).

```
export function MSALInterceptorConfigFactory(): MsalInterceptorConfiguration {
  const protectedResourceMap = new Map<string, Array<string>>();
  protectedResourceMap.set('Enter_the_Graph_Endpoint_Herev1.0/me', ['user.read']);

  return {
    interactionType: InteractionType.Redirect,
    protectedResourceMap
  };
}
```

Step 4: Run the project

Run the project with a web server by using Node.js:

- To start the server, run the following commands from within the project directory:

```
npm install
npm start
```

- Browse to `http://localhost:4200/`.

- Select **Login** to start the sign-in process and then call the Microsoft Graph API.

The first time you sign in, you're prompted to provide your consent to allow the application to access your profile and sign you in. After you're signed in successfully, click the **Profile** button to display your user information on the page.

More information

How the sample works



msal.js

The MSAL.js library signs in users and requests the tokens that are used to access an API that's protected by the Microsoft identity platform.

If you have Node.js installed, you can download the latest version by using the Node.js Package Manager (npm):

```
npm install @azure/msal-browser @azure/msal-angular@2
```

Next steps

For a detailed step-by-step guide on building the auth code flow application using vanilla JavaScript, see the following tutorial:

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

In this quickstart, you download and run a code sample that demonstrates how a JavaScript single-page application (SPA) can sign in users and call Microsoft Graph using the authorization code flow with Proof Key for Code Exchange (PKCE). The code sample demonstrates how to 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 an Azure subscription for free](#)
- [Node.js](#)
- [Visual Studio Code](#) or another code editor

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. Go to the [Azure portal - App registrations](#).
2. Enter a name for your application.
3. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
4. Select **Register**.
5. Go to the quickstart pane and 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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application. Users of your app might see this name, and you can change it later.
6. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
7. Select **Register**. On the app **Overview** page, note the **Application (client) ID** value for later use.
8. Under **Manage**, select **Authentication**.
9. Under **Platform configurations**, select **Add a platform**. In the pane that opens select **Single-page application**.
10. Set the **Redirect URI** value to `http://localhost:3000/`.

11. Select **Configure**.

Step 2: Download the project

To run the project with a web server by using Node.js, [download the core project files](#).

Step 3: Configure your JavaScript app

In the `app` folder, open the `authConfig.js` file, and then update the `clientID`, `authority`, and `redirectUri` values in the `msalConfig` object.

```
// Config object to be passed to MSAL on creation
const msalConfig = {
  auth: {
    clientId: "Enter_the_Application_Id_Here",
    authority: "Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Info_Here",
    redirectUri: "Enter_the_Redirect_Uri_Here",
  },
  cache: {
    cacheLocation: "sessionStorage", // This configures where your cache will be stored
    storeAuthStateInCookie: false, // Set this to "true" if you are having issues on IE11 or Edge
  }
};
```

Modify the values in the `msalConfig` section:

- `Enter_the_Application_Id_Here` is the **Application (client) ID** for the application you registered.

To find the value of **Application (client) ID**, go to the app registration's **Overview** page in the Azure portal.

- `Enter_the_Cloud_Instance_Id_Here` is the Azure cloud instance. For the main or global Azure cloud, enter <https://login.microsoftonline.com>. For **national** clouds (for example, China), see [National clouds](#).
- `Enter_the_Tenant_info_here` is one of the following:

- If your application supports *accounts in this organizational directory*, replace this value with the **Tenant ID** or **Tenant name**. For example, `contoso.microsoft.com`.

To find the value of the **Directory (tenant) ID**, go to the app registration's **Overview** page in the Azure portal.

- 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`. For this quickstart, use `common`.
 - To restrict support to *personal Microsoft accounts only*, replace this value with `consumers`.

To find the value of **Supported account types**, go to the app registration's **Overview** page in the Azure portal.

- `Enter_the_Redirect_Uri_Here` is `http://localhost:3000/`.

The `authority` value in your `authConfig.js` should be similar to the following if you're using the main (global) Azure cloud:

```
authority: "https://login.microsoftonline.com/common",
```

Next, open the `graphConfig.js` file to update the `graphMeEndpoint` and `graphMailEndpoint` values in the `apiConfig` object.

```
// Add here the endpoints for MS Graph API services you would like to use.
const graphConfig = {
    graphMeEndpoint: "Enter_the_Graph_Endpoint_Herev1.0/me",
    graphMailEndpoint: "Enter_the_Graph_Endpoint_Herev1.0/me/messages"
};

// Add here scopes for access token to be used at MS Graph API endpoints.
const tokenRequest = {
    scopes: ["Mail.Read"]
};
```

`Enter_the_Graph_Endpoint_Here` is the endpoint that API calls are made against. For the main (global) Microsoft Graph API service, enter `https://graph.microsoft.com/` (include the trailing forward-slash). For more information about Microsoft Graph on national clouds, see [National cloud deployment](#).

If you're using the main (global) Microsoft Graph API service, the `graphMeEndpoint` and `graphMailEndpoint` values in the `graphConfig.js` file should be similar to the following:

```
graphMeEndpoint: "https://graph.microsoft.com/v1.0/me",
graphMailEndpoint: "https://graph.microsoft.com/v1.0/me/messages"
```

Step 4: Run the project

Run the project with a web server by using Node.js.

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

```
npm install
npm start
```

2. Go to `http://localhost:3000/`.

3. Select **Sign In** to start the sign-in process and then call the Microsoft Graph API.

The first time you sign in, you're prompted to provide your consent to allow the application to access your profile and sign you in. After you're signed in successfully, your user profile information is displayed on the page.

More information

How the sample works



MSAL.js

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

```
<script type="text/javascript" src="https://alcdn.msauth.net/browser/2.0.0-beta.0/js/msal-browser.js"
integrity=
"sha384-r7Qxfs6PYHyfoBR6zG62DGzptfLBxnREThAlcJyEfzJ4dq5rqExc1Xj3TPFE/9TH" crossorigin="anonymous"></script>
```

If you have Node.js installed, you can download the latest version by using the Node.js Package Manager (npm):

```
npm install @azure/msal-browser
```

Next steps

For a more detailed step-by-step guide on building the application used in this quickstart, see the following tutorial:

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

In this quickstart, you download and run a code sample that demonstrates how a JavaScript React single-page application (SPA) can sign in users and call Microsoft Graph using the authorization code flow. The code sample demonstrates how to 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 an Azure subscription for free](#)
- [Node.js](#)
- [Visual Studio Code](#) or another code editor

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. Go to the [Azure portal - App registrations](#) quickstart experience.
2. Enter a name for your application.
3. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
4. Select **Register**.
5. Go to the quickstart pane and 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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**. Under **Manage**, select **App registrations > New registration**.
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. Select **Register**. On the app **Overview** page, note the **Application (client) ID** value for later use.
7. Under **Manage**, select **Authentication**.
8. Under **Platform configurations**, select **Add a platform**. In the pane that opens select **Single-page application**.
9. Set the **Redirect URIs** value to `http://localhost:3000/`. This is the default port NodeJS will listen on your local machine. We'll return the authentication response to this URI after successfully authenticating the user.
10. Select **Configure** to apply the changes.
11. Under **Platform Configurations** expand **Single-page application**.
12. Confirm that under **Grant types** Your Redirect URI is eligible for the Authorization Code Flow with PKCE.

Step 2: Download the project

To run the project with a web server by using Node.js, [download the core project files](#).

Step 3: Configure your JavaScript app

In the `src` folder, open the `authConfig.js` file and update the `clientId`, `authority`, and `redirectUri` values in the `msalConfig` object.

```
/**
 * Configuration object to be passed to MSAL instance on creation.
 * For a full list of MSAL.js configuration parameters, visit:
 * https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-
browser/docs/configuration.md
 */
export const msalConfig = {
  auth: {
    clientId: "Enter_the_Application_Id_Here",
    authority: "Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Info_Here",
    redirectUri: "Enter_the_Redirect_Uri_Here"
  },
  cache: {
    cacheLocation: "sessionStorage", // This configures where your cache will be stored
    storeAuthStateInCookie: false, // Set this to "true" if you are having issues on IE11 or Edge
  },
};
```

Modify the values in the `msalConfig` section as described here:

- `Enter_the_Application_Id_Here` is the **Application (client) ID** for the application you registered.

To find the value of **Application (client) ID**, go to the app registration's **Overview** page in the Azure portal.

- `Enter_the_Cloud_Instance_Id_Here` is the instance of the Azure cloud. For the main or global Azure cloud, enter `https://login.microsoftonline.com`. For **national** clouds (for example, China), see [National clouds](#).
- `Enter_the_Tenant_info_here` is set to one of the following:
 - If your application supports *accounts in this organizational directory*, replace this value with the **Tenant ID** or **Tenant name**. For example, `contoso.microsoft.com`.

To find the value of the **Directory (tenant) ID**, go to the app registration's **Overview** page in the Azure portal.

- 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`. For this quickstart, use `common`.

- To restrict support to *personal Microsoft accounts only*, replace this value with `consumers`.

To find the value of **Supported account types**, go to the app registration's **Overview** page in the Azure portal.

- `Enter_the_Redirect_Uri_Here` is `http://localhost:3000/`.

The `authority` value in your *authConfig.js* should be similar to the following if you're using the main (global) Azure cloud:

```
authority: "https://login.microsoftonline.com/common",
```

Scroll down in the same file and update the `graphMeEndpoint`.

- Replace the string `Enter_the_Graph_Endpoint_Herev1.0/me` with `https://graph.microsoft.com/v1.0/me`
- `Enter_the_Graph_Endpoint_Herev1.0/me` is the endpoint that API calls will be made against. For the main (global) Microsoft Graph API service, enter `https://graph.microsoft.com/` (include the trailing forward-slash). For more information, see the [documentation](#).

```
// Add here the endpoints for MS Graph API services you would like to use.  
export const graphConfig = {  
    graphMeEndpoint: "Enter_the_Graph_Endpoint_Herev1.0/me"  
};
```

Step 4: Run the project

Run the project with a web server by using Node.js:

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

```
npm install  
npm start
```

2. Browse to `http://localhost:3000/`.

3. Select **Sign In** to start the sign-in process and then call the Microsoft Graph API.

The first time you sign in, you're prompted to provide your consent to allow the application to access your profile and sign you in. After you're signed in successfully, click on the **Request Profile Information** to display your profile information on the page.

More information

How the sample works



msal.js

The MSAL.js library signs in users and requests the tokens that are used to access an API that's protected by the Microsoft identity platform.

If you have Node.js installed, you can download the latest version by using the Node.js Package Manager (npm):

```
npm install @azure/msal-browser @azure/msal-react
```

Next steps

Next, try a step-by-step tutorial to learn how to build a React SPA from scratch that signs in users and calls the Microsoft Graph API to get user profile data:

[Tutorial: Sign in users and call Microsoft Graph](#)

Tutorial: Sign in users and call the Microsoft Graph API from an Angular single-page application (SPA) using auth code flow

4/12/2022 • 18 minutes to read • [Edit Online](#)

In this tutorial, you build an Angular single-page application (SPA) that signs in users and calls the Microsoft Graph API by using the authorization code flow with PKCE. The SPA you build uses the Microsoft Authentication Library (MSAL) for Angular v2.

In this tutorial:

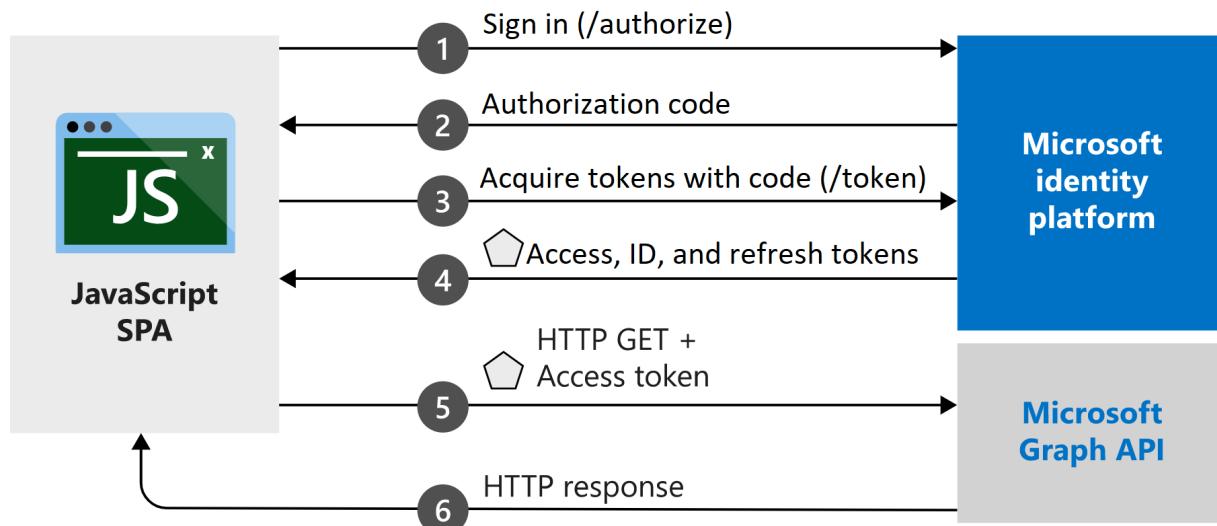
- Create an Angular project with `npm`
- Register the application in the Azure portal
- Add code to support user sign-in and sign-out
- Add code to call Microsoft Graph API
- Test the app

MSAL Angular v2 improves on MSAL Angular v1 by supporting the authorization code flow in the browser instead of the implicit grant flow. MSAL Angular v2 does **NOT** support the implicit flow.

Prerequisites

- [Node.js](#) for running a local web server.
- [Visual Studio Code](#) or other editor for modifying project files.

How the sample app works



The sample application created in this tutorial enables an Angular SPA to query the Microsoft Graph API or a web API that accepts tokens issued by the Microsoft identity platform. It uses the Microsoft Authentication Library (MSAL) for Angular v2, a wrapper of the MSAL.js v2 library. MSAL Angular enables Angular 9+ applications to authenticate enterprise users by using Azure Active Directory (Azure AD), and also users with Microsoft accounts and social identities like Facebook, Google, and LinkedIn. The library also enables applications to get access to Microsoft cloud services and Microsoft Graph.

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 MSAL.

Libraries

This tutorial uses the following libraries:

LIBRARY	DESCRIPTION
MSAL Angular	Microsoft Authentication Library for JavaScript Angular Wrapper
MSAL Browser	Microsoft Authentication Library for JavaScript v2 browser package

You can find the source code for all of the MSAL.js libraries in the [AzureAD/microsoft-authentication-library-for-js](#) repository on GitHub.

Create your project

Once you have [Node.js](#) installed, open up a terminal window and then run the following commands to generate a new Angular application:

```
npm install -g @angular/cli          # Install the Angular CLI
ng new msal-angular-tutorial --routing=true --style=css --strict=false    # Generate a new Angular app
cd msal-angular-tutorial            # Change to the app directory
npm install @angular/material @angular/cdk        # Install the Angular Material component library
(optional, for UI)
npm install @azure/msal-browser @azure/msal-angular # Install MSAL Browser and MSAL Angular in your
application
ng generate component home           # To add a home page
ng generate component profile       # To add a profile page
```

Register your application

Follow the [instructions to register a single-page application](#) in the Azure portal.

On the app **Overview** page of your registration, note the **Application (client) ID** value for later use.

Register your **Redirect URI** value as `http://localhost:4200/` and type as 'SPA'.

Configure the application

1. In the `src/app` folder, edit `app.module.ts` and add `MsalModule` and `MsalInterceptor` to `imports` as well as the `isIE` constant. Your code should look like this:

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { ProfileComponent } from './profile/profile.component';

import { MsalModule } from '@azure/msal-angular';
import { PublicClientApplication } from '@azure/msal-browser';

const isIE = window.navigator.userAgent.indexOf('MSIE ') > -1 ||
window.navigator.userAgent.indexOf('Trident/') > -1;

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ProfileComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    MsalModule.forRoot( new PublicClientApplication({
      auth: {
        clientId: 'Enter_the_Application_Id_here', // Application (client) ID from the app
registration
        authority: 'Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Info_Here', // The Azure cloud
instance and the app's sign-in audience (tenant ID, common, organizations, or consumers)
        redirectUri: 'Enter_the_Redirect_Uri_Here'// This is your redirect URI
      },
      cache: {
        cacheLocation: 'localStorage',
        storeAuthStateInCookie: isIE, // Set to true for Internet Explorer 11
      }
    }), null, null)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Replace these values:

VALUE NAME	ABOUT
Enter_the_Application_Id_Here	On the Overview page of your application registration, this is your Application (client) ID value.
Enter_the_Cloud_Instance_Id_Here	This is the instance of the Azure cloud. For the main or global Azure cloud, enter https://login.microsoftonline.com . For national clouds (for example, China), see National clouds .

Value Name	About
Enter_the_Tenant_Info_Here	Set to one of the following options: If your application supports <i>accounts in this organizational directory</i> , replace this value with the directory (tenant) ID or tenant name (for example, <code>contoso.microsoft.com</code>). If your application supports <i>accounts in any organizational directory</i> , replace this value with organizations . If your application supports <i>accounts in any organizational directory and personal Microsoft accounts</i> , replace this value with common . To restrict support to <i>personal Microsoft accounts only</i> , replace this value with consumers .
Enter_the_Redirect_Uri_Here	Replace with <code>http://localhost:4200</code> .

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

2. Add routes to the home and profile components in the `src/app/app-routing.module.ts`. Your code should look like the following:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { ProfileComponent } from './profile/profile.component';

const routes: Routes = [
  {
    path: 'profile',
    component: ProfileComponent,
  },
  {
    path: '',
    component: HomeComponent
  },
];

const isIframe = window !== window.parent && !window.opener;

@NgModule({
  imports: [RouterModule.forRoot(routes, {
    initialNavigation: !isIframe ? 'enabled' : 'disabled' // Don't perform initial navigation in
iframes
  })],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Replace base UI

1. Replace the placeholder code in `src/app/app.component.html` with the following:

```
<mat-toolbar color="primary">
  <a class="title" href="/">{{ title }}</a>

  <div class="toolbar-spacer"></div>

  <a mat-button [routerLink]=["'profile'"]>Profile</a>

  <button mat-raised-button *ngIf="!loginDisplay" (click)="login()">Login</button>

</mat-toolbar>
<div class="container">
  <!--This is to avoid reload during acquireTokenSilent() because of hidden iframe -->
  <router-outlet *ngIf="!isIframe"></router-outlet>
</div>
```

2. Add material modules to `src/app/app.module.ts`. Your `AppModule` should look like this:

```

import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { NgModule } from '@angular/core';

import { MatButtonModule } from '@angular/material/button';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatListModule } from '@angular/material/list';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { ProfileComponent } from './profile/profile.component';

import { MsalModule } from '@azure/msal-angular';
import { PublicClientApplication } from '@azure/msal-browser';

const isIE = window.navigator.userAgent.indexOf('MSIE ') > -1 || 
window.navigator.userAgent.indexOf('Trident/') > -1;

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ProfileComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    AppRoutingModule,
    MatButtonModule,
    MatToolbarModule,
    MatListModule,
    MsalModule.forRoot( new PublicClientApplication({
      auth: {
        clientId: 'Enter_the_Application_Id_here',
        authority: 'Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Info_Here',
        redirectUri: 'Enter_the_Redirect_Uri_Here'
      },
      cache: {
        cacheLocation: 'localStorage',
        storeAuthStateInCookie: isIE,
      }
    }), null, null)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

3. (OPTIONAL) Add CSS to *src/style.css*:

```

@import '~@angular/material/prebuilt-themes/deeppurple-amber.css';

html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }
.container { margin: 1%; }

```

4. (OPTIONAL) Add CSS to *src/app/app.component.css*:

```
.toolbar-spacer {  
    flex: 1 1 auto;  
}  
  
.title {  
    color: white;  
}
```

Sign in a user

Add the code from the following sections to invoke login using a pop-up window or a full-frame redirect:

Sign in using pop-ups

1. Change the code in `src/app/app.component.ts` to the following to sign in a user using a pop-up window:

```
import { MsalService } from '@azure/msal-angular';  
import { Component, OnInit } from '@angular/core';  
  
@Component({  
    selector: 'app-root',  
    templateUrl: './app.component.html',  
    styleUrls: ['./app.component.css']  
})  
export class AppComponent implements OnInit {  
    title = 'msal-angular-tutorial';  
    isIframe = false;  
    loginDisplay = false;  
  
    constructor(private authService: MsalService) { }  
  
    ngOnInit() {  
        this.isIframe = window !== window.parent && !window.opener;  
    }  
  
    login() {  
        this.authService.loginPopup()  
            .subscribe({  
                next: (result) => {  
                    console.log(result);  
                    this.setLoginDisplay();  
                },  
                error: (error) => console.log(error)  
            });  
    }  
  
    setLoginDisplay() {  
        this.loginDisplay = this.authService.instance.getAllAccounts().length > 0;  
    }  
}
```

NOTE

The rest of this tutorial uses the `loginRedirect` method with Microsoft Internet Explorer because of a [known issue](#) related to the handling of pop-up windows by Internet Explorer.

Sign in using redirects

1. Update `src/app/app.module.ts` to bootstrap the `MsalRedirectComponent`. This is a dedicated redirect component which will handle redirects. Your code should now look like this:

```

import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { NgModule } from '@angular/core';

import { MatButtonModule } from '@angular/material/button';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatListModule } from '@angular/material/list';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { ProfileComponent } from './profile/profile.component';

import { MsalModule, MsalRedirectComponent } from '@azure/msal-angular'; // Updated import
import { PublicClientApplication } from '@azure/msal-browser';

const isIE = window.navigator.userAgent.indexOf('MSIE ') > -1 ||
window.navigator.userAgent.indexOf('Trident/') > -1;

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ProfileComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    AppRoutingModule,
    MatButtonModule,
    MatToolbarModule,
    MatListModule,
    MsalModule.forRoot( new PublicClientApplication({
      auth: {
        clientId: 'Enter_the_Application_Id_here',
        authority: 'Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Info_Here',
        redirectUri: 'Enter_the_Redirect_Uri_Here'
      },
      cache: {
        cacheLocation: 'localStorage',
        storeAuthStateInCookie: isIE,
      }
    }), null, null)
  ],
  providers: [],
  bootstrap: [AppComponent, MsalRedirectComponent] // MsalRedirectComponent bootstrapped here
})
export class AppModule { }

```

2. Add the `<app-redirect>` selector to `src/index.html`. This selector is used by the `MsalRedirectComponent`. Your `src/index.html` should look like this:

```

<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>msal-angular-tutorial</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
  <app-redirect></app-redirect>
</body>
</html>

```

3. Replace the code in *src/app/app.component.ts* with the following to sign in a user using a full-frame redirect:

```

import { MsalService } from '@azure/msal-angular';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'msal-angular-tutorial';
  isIframe = false;
  loginDisplay = false;

  constructor(private authService: MsalService) { }

  ngOnInit() {
    this.isIframe = window !== window.parent && !window.opener;
  }

  login() {
    this.authService.loginRedirect();
  }

  setLoginDisplay() {
    this.loginDisplay = this.authService.instance.getAllAccounts().length > 0;
  }
}

```

4. Replace existing code in *src/app/home/home.component.ts* to subscribe to the `LOGIN_SUCCESS` event. This will allow you to access the result from the successful login with redirect. Your code should look like this:

```
import { Component, OnInit } from '@angular/core';
import { MsalBroadcastService, MsalService } from '@azure/msal-angular';
import { EventMessage, EventType, InteractionStatus } from '@azure/msal-browser';
import { filter } from 'rxjs/operators';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
  constructor(private authService: MsalService, private msalBroadcastService: MsalBroadcastService) {}

  ngOnInit(): void {
    this.msalBroadcastService.msalSubject$.
      pipe(
        filter((msg: EventMessage) => msg.eventType === EventType.LOGIN_SUCCESS),
      )
      .subscribe((result: EventMessage) => {
        console.log(result);
      });
  }
}
```

Conditional rendering

In order to render certain UI only for authenticated users, components have to subscribe to the `MsalBroadcastService` to see if users have been signed in and interaction has completed.

1. Add the `MsalBroadcastService` to `src/app/app.component.ts` and subscribe to the `inProgress$` observable to check if interaction is complete and an account is signed in before rendering UI. Your code should now look like this:

```

import { Component, OnInit, OnDestroy } from '@angular/core';
import { MsalService, MsalBroadcastService } from '@azure/msal-angular';
import { InteractionStatus } from '@azure/msal-browser';
import { Subject } from 'rxjs';
import { filter, takeUntil } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, OnDestroy {
  title = 'msal-angular-tutorial';
  isIframe = false;
  loginDisplay = false;
  private readonly _destroying$ = new Subject<void>();

  constructor(private broadcastService: MsalBroadcastService, private authService: MsalService) { }

  ngOnInit() {
    this.isIframe = window !== window.parent && !window.opener;

    this.broadcastService.inProgress$
      .pipe(
        filter((status: InteractionStatus) => status === InteractionStatus.None),
        takeUntil(this._destroying$)
      )
      .subscribe(() => {
        this.setLoginDisplay();
      })
  }

  login() {
    this.authService.loginRedirect();
  }

  setLoginDisplay() {
    this.loginDisplay = this.authService.instance.getAllAccounts().length > 0;
  }

  ngOnDestroy(): void {
    this._destroying$.next(undefined);
    this._destroying$.complete();
  }
}

```

2. Update the code in *src/app/home/home.component.ts* to also check for interaction to be completed before updating UI. Your code should now look like this:

```

import { Component, OnInit } from '@angular/core';
import { MsalBroadcastService, MsalService } from '@azure/msal-angular';
import { EventMessage, EventType, InteractionStatus } from '@azure/msal-browser';
import { filter } from 'rxjs/operators';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
  loginDisplay = false;

  constructor(private authService: MsalService, private msalBroadcastService: MsalBroadcastService) {}

  ngOnInit(): void {
    this.msalBroadcastService.msalSubject$
      .pipe(
        filter((msg: EventMessage) => msg.eventType === EventType.LOGIN_SUCCESS),
      )
      .subscribe((result: EventMessage) => {
        console.log(result);
      });

    this.msalBroadcastService.inProgress$
      .pipe(
        filter((status: InteractionStatus) => status === InteractionStatus.None)
      )
      .subscribe(() => {
        this.setLoginDisplay();
      })
  }

  setLoginDisplay() {
    this.loginDisplay = this.authService.instance.getAllAccounts().length > 0;
  }
}

```

3. Replace the code in *src/app/home/home.component.html* with the following conditional displays:

```

<div *ngIf="!loginDisplay">
  <p>Please sign-in to see your profile information.</p>
</div>

<div *ngIf="loginDisplay">
  <p>Login successful!</p>
  <p>Request your profile information by clicking Profile above.</p>
</div>

```

Guarding routes

Angular Guard

MSAL Angular provides `MsalGuard`, a class you can use to protect routes and require authentication before accessing the protected route. The steps below add the `MsalGuard` to the `Profile` route. Protecting the `Profile` route means that even if a user does not sign in using the `Login` button, if they try to access the `Profile` route or click the `Profile` button, the `MsalGuard` will prompt the user to authenticate via pop-up or redirect before showing the `Profile` page.

`MsalGuard` is a convenience class you can use to improve the user experience, but it should not be relied upon for security. Attackers can potentially get around client-side guards, and you should ensure that the server does not

return any data the user should not access.

1. Add the `MsalGuard` class as a provider in your application in `src/app/app.module.ts`, and add the configurations for the `MsalGuard`. Scopes needed for acquiring tokens later can be provided in the `authRequest`, and the type of interaction for the Guard can be set to `Redirect` or `Popup`. Your code should look like the following:

```
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { NgModule } from '@angular/core';

import { MatButtonModule } from '@angular/material/button';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatListModule } from '@angular/material/list';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { ProfileComponent } from './profile/profile.component';

import { MsalModule, MsalRedirectComponent, MsalGuard } from '@azure/msal-angular'; // MsalGuard added to imports
import { PublicClientApplication, InteractionType } from '@azure/msal-browser'; // InteractionType added to imports

const isIE = window.navigator.userAgent.indexOf('MSIE ') > -1 ||
window.navigator.userAgent.indexOf('Trident/') > -1;

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ProfileComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    AppRoutingModule,
    MatButtonModule,
    MatToolbarModule,
    MatListModule,
    MsalModule.forRoot( new PublicClientApplication({
      auth: {
        clientId: 'Enter_the_Application_Id_here',
        authority: 'Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Info_Here',
        redirectUri: 'Enter_the_Redirect_Uri_Here'
      },
      cache: {
        cacheLocation: 'localStorage',
        storeAuthStateInCookie: isIE,
      }
    )),
    {
      interactionType: InteractionType.Redirect, // MSAL Guard Configuration
      authRequest: {
        scopes: ['user.read']
      }
    }, null)
  ],
  providers: [
    MsalGuard // MsalGuard added as provider here
  ],
  bootstrap: [AppComponent, MsalRedirectComponent]
})
export class AppModule { }
```

2. Set the `MsalGuard` on the routes you wish to protect in `src/app/app-routing.module.ts`:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { ProfileComponent } from './profile/profile.component';
import { MsalGuard } from '@azure/msal-angular';

const routes: Routes = [
{
  path: 'profile',
  component: ProfileComponent,
  canActivate: [MsalGuard]
},
{
  path: '',
  component: HomeComponent
},
];

const isIframe = window !== window.parent && !window.opener;

@NgModule({
  imports: [RouterModule.forRoot(routes, {
    initialNavigation: !isIframe ? 'enabled' : 'disabled' // Don't perform initial navigation in
iframes
  })],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

3. Adjust the login calls in `src/app/app.component.ts` to take the `authRequest` set in the guard configurations into account. Your code should now look like the following:

```

import { Component, OnInit, OnDestroy, Inject } from '@angular/core';
import { MsalService, MsalBroadcastService, MSAL_GUARD_CONFIG, MsalGuardConfiguration } from
'@azure/msal-angular';
import { InteractionStatus, RedirectRequest } from '@azure/msal-browser';
import { Subject } from 'rxjs';
import { filter, takeUntil } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, OnDestroy {
  title = 'msal-angular-tutorial';
  isIframe = false;
  loginDisplay = false;
  private readonly _destroying$ = new Subject<void>();

  constructor(@Inject(MSAL_GUARD_CONFIG) private msalGuardConfig: MsalGuardConfiguration, private
broadcastService: MsalBroadcastService, private authService: MsalService) { }

  ngOnInit() {
    this.isIframe = window !== window.parent && !window.opener;

    this.broadcastService.inProgress$
    .pipe(
      filter((status: InteractionStatus) => status === InteractionStatus.None),
      takeUntil(this._destroying$)
    )
    .subscribe(() => {
      this.setLoginDisplay();
    })
  }

  login() {
    if (this.msalGuardConfig.authRequest){
      this.authService.loginRedirect({...this.msalGuardConfig.authRequest} as RedirectRequest);
    } else {
      this.authService.loginRedirect();
    }
  }

  setLoginDisplay() {
    this.loginDisplay = this.authService.instance.getAllAccounts().length > 0;
  }

  ngOnDestroy(): void {
    this._destroying$.next(undefined);
    this._destroying$.complete();
  }
}

```

Acquire a token

Angular Interceptor

MSAL Angular provides an `Interceptor` class that automatically acquires tokens for outgoing requests that use the Angular `http` client to known protected resources.

1. Add the `Interceptor` class as a provider to your application in `src/app/app.module.ts`, with its configurations. Your code should now look like this:

```

import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

```

```

import { NgModule } from '@angular/core';
import { HTTP_INTERCEPTORS, HttpClientModule } from "@angular/common/http"; // Import

import { MatButtonModule } from '@angular/material/button';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatListModule } from '@angular/material/list';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { ProfileComponent } from './profile/profile.component';

import { MsalModule, MsalRedirectComponent, MsalGuard, MsalInterceptor } from '@azure/msal-angular';
// Import MsalInterceptor
import { InteractionType, PublicClientApplication } from '@azure/msal-browser';

const isIE = window.navigator.userAgent.indexOf('MSIE ') > -1 ||
window.navigator.userAgent.indexOf('Trident/') > -1;

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ProfileComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    AppRoutingModule,
    MatButtonModule,
    MatToolbarModule,
    MatListModule,
    HttpClientModule,
    MsalModule.forRoot( new PublicClientApplication({
      auth: {
        clientId: 'Enter_the_Application_Id_Here',
        authority: 'Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Info_Here',
        redirectUri: 'Enter_the_Redirect_Uri_Here',
      },
      cache: {
        cacheLocation: 'localStorage',
        storeAuthStateInCookie: isIE,
      }
    )),
    {
      interactionType: InteractionType.Redirect,
      authRequest: {
        scopes: ['user.read']
      }
    },
    {
      interactionType: InteractionType.Redirect, // MSAL Interceptor Configuration
      protectedResourceMap: new Map([
        ['Enter_the_Graph_Endpoint_Here/v1.0/me', ['user.read']]
      ])
    }
  ],
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: MsalInterceptor,
      multi: true
    },
    MsalGuard
  ],
  bootstrap: [AppComponent, MsalRedirectComponent]
})
export class AppModule { }

```

The protected resources are provided as a `protectedResourceMap`. The URLs you provide in the `protectedResourceMap` collection are case-sensitive. For each resource, add scopes being requested to be returned in the access token.

For example:

- `["user.read"]` for Microsoft Graph
- `[<Application ID URL>/scope"]` for custom web APIs (that is, `api://<Application ID>/access_as_user`)

Modify the values in the `protectedResourceMap` as described here:

VALUE NAME	ABOUT
<code>Enter_the_Graph_Endpoint_Here</code>	The instance of the Microsoft Graph API the application should communicate with. For the global Microsoft Graph API endpoint, replace both instances of this string with https://graph.microsoft.com . For endpoints in national cloud deployments, see National cloud deployments in the Microsoft Graph documentation.

2. Replace the code in `src/app/profile/profile.component.ts` to retrieve a user's profile with an HTTP request:

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

const GRAPH_ENDPOINT = 'Enter_the_Graph_Endpoint_Here/v1.0/me';

type ProfileType = {
  givenName?: string,
  surname?: string,
  userPrincipalName?: string,
  id?: string
};

@Component({
  selector: 'app-profile',
  templateUrl: './profile.component.html',
  styleUrls: ['./profile.component.css']
})
export class ProfileComponent implements OnInit {
  profile!: ProfileType;

  constructor(
    private http: HttpClient
  ) { }

  ngOnInit() {
    this.getProfile();
  }

  getProfile() {
    this.http.get(GRAPH_ENDPOINT)
      .subscribe(profile => {
        this.profile = profile;
      });
  }
}
```

3. Replace the UI in `src/app/profile/profile.component.html` to display profile information:

```
<div>
  <p><strong>First Name: </strong> {{profile?.givenName}}</p>
  <p><strong>Last Name: </strong> {{profile?.surname}}</p>
  <p><strong>Email: </strong> {{profile?.userPrincipalName}}</p>
  <p><strong>Id: </strong> {{profile?.id}}</p>
</div>
```

Sign out

Update the code in `src/app/app.component.html` to conditionally display a `Logout` button:

```
<mat-toolbar color="primary">
  <a class="title" href="/">{{ title }}</a>

  <div class="toolbar-spacer"></div>

  <a mat-button [routerLink]=["/profile"]>Profile</a>

  <button mat-raised-button *ngIf="!loginDisplay" (click)="login()">Login</button>
  <button mat-raised-button *ngIf="loginDisplay" (click)="logout()">Logout</button>

</mat-toolbar>
<div class="container">
  <!--This is to avoid reload during acquireTokenSilent() because of hidden iframe -->
  <router-outlet *ngIf="!isIframe"></router-outlet>
</div>
```

Sign out using redirects

Update the code in `src/app/app.component.ts` to sign out a user using redirects:

```

import { Component, OnInit, OnDestroy, Inject } from '@angular/core';
import { MsalService, MsalBroadcastService, MSAL_GUARD_CONFIG, MsalGuardConfiguration } from '@azure/msal-angular';
import { InteractionStatus, RedirectRequest } from '@azure/msal-browser';
import { Subject } from 'rxjs';
import { filter, takeUntil } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, OnDestroy {
  title = 'msal-angular-tutorial';
  isIframe = false;
  loginDisplay = false;
  private readonly _destroying$ = new Subject<void>();

  constructor(@Inject(MSAL_GUARD_CONFIG) private msalGuardConfig: MsalGuardConfiguration, private broadcastService: MsalBroadcastService, private authService: MsalService) { }

  ngOnInit() {
    this.isIframe = window !== window.parent && !window.opener;

    this.broadcastService.inProgress$
      .pipe(
        filter((status: InteractionStatus) => status === InteractionStatus.None),
        takeUntil(this._destroying$)
      )
      .subscribe(() => {
        this.setLoginDisplay();
      })
  }

  login() {
    if (this.msalGuardConfig.authRequest){
      this.authService.loginRedirect({...this.msalGuardConfig.authRequest} as RedirectRequest);
    } else {
      this.authService.loginRedirect();
    }
  }

  logout() { // Add log out function here
    this.authService.logoutRedirect({
      postLogoutRedirectUri: 'http://localhost:4200'
    });
  }

  setLoginDisplay() {
    this.loginDisplay = this.authService.instance.getAllAccounts().length > 0;
  }

  ngOnDestroy(): void {
    this._destroying$.next(undefined);
    this._destroying$.complete();
  }
}

```

Sign out using pop-ups

Update the code in `src/app/app.component.ts` to sign out a user using pop-ups:

```

import { Component, OnInit, OnDestroy, Inject } from '@angular/core';
import { MsalService, MsalBroadcastService, MSAL_GUARD_CONFIG, MsalGuardConfiguration } from '@azure/msal-angular';
import { InteractionStatus, PopupRequest } from '@azure/msal-browser';

```

```

import { Subject } from 'rxjs';
import { filter, takeUntil } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, OnDestroy {
  title = 'msal-angular-tutorial';
  isIframe = false;
  loginDisplay = false;
  private readonly _destroying$ = new Subject<void>();

  constructor(@Inject(MSAL_GUARD_CONFIG) private msalGuardConfig: MsalGuardConfiguration, private broadcastService: MsalBroadcastService, private authService: MsalService) { }

  ngOnInit() {
    this.isIframe = window !== window.parent && !window.opener;

    this.broadcastService.inProgress$
    .pipe(
      filter((status: InteractionStatus) => status === InteractionStatus.None),
      takeUntil(this._destroying$)
    )
    .subscribe(() => {
      this.setLoginDisplay();
    })
  }

  login() {
    if (this.msalGuardConfig.authRequest){
      this.authService.loginPopup({...this.msalGuardConfig.authRequest} as PopupRequest)
        .subscribe({
          next: (result) => {
            console.log(result);
            this.setLoginDisplay();
          },
          error: (error) => console.log(error)
        });
    } else {
      this.authService.loginPopup()
        .subscribe({
          next: (result) => {
            console.log(result);
            this.setLoginDisplay();
          },
          error: (error) => console.log(error)
        });
    }
  }

  logout() { // Add log out function here
    this.authService.logoutPopup({
      mainWindowRedirectUri: "/"
    });
  }

  setLoginDisplay() {
    this.loginDisplay = this.authService.instance.getAllAccounts().length > 0;
  }

  ngOnDestroy(): void {
    this._destroying$.next(undefined);
    this._destroying$.complete();
  }
}

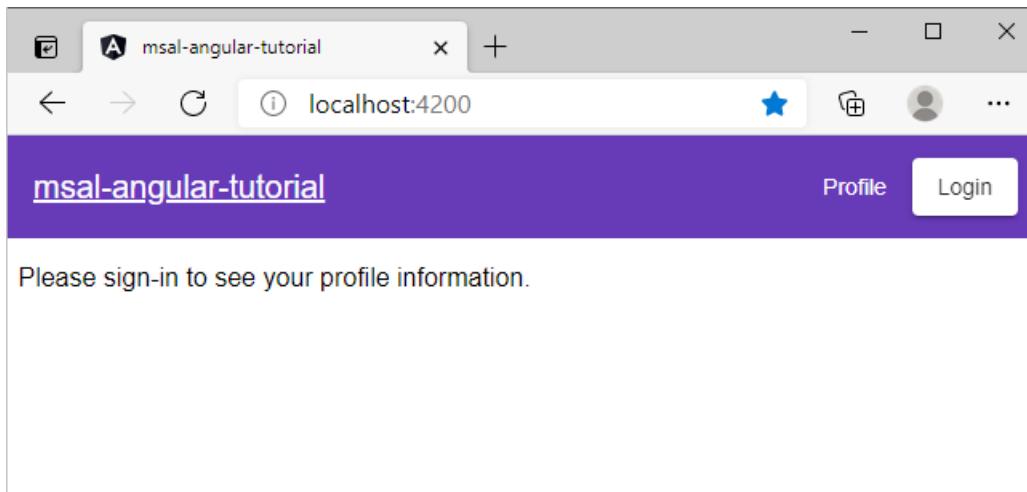
```

Test your code

1. Start the web server to listen to the port by running the following commands at a command-line prompt from the application folder:

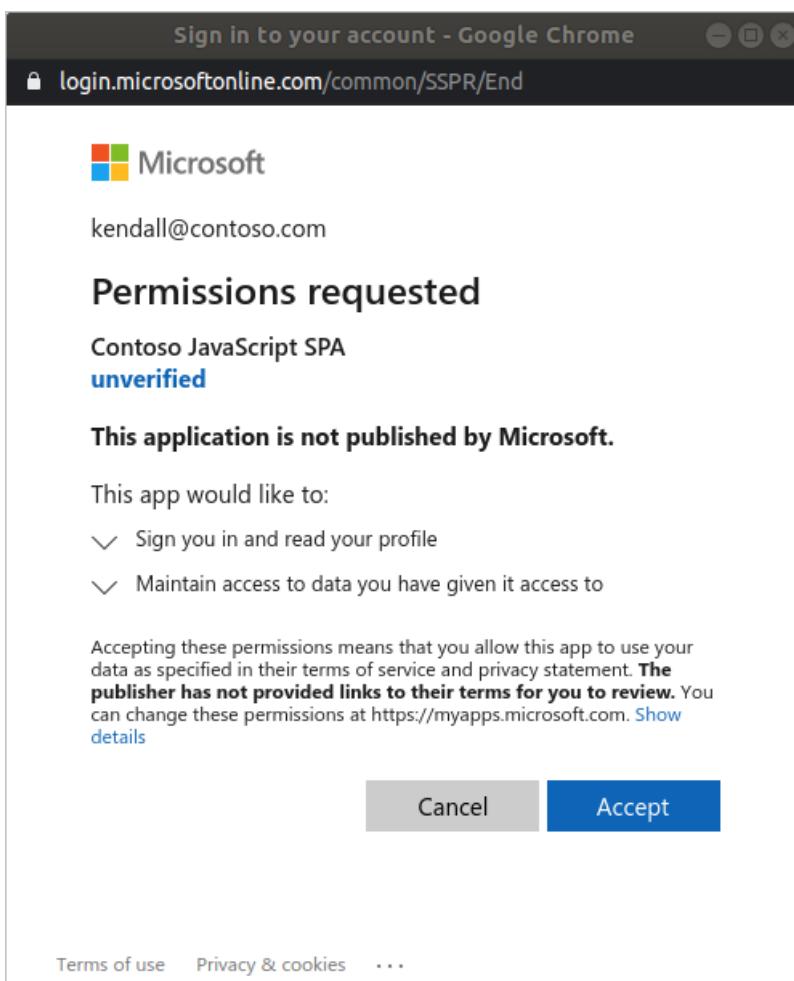
```
npm install  
npm start
```

2. In your browser, enter `http://localhost:4200` or `http://localhost:{port}`, where *port* is the port that your web server is listening on. You should see a page that looks like the one below.

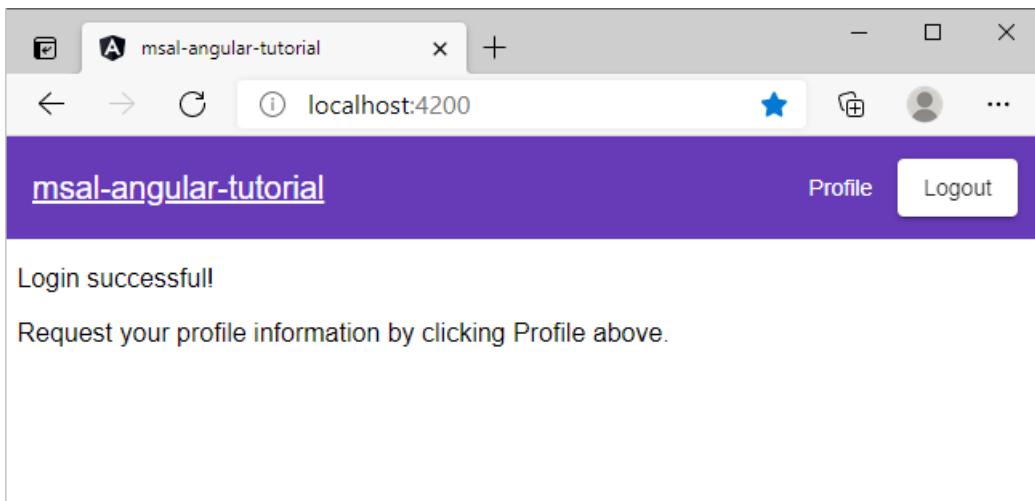


Provide consent for application access

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

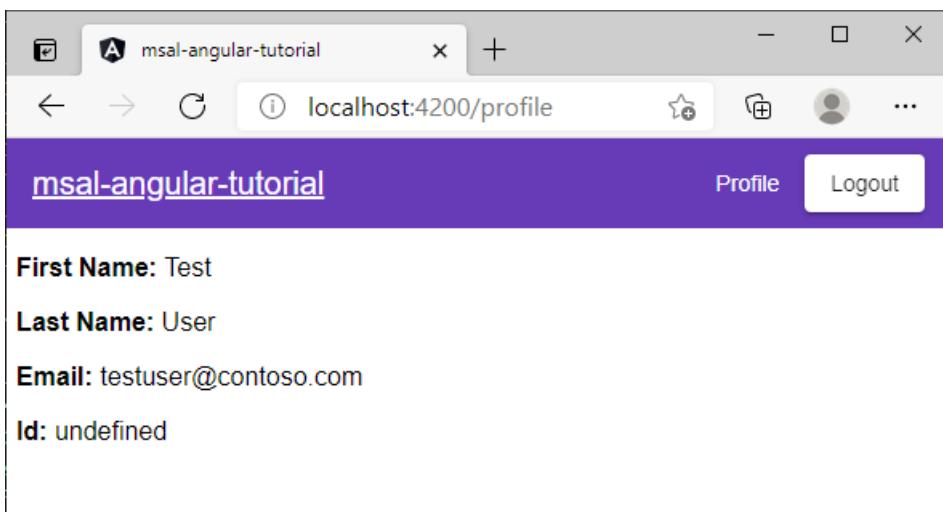


If you consent to the requested permissions, the web application shows a successful login page:



Call the Graph API

After you sign in, select **Profile** to view the user profile information returned in the response from the call to the Microsoft Graph API:



Add scopes and delegated permissions

The Microsoft Graph API requires the *User.Read* scope to read a user's profile. The *User.Read* scope is added automatically to every app registration you create in the Azure 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 *Mail.Read* scope in order to list the user's email.

As you add scopes, your users might be prompted to provide additional consent for the added scopes.

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 about your support options, see [Help and support for developers](#).

Next steps

Delve deeper into single-page application (SPA) development on the Microsoft identity platform in our the multi-part article series.

[Scenario: Single-page application](#)

Tutorial: Sign in users and call a protected API from a Blazor WebAssembly app

4/12/2022 • 6 minutes to read • [Edit Online](#)

In this tutorial, you build a Blazor WebAssembly app that signs in users and gets data from Microsoft Graph by using the Microsoft identity platform and registering your app in Azure Active Directory (Azure AD).

In this tutorial:

- Create a new Blazor WebAssembly app configured to use Azure Active Directory (Azure AD) for [authentication and authorization](#) using the Microsoft identity platform
- Retrieve data from a protected web API, in this case [Microsoft Graph](#)

This tutorial uses .NET Core 3.1. The .NET docs contain instructions on [how to secure a Blazor WebAssembly app](#) using ASP.NET Core 5.0.

We also have a [tutorial for Blazor Server](#).

Prerequisites

- [.NET Core 3.1 SDK](#)
- An Azure AD tenant where you can register an app. If you don't have access to an Azure AD tenant, you can get one by registering with the [Microsoft 365 Developer Program](#) or by creating an [Azure free account](#).

Register the app in the Azure portal

Every app that uses Azure Active Directory (Azure AD) for authentication must be registered with Azure AD. Follow the instructions in [Register an application](#) with these specifications:

- For **Supported account types**, select **Accounts in this organizational directory only**.
- Set the **Redirect URI** drop down to **Single-page application (SPA)** and enter `https://localhost:5001/authentication/login-callback`. The default port for an app running on Kestrel is 5001. If the app is available on a different port, specify that port number instead of `5001`.

Once registered, under **Manage**, select **Authentication > Implicit grant and hybrid flows**. Select **Access tokens** and **ID tokens**, and then select **Save**.

Note: if you're using .NET 6 or later then you don't need to use Implicit grant. The latest template uses MSAL Browser 2.0 and supports Auth Code Flow with PKCE

Create the app using the .NET Core CLI

To create the app you need the latest Blazor templates. You can install them for the .NET Core CLI with the following command:

```
dotnet new -i Microsoft.Identity.Web.ProjectTemplates::1.9.1
```

Then run the following command to create the application. Replace the placeholders in the command with the proper information from your app's overview page and execute the command in a command shell. The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the

app's name.

```
dotnet new blazorwasm2 --auth SingleOrg --calls-graph -o {APP NAME} --client-id "{CLIENT ID}" --tenant-id "{TENANT ID}"
```

PLACEHOLDER	AZURE PORTAL NAME	EXAMPLE
{APP NAME}	—	BlazorWASMSample
{CLIENT ID}	Application (client) ID	41451fa7-0000-0000-0000-69eff5a761fd
{TENANT ID}	Directory (tenant) ID	e86c78e2-0000-0000-0000-918e0565a45e

Test the app

You can now build and run the app. When you run this template app, you must specify the framework to run using `--framework`. This tutorial uses the .NET Standard 2.1, but the template supports other frameworks as well.

```
dotnet run --framework netstandard2.1
```

In your browser, navigate to <https://localhost:5001>, and log in using an Azure AD user account to see the app running and logging users in with the Microsoft identity platform.

The components of this template that enable logins with Azure AD using the Microsoft identity platform are explained in the [ASP.NET doc on this topic](#).

Retrieving data from a protected API (Microsoft Graph)

[Microsoft Graph](#) contains APIs that provide access to Microsoft 365 data for your users, and it supports the tokens issued by the Microsoft identity platform, which makes it a good protected API to use as an example. In this section, you add code to call Microsoft Graph and display the user's emails on the application's "Fetch data" page.

This section is written using a common approach to calling a protected API using a named client. The same method can be used for other protected APIs you want to call. However, if you do plan to call Microsoft Graph from your application you can use the Graph SDK to reduce boilerplate. The .NET docs contain instructions on [how to use the Graph SDK](#).

Before you start, log out of your app since you'll be making changes to the required permissions, and your current token won't work. If you haven't already, run your app again and select **Log out** before updating the code below.

Now you will update your app's registration and code to pull a user's emails and display the messages within the app.

First, add the `Mail.Read` API permission to the app's registration so that Azure AD is aware that the app will request to access its users' email.

1. In the Azure portal, select your app in **App registrations**.
2. Under **Manage**, select **API permissions**.
3. Select **Add a permission > Microsoft Graph**.

4. Select **Delegated Permissions**, then search for and select the **Mail.Read** permission.

5. Select **Add permissions**.

Next, add the following to your project's `.csproj` file in the `netstandard2.1 ItemGroup`. This will allow you to create the custom `HttpClient` in the next step.

```
<PackageReference Include="Microsoft.Extensions.Http" Version="3.1.7" />
```

Then modify the code as specified in the next few steps. These changes will add [access tokens](#) to the outgoing requests sent to the Microsoft Graph API. This pattern is discussed in more detail in [ASP.NET Core Blazor WebAssembly additional security scenarios](#).

First, create a new file named `GraphAPIAuthorizationMessageHandler.cs` with the following code. This handler will be used to add an access token for the `User.Read` and `Mail.Read` scopes to outgoing requests to the Microsoft Graph API.

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class GraphAPIAuthorizationMessageHandler : AuthorizationMessageHandler
{
    public GraphAPIAuthorizationMessageHandler(IAccessTokenProvider provider,
        NavigationManager navigationManager)
        : base(provider, navigationManager)
    {
        ConfigureHandler(
            authorizedUrls: new[] { "https://graph.microsoft.com" },
            scopes: new[] { "https://graph.microsoft.com/User.Read", "https://graph.microsoft.com/Mail.Read" });
    }
}
```

Then, replace the contents of the `Main` method in `Program.cs` with the following code. This code makes use of the new `GraphAPIAuthorizationMessageHandler` and adds `User.Read` and `Mail.Read` as default scopes the app will request when the user first signs in.

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("app");

builder.Services.AddScoped<GraphAPIAuthorizationMessageHandler>();

builder.Services.AddHttpClient("GraphAPI",
    client => client.BaseAddress = new Uri("https://graph.microsoft.com"))
    .AddHttpMessageHandler<GraphAPIAuthorizationMessageHandler>();

builder.Services.AddMsalAuthentication(options =>
{
    builder.Configuration.Bind("AzureAd", options.ProviderOptions.Authentication);
    options.ProviderOptions.DefaultAccessTokenScopes.Add("User.Read");
    options.ProviderOptions.DefaultAccessTokenScopes.Add("Mail.Read");
});

await builder.Build().RunAsync();
```

Finally, replace the contents of the `FetchData.razor` page with the following code. This code fetches user email data from the Microsoft Graph API and displays them as a list. In `onInitializedAsync`, the new `HttpClient` that uses the proper access token is created and used to make the request to the Microsoft Graph API.

```
@page "/fetchdata"
```

```

@using System.ComponentModel.DataAnnotations
@using System.Text.Json.Serialization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@inject IAccessTokenProvider TokenProvider
@inject IHttpClientFactory ClientFactory
@inject IHttpClientFactory HttpClientFactory

<p>This component demonstrates fetching data from a service.</p>

@if (messages == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <h1>Hello @userDisplayName !!!!</h1>
    <table class="table">
        <thead>
            <tr>
                <th>Subject</th>
                <th>Sender</th>
                <th>Received Time</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var mail in messages)
            {
                <tr>
                    <td>@mail.Subject</td>
                    <td>@mail.Sender</td>
                    <td>@mail.ReceivedTime</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {

    private string userDisplayName;
    private List<MailMessage> messages = new List<MailMessage>();

    private HttpClient _httpClient;

    protected override async Task OnInitializedAsync()
    {
        _httpClient = HttpClientFactory.CreateClient("GraphAPI");
        try {
            var dataRequest = await _httpClient.GetAsync("https://graph.microsoft.com/beta/me");

            if (dataRequest.IsSuccessStatusCode)
            {
                var userData = System.Text.Json.JsonDocument.Parse(await
dataRequest.Content.ReadAsStreamAsync());
                userDisplayName = userData.RootElement.GetProperty("displayName").GetString();
            }

            var mailRequest = await _httpClient.GetAsync("https://graph.microsoft.com/beta/me/messages?
$select=subject,receivedDateTime,sender&$top=10");

            if (mailRequest.IsSuccessStatusCode)
            {
                var mailData = System.Text.Json.JsonDocument.Parse(await
mailRequest.Content.ReadAsStreamAsync());
                var messagesArray = mailData.RootElement.GetProperty("value").EnumerateArray();

                foreach (var m in messagesArray)
                {

```

```

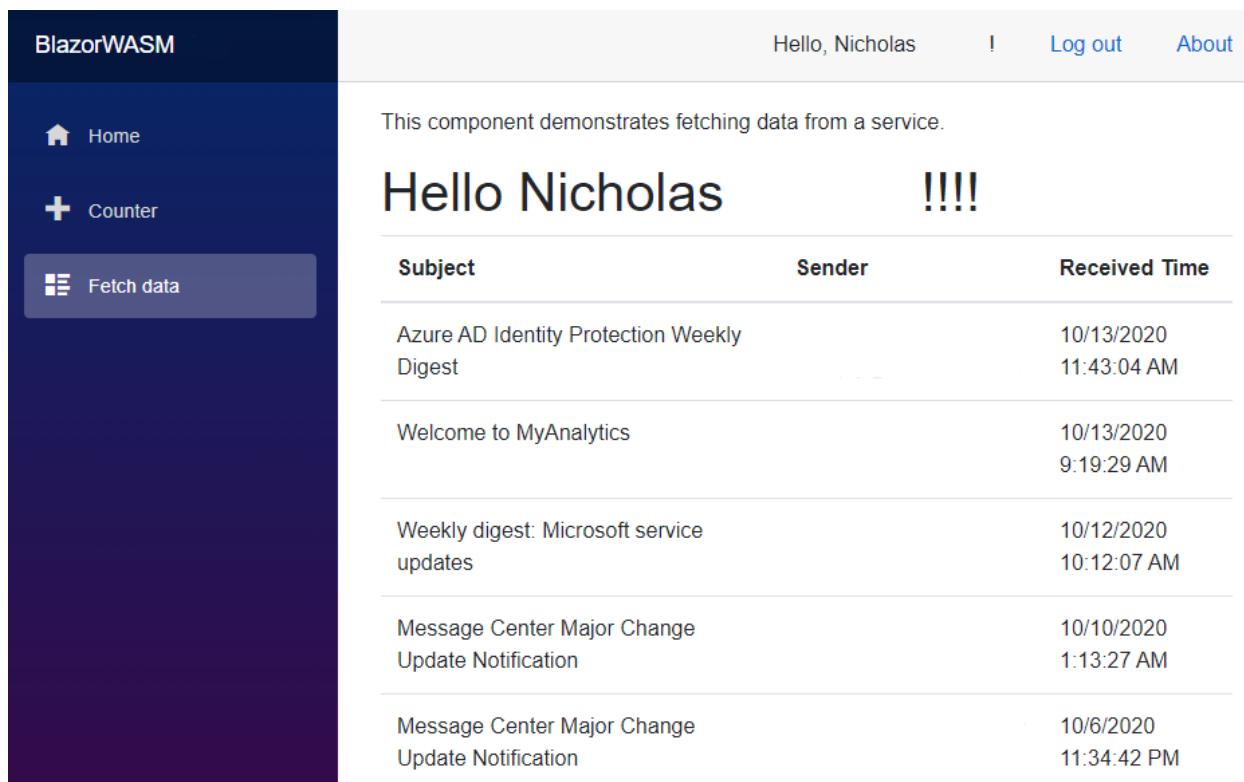
        var message = new MailMessage();
        message.Subject = m.GetProperty("subject").GetString();
        message.Sender =
m.GetProperty("sender").GetProperty("emailAddress").GetProperty("address").GetString();
            message.ReceivedTime = m.GetProperty("receivedDateTime").GetDateTime();
            messages.Add(message);
        }
    }
}
catch (AccessTokenNotAvailableException ex)
{
    // Tokens are not valid - redirect the user to log in again
    ex.Redirect();
}
}

public class MailMessage
{
    public string Subject;
    public string Sender;
    public DateTime ReceivedTime;
}
}

```

Now launch the app again. You'll notice that you're prompted to give the app access to read your mail. This is expected when an app requests the `Mail.Read` scope.

After granting consent, navigate to the "Fetch data" page to read some email.



The screenshot shows the BlazorWASM application interface. On the left is a dark sidebar with three items: "Home", "Counter", and "Fetch data", which is highlighted with a blue background. The main content area has a header "Hello, Nicholas" and a "Log out" link. Below the header is a text "This component demonstrates fetching data from a service." followed by a large "Hello Nicholas" and four exclamation marks. A table lists five received emails:

Subject	Sender	Received Time
Azure AD Identity Protection Weekly Digest		10/13/2020 11:43:04 AM
Welcome to MyAnalytics		10/13/2020 9:19:29 AM
Weekly digest: Microsoft service updates		10/12/2020 10:12:07 AM
Message Center Major Change Update Notification		10/10/2020 1:13:27 AM
Message Center Major Change Update Notification		10/6/2020 11:34:42 PM

Next steps

[Microsoft identity platform best practices and recommendations](#)

Tutorial: Sign in users and call the Microsoft Graph API from a JavaScript single-page app (SPA) using auth code flow

4/12/2022 • 15 minutes to read • [Edit Online](#)

In this tutorial, you build a JavaScript single-page application (SPA) that signs in users and calls Microsoft Graph by using the authorization code flow with PKCE. The SPA you build uses the Microsoft Authentication Library (MSAL) for JavaScript v2.0.

In this tutorial:

- Perform the OAuth 2.0 authorization code flow with PKCE
- Sign in personal Microsoft accounts as well as work and school accounts
- Acquire an access token
- Call Microsoft Graph or your own API that requires access tokens obtained from the Microsoft identity platform

MSAL.js 2.0 improves on MSAL.js 1.0 by supporting the authorization code flow in the browser instead of the implicit grant flow. MSAL.js 2.0 does **NOT** support the implicit flow.

Prerequisites

- [Node.js](#) for running a local webserver
- [Visual Studio Code](#) or another code editor

How the tutorial app works



The application you create in this tutorial enables a JavaScript SPA to query the Microsoft Graph API by acquiring security tokens from the Microsoft identity platform. In this scenario, after a user signs in, an access token is requested and added to HTTP requests in the authorization header. Token acquisition and renewal are handled by the Microsoft Authentication Library for JavaScript (MSAL.js).

This tutorial uses the following library:

Get the completed code sample

Prefer to download this tutorial's completed sample project instead? Clone the [ms-identity-javascript-v2](https://github.com/Azure-Samples/ms-identity-javascript-v2) repository.

```
git clone https://github.com/Azure-Samples/ms-identity-javascript-v2
```

To run the downloaded project on your local development environment, start by creating a localhost server for your application as described in step 1 of [create your project](#). Once done, you can configure the code sample by skipping to the [configuration step](#).

To continue with the tutorial and build the application yourself, move on to the next section, [Create your project](#).

Create your project

Once you have [Node.js](#) installed, create a folder to host your application, for example *msal-spa-tutorial*.

Next, implement a small [Express](#) web server to serve your *index.htm*/file.

1. First, change to your project directory in your terminal and then run the following [npm](#) commands:

```
npm init -y
npm install @azure/msal-browser
npm install express
npm install morgan
npm install yargs
```

2. Next, create file named *server.js* and add the following code:

```

const express = require('express');
const morgan = require('morgan');
const path = require('path');
const argv = require('yargs')
  .usage('Usage: $0 -p [PORT]')
  .alias('p', 'port')
  .describe('port', '(Optional) Port Number - default is 3000')
  .strict()
  .argv;

const DEFAULT_PORT = 3000;

// Initialize express.
const app = express();

// Initialize variables.
let port = DEFAULT_PORT; // -p {PORT} || 3000;
if (argv.p) {
  port = argv.p;
}

// Configure morgan module to log all requests.
app.use(morgan('dev'));

// Set the front-end folder to serve public assets.
app.use("/lib", express.static(path.join(__dirname, "../../lib/msal-browser/lib")));

// Setup app folders
app.use(express.static('app'));

// Set up a route for index.html.
app.get('*', function (req, res) {
  res.sendFile(path.join(__dirname + '/index.html'));
});

// Start the server.
app.listen(port);
console.log(`Listening on port ${port}...`);

```

You now have a small webserver to serve your SPA. After completing the rest of the tutorial, the file and folder structure of your project should look similar to the following:

```

msal-spa-tutorial/
├── app
│   ├── authConfig.js
│   ├── authPopup.js
│   ├── authRedirect.js
│   ├── graphConfig.js
│   ├── graph.js
│   ├── index.html
│   └── ui.js
└── server.js

```

Create the SPA UI

1. Create an *app* folder in your project directory, and in it create an *index.html* file for your JavaScript SPA.

This file implements a UI built with the **Bootstrap 4 Framework** and imports script files for configuration, authentication, and API calls.

In the *index.html* file, add the following code:

```
<!DOCTYPE html>
```

```

<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0, shrink-to-fit=no">
    <title>Tutorial | MSAL.js JavaScript SPA</title>

    <!-- IE support: add promises polyfill before msal.js -->
    <script type="text/javascript"
      src="//cdn.jsdelivr.net/npm/bluebird@3.7.2/js/browser/bluebird.min.js"></script>
    <script type="text/javascript" src="https://alcdn.msauth.net/browser/2.0.0-beta.4/js/msal-
    browser.js" integrity="sha384-7sxY2tN3GMVE5jXH2RL9Adb06s46vUh9lUid4yNCHJMUzDoj+0N4ve6rLOmR88yN"
    crossorigin="anonymous"></script>

    <!-- adding Bootstrap 4 for UI components -->
    <link rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css" integrity="sha384-
    Vkoo8x4CGs03+Hhxv8T/Q5PaXtkKtu6ug5T0eNV6gBiFeWPGFN9MuhOf23Q9Ifjh" crossorigin="anonymous">
    <link rel="SHORTCUT ICON" href="https://c.s-microsoft.com/favicon.ico?v2" type="image/x-icon">
  </head>
  <body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-primary">
      <a class="navbar-brand" href="/">Microsoft identity platform</a>
      <div class="btn-group ml-auto dropleft">
        <button type="button" id="SignIn" class="btn btn-secondary" onclick="signIn()">
          Sign In
        </button>
      </div>
    </nav>
    <br>
    <h5 class="card-header text-center">JavaScript SPA calling Microsoft Graph API with MSAL.js</h5>
    <br>
    <div class="row" style="margin:auto" >
      <div id="card-div" class="col-md-3" style="display:none">
        <div class="card text-center">
          <div class="card-body">
            <h5 class="card-title" id="WelcomeMessage">Please sign-in to see your profile and read your
            mails</h5>
            <div id="profile-div"></div>
            <br>
            <br>
            <button class="btn btn-primary" id="seeProfile" onclick="seeProfile()">See Profile</button>
            <br>
            <br>
            <button class="btn btn-primary" id="readMail" onclick="readMail()">Read Mail</button>
          </div>
        </div>
        <br>
        <br>
        <div class="col-md-4">
          <div class="list-group" id="list-tab" role="tablist">
            </div>
          </div>
          <div class="col-md-5">
            <div class="tab-content" id="nav-tabContent">
              </div>
            </div>
          </div>
          <br>
          <br>
        </div>
        <!-- importing bootstrap.js and supporting js libraries -->
        <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js" integrity="sha384-
        J6qa4849b1E2+poT4WnyKh5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n" crossorigin="anonymous"></script>
        <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
        integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
        crossorigin="anonymous"></script>
        <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"
        integrity="sha384-wfSDF2E50Y2D1uUdj003uMBJnjuUD4IH7YwaYd1iqfktj0Uod8GCEx130g8ifwB6"></script>
      </div>
    </div>
  </body>

```

```

crossorigin="anonymous"></script>

<!-- importing app scripts (load order is important) -->
<script type="text/javascript" src=".authConfig.js"></script>
<script type="text/javascript" src=".graphConfig.js"></script>
<script type="text/javascript" src=".ui.js"></script>

<!-- <script type="text/javascript" src=".authRedirect.js"></script> -->
<!-- uncomment the above line and comment the line below if you would like to use the redirect
flow -->
<script type="text/javascript" src=".authPopup.js"></script>
<script type="text/javascript" src=".graph.js"></script>
</body>
</html>

```

2. Next, also in the *app* folder, create a file named *ui.js* and add the following code. This file will access and update DOM elements.

```

// Select DOM elements to work with
const welcomeDiv = document.getElementById("WelcomeMessage");
const signInButton = document.getElementById("SignIn");
const cardDiv = document.getElementById("card-div");
const mailButton = document.getElementById("readMail");
const profileButton = document.getElementById("seeProfile");
const profileDiv = document.getElementById("profile-div");

function showWelcomeMessage(account) {
    // Reconfiguring DOM elements
    cardDiv.style.display = 'initial';
    welcomeDiv.innerHTML = `Welcome ${account.username}`;
    signInButton.setAttribute("onclick", "signOut();");
    signInButton.setAttribute('class', "btn btn-success")
    signInButton.innerHTML = "Sign Out";
}

function updateUI(data, endpoint) {
    console.log('Graph API responded at: ' + new Date().toString());

    if (endpoint === graphConfig.graphMeEndpoint) {
        const title = document.createElement('p');
        title.innerHTML = "<strong>Title: </strong>" + data.jobTitle;
        const email = document.createElement('p');
        email.innerHTML = "<strong>Mail: </strong>" + data.mail;
        const phone = document.createElement('p');
        phone.innerHTML = "<strong>Phone: </strong>" + data.businessPhones[0];
        const address = document.createElement('p');
        address.innerHTML = "<strong>Location: </strong>" + data.officeLocation;
        profileDiv.appendChild(title);
        profileDiv.appendChild(email);
        profileDiv.appendChild(phone);
        profileDiv.appendChild(address);
    } else if (endpoint === graphConfig.graphMailEndpoint) {
        if (data.value.length < 1) {
            alert("Your mailbox is empty!")
        } else {
            const tabList = document.getElementById("list-tab");
            tabList.innerHTML = ''; // clear tabList at each readMail call
            const tabContent = document.getElementById("nav-tabContent");

            data.value.map((d, i) => {
                // Keeping it simple
                if (i < 10) {
                    const listItem = document.createElement("a");
                    listItem.setAttribute("class", "list-group-item list-group-item-action");
                    listItem.setAttribute("id", "list" + i + "list")
                    listItem.setAttribute("data-toggle", "list")

```

```

        listItem.setAttribute("data-toggle", "list");
        listItem.setAttribute("href", "#list" + i);
        listItem.setAttribute("role", "tab");
        listItem.setAttribute("aria-controls", i);
        listItem.innerHTML = d.subject;
        tabList.appendChild(listItem);

        const contentItem = document.createElement("div");
        contentItem.setAttribute("class", "tab-pane fade");
        contentItem.setAttribute("id", "list" + i);
        contentItem.setAttribute("role", "tabpanel");
        contentItem.setAttribute("aria-labelledby", "list" + i + "list");
        contentItem.innerHTML = "<strong> from: " + d.from.emailAddress.address + "</strong><br><br>" + d.bodyPreview + "...";
        tabContent.appendChild(contentItem);
    }
});

}

}
}

```

Register your application

Follow the steps in [Single-page application: App registration](#) to create an app registration for your SPA.

In the [Redirect URI: MSAL.js 2.0 with auth code flow](#) step, enter `http://localhost:3000`, the default location where this tutorial's application runs.

If you'd like to use a different port, enter `http://localhost:<port>`, where `<port>` is your preferred TCP port number. If you specify a port number other than `3000`, also update `server.js` with your preferred port number.

Configure your JavaScript SPA

Create a file named `authConfig.js` in the `app` folder to contain your configuration parameters for authentication, and then add the following code:

```

const msalConfig = {
    auth: {
        clientId: "Enter_the_Application_Id_Here",
        authority: "Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Info_Here",
        redirectUri: "Enter_the_Redirect_Uri_Here",
    },
    cache: {
        cacheLocation: "sessionStorage", // This configures where your cache will be stored
        storeAuthStateInCookie: false, // Set this to "true" if you are having issues on IE11 or Edge
    }
};

// Add scopes here for ID token to be used at Microsoft identity platform endpoints.
const loginRequest = {
    scopes: ["openid", "profile", "User.Read"]
};

// Add scopes here for access token to be used at Microsoft Graph API endpoints.
const tokenRequest = {
    scopes: ["User.Read", "Mail.Read"]
};

```

Modify the values in the `msalConfig` section as described here:

- `Enter_the_Application_Id_Here`: The **Application (client) ID** of the application you registered.
- `Enter_the_Cloud_Instance_Id_Here`: The Azure cloud instance in which your application is registered.
 - For the main (or *global*) Azure cloud, enter `https://login.microsoftonline.com`.

- For **national** clouds (for example, China), you can find appropriate values in [National clouds](#).
- `Enter_the_Tenant_info_here` should be one of the following:
 - 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`.
- `Enter_the_Redirect_Uri_Here` is `http://localhost:3000`.

The `authority` value in your `authConfig.js` should be similar to the following if you're using the global Azure cloud:

```
authority: "https://login.microsoftonline.com/common",
```

Still in the `app` folder, create a file named `graphConfig.js`. Add the following code to provide your application the configuration parameters for calling the Microsoft Graph API:

```
// Add the endpoints here for Microsoft Graph API services you'd like to use.
const graphConfig = {
  graphMeEndpoint: "Enter_the_Graph_Endpoint_Here/v1.0/me",
  graphMailEndpoint: "Enter_the_Graph_Endpoint_Here/v1.0/me/messages"
};
```

Modify the values in the `graphConfig` section as described here:

- `Enter_the_Graph_Endpoint_Here` is the instance of the Microsoft Graph API the application should communicate with.
 - For the **global** Microsoft Graph API endpoint, replace both instances of this string with `https://graph.microsoft.com`.
 - For endpoints in **national** cloud deployments, see [National cloud deployments](#) in the Microsoft Graph documentation.

The `graphMeEndpoint` and `graphMailEndpoint` values in your `graphConfig.js` should be similar to the following if you're using the global endpoint:

```
graphMeEndpoint: "https://graph.microsoft.com/v1.0/me",
graphMailEndpoint: "https://graph.microsoft.com/v1.0/me/messages"
```

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

Pop-up

In the `app` folder, create a file named `authPopup.js` and add the following authentication and token acquisition code for the login pop-up:

```
// Create the main myMSALObj instance
// configuration parameters are located at authConfig.js
const myMSALObj = new msal.PublicClientApplication(msalConfig);

let username = "";

function loadPage() {
```

```

    /**
     * See here for more info on account retrieval:
     * https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-common/docs/Accounts.md
     */
    const currentAccounts = myMSALObj.getAllAccounts();
    if (currentAccounts === null) {
        return;
    } else if (currentAccounts.length > 1) {
        // Add choose account code here
        console.warn("Multiple accounts detected.");
    } else if (currentAccounts.length === 1) {
        username = currentAccounts[0].username;
        showWelcomeMessage(currentAccounts[0]);
    }
}

function handleResponse(resp) {
    if (resp !== null) {
        username = resp.account.username;
        showWelcomeMessage(resp.account);
    } else {
        loadPage();
    }
}

function signIn() {
    myMSALObj.loginPopup(loginRequest).then(handleResponse).catch(error => {
        console.error(error);
    });
}

function signOut() {
    const logoutRequest = {
        account: myMSALObj.getAccountByUsername(username)
    };

    myMSALObj.logout(logoutRequest);
}

function getTokenPopup(request) {
    /**
     * See here for more info on account retrieval:
     * https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-common/docs/Accounts.md
     */
    request.account = myMSALObj.getAccountByUsername(username);
    return myMSALObj.acquireTokenSilent(request).catch(error => {
        console.warn("silent token acquisition fails. acquiring token using redirect");
        if (error instanceof msal.InteractionRequiredAuthError) {
            // fallback to interaction when silent call fails
            return myMSALObj.acquireTokenPopup(request).then(tokenResponse => {
                console.log(tokenResponse);

                return tokenResponse;
            }).catch(error => {
                console.error(error);
            });
        } else {
            console.warn(error);
        }
    });
}

function seeProfile() {
    getTokenPopup(loginRequest).then(response => {
        callMSGraph(graphConfig.graphMeEndpoint, response.accessToken, updateUI);
        profileButton.classList.add('d-none');
        mailButton.classList.remove('d-none');
    });
}

```

```

        mailButton.classList.remove('a-none'),
    }).catch(error => {
    console.error(error);
});
}

function readMail() {
    getTokenPopup(tokenRequest).then(response => {
        callMSGraph(graphConfig.graphMailEndpoint, response.accessToken, updateUI);
    }).catch(error => {
    console.error(error);
});
}

loadPage();

```

Redirect

Create a file named *authRedirect.js* in the *app* folder and add the following authentication and token acquisition code for login redirect:

```

// Create the main myMSALObj instance
// configuration parameters are located at authConfig.js
const myMSALObj = new msal.PublicClientApplication(msalConfig);

let accessToken;
let username = "";

// Redirect: once login is successful and redirects with tokens, call Graph API
myMSALObj.handleRedirectPromise().then(handleResponse).catch(err => {
    console.error(err);
});

function handleResponse(resp) {
    if (resp !== null) {
        username = resp.account.username;
        showWelcomeMessage(resp.account);
    } else {
        /**
         * See here for more info on account retrieval:
         * https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-common/docs/Accounts.md
         */
        const currentAccounts = myMSALObj.getAllAccounts();
        if (currentAccounts === null) {
            return;
        } else if (currentAccounts.length > 1) {
            // Add choose account code here
            console.warn("Multiple accounts detected.");
        } else if (currentAccounts.length === 1) {
            username = currentAccounts[0].username;
            showWelcomeMessage(currentAccounts[0]);
        }
    }
}

function signIn() {
    myMSALObj.loginRedirect(loginRequest);
}

function signOut() {
    const logoutRequest = {
        account: myMSALObj.getAccountByUsername(username)
    };
    myMSALObj.logout(logoutRequest);
}

```

```

function getTokenRedirect(request) {
    /**
     * See here for more info on account retrieval:
     * https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-common/docs/Accounts.md
     */
    request.account = myMSALObj.getAccountByUsername(username);
    return myMSALObj.acquireTokenSilent(request).catch(error => {
        console.warn("silent token acquisition fails. acquiring token using redirect");
        if (error instanceof msal.InteractionRequiredAuthError) {
            // fallback to interaction when silent call fails
            return myMSALObj.acquireTokenRedirect(request);
        } else {
            console.warn(error);
        }
    });
}

function seeProfile() {
    getTokenRedirect(loginRequest).then(response => {
        callMSGraph(graphConfig.graphMeEndpoint, response.accessToken, updateUI);
        profileButton.classList.add('d-none');
        mailButton.classList.remove('d-none');
    }).catch(error => {
        console.error(error);
    });
}

function readMail() {
    getTokenRedirect(tokenRequest).then(response => {
        callMSGraph(graphConfig.graphMailEndpoint, response.accessToken, updateUI);
    }).catch(error => {
        console.error(error);
    });
}

```

How the code works

When a user selects the **Sign In** button for the first time, the `signIn` method calls `loginPopup` to sign in the user. The `loginPopup` 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, *msal.js* initiates the [authorization code flow](#).

At this point, a PKCE-protected authorization code is sent to the CORS-protected token endpoint and is exchanged for tokens. An ID token, access token, and refresh token are received by your application and processed by *msal.js*, and the information contained in the tokens is cached.

The ID token contains basic information about the user, like their display name. If you plan to use any data provided by the ID token, your back-end server *must* validate it to guarantee the token was issued to a valid user for your application.

The access token has a limited lifetime and expires after 24 hours. The refresh token can be used to silently acquire new access tokens.

The SPA you've created in this tutorial 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, see the [active-directory-javascript-singlepageapp-dotnet-webapi-v2](#) sample application on GitHub. The sample uses an ASP.NET web API for token validation.

Get a user token interactively

After their initial sign-in, your app shouldn't ask users to reauthenticate every time they need to access a protected resource (that is, to request a token). To prevent such reauthentication requests, call `acquireTokenSilent`. There are some situations, however, where you might need to force users to interact with the Microsoft identity platform. For example:

- Users need to re-enter 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). In that window, users need to interact by confirming their credentials, giving consent to the required resource, or completing the two-factor authentication.

Get 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 to trigger 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. Visually indicate 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 technique 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 tutorial uses the `loginPopup` and `acquireTokenPopup` methods by default. If you're using Internet Explorer, we recommend that you use the `loginRedirect` and `acquireTokenRedirect` methods due to a [known issue](#) with Internet Explorer and pop-up windows. For an example of achieving the same result by using redirect methods, see [authRedirect.js](#) on GitHub.

Call the Microsoft Graph API

Create file named `graph.js` in the `app` folder and add the following code for making REST calls to the Microsoft Graph API:

```
// Helper function to call Microsoft Graph API endpoint
// using authorization bearer token scheme
function callMSGraph(endpoint, token, callback) {
    const headers = new Headers();
    const bearer = `Bearer ${token}`;

    headers.append("Authorization", bearer);

    const options = {
        method: "GET",
        headers: headers
    };

    console.log('request made to Graph API at: ' + new Date().toString());

    fetch(endpoint, options)
        .then(response => response.json())
        .then(response => callback(response, endpoint))
        .catch(error => console.log(error));
}
```

In the sample application created in this tutorial, 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*. In the sample application created in this tutorial, the protected resource is the Microsoft Graph API *me* endpoint which displays the signed-in user's profile information.

Test your application

You've completed creation of the application and are now ready to launch the Node.js web server and test the app's functionality.

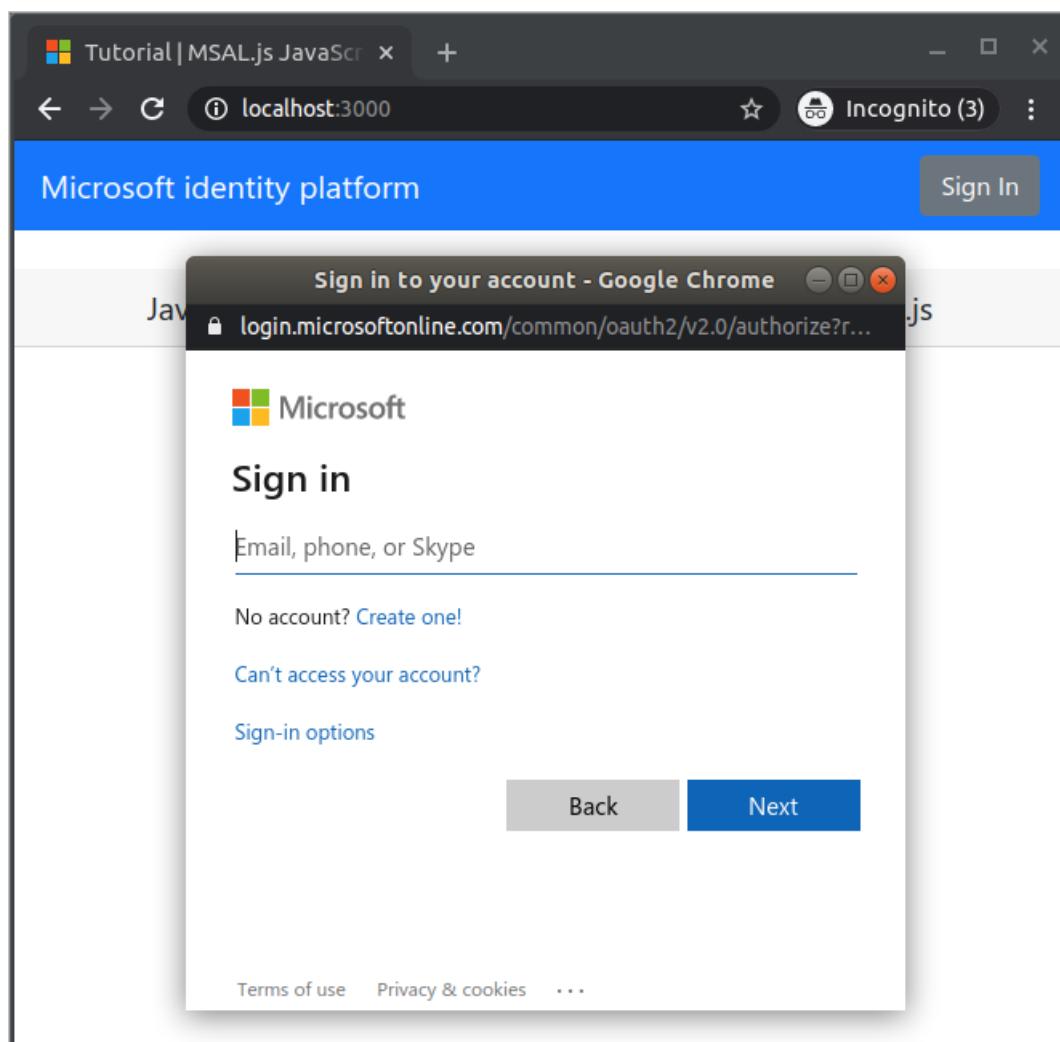
1. Start the Node.js web server by running the following command from within the root of your project folder:

```
npm start
```

2. In your browser, navigate to `http://localhost:3000` or `http://localhost:<port>`, where `<port>` is the port that your web server is listening on. You should see the contents of your *index.html* file and the **Sign In** button.

Sign in to the application

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



Provide consent for application access

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

Sign in to your account - Google Chrome

login.microsoftonline.com/common/SSPR/End

 Microsoft

kendall@contoso.com

Permissions requested

Contoso JavaScript SPA
unverified

This application is not published by Microsoft.

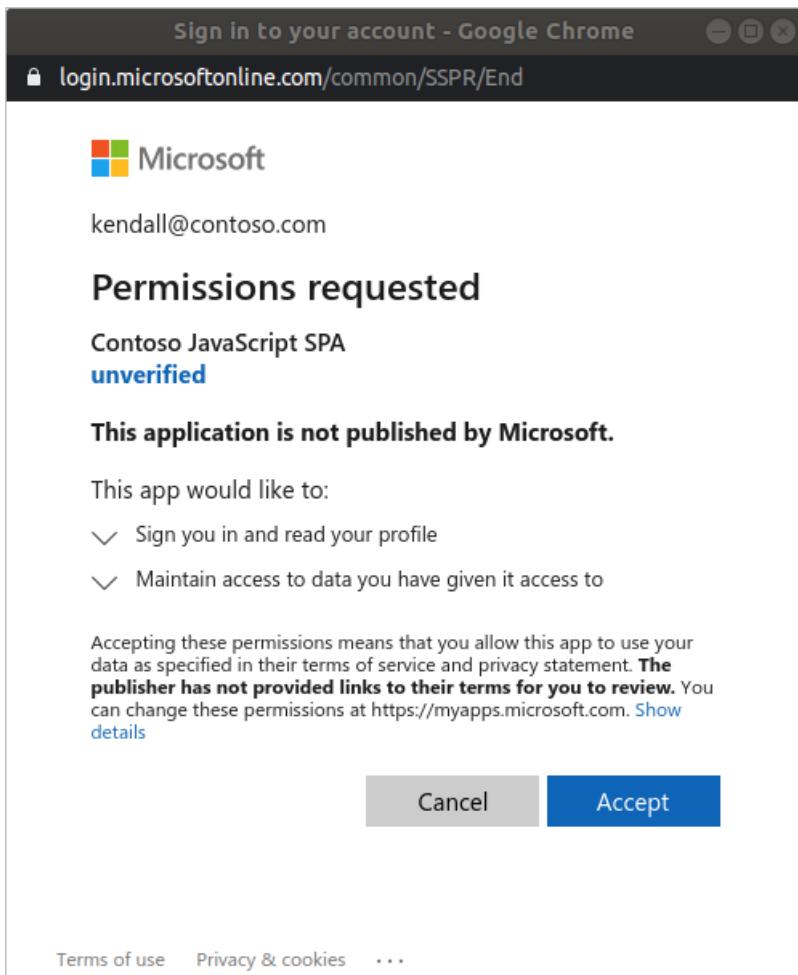
This app would like to:

- ✓ Sign you in and read your profile
- ✓ Maintain access to data you have given it access to

Accepting these permissions means that you allow this app to use your data as specified in their terms of service and privacy statement. **The publisher has not provided links to their terms for you to review.** You can change these permissions at <https://myapps.microsoft.com>. [Show details](#)

[Cancel](#) [Accept](#)

[Terms of use](#) [Privacy & cookies](#) ...



If you consent to the requested permissions, the web application displays your user name, signifying a successful login:

Tutorial | MSAL.js JavaScript x +

localhost:3000 Incognito (2) :

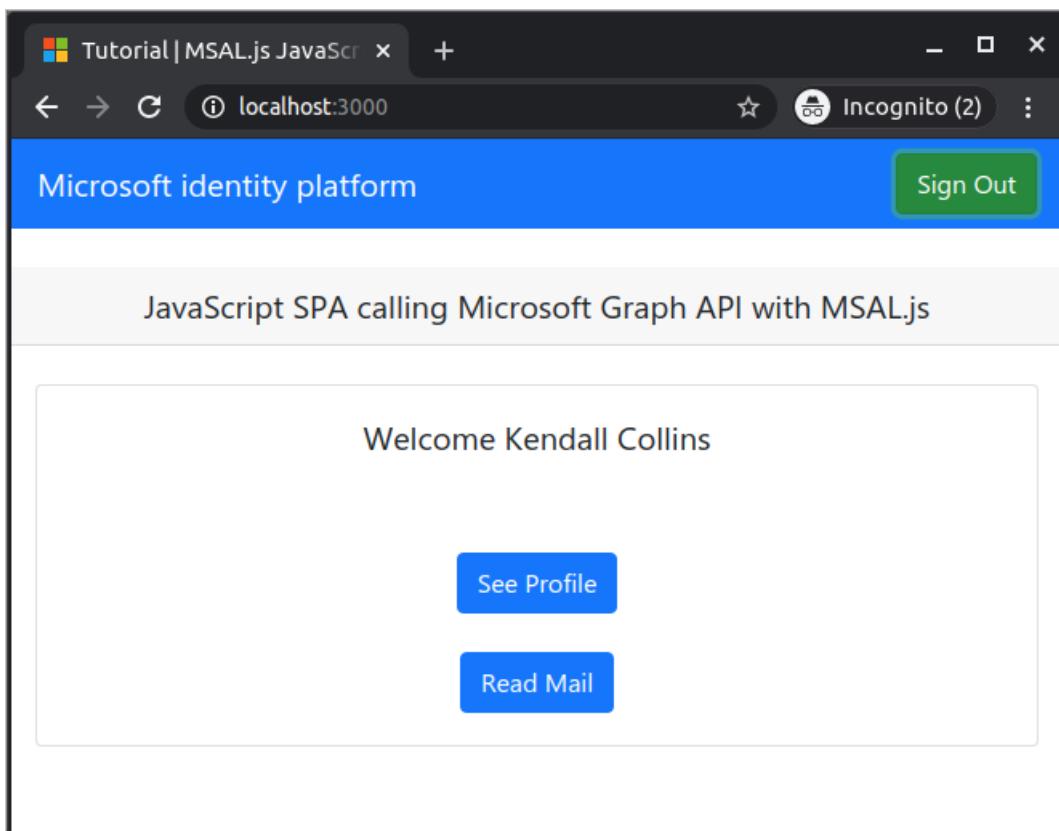
Microsoft identity platform Sign Out

JavaScript SPA calling Microsoft Graph API with MSAL.js

Welcome Kendall Collins

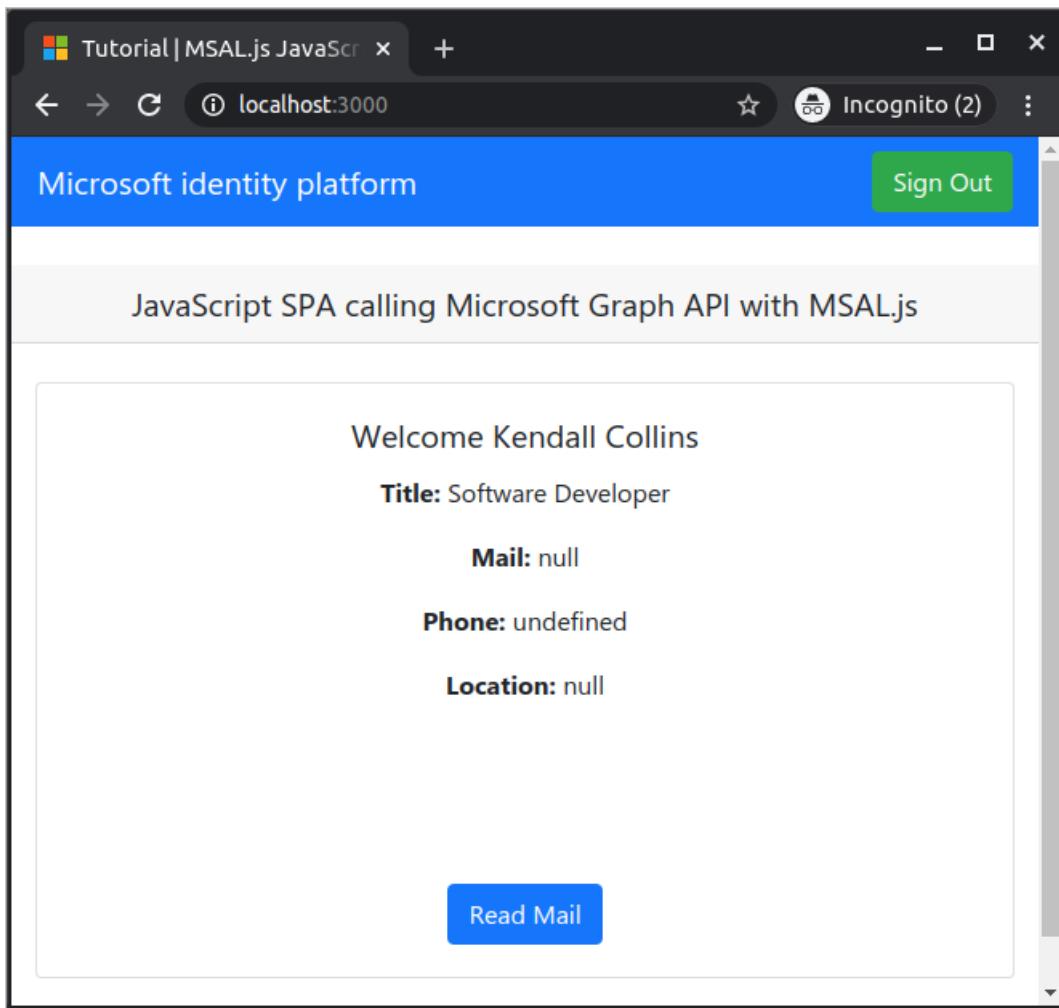
See Profile

Read Mail



Call the Graph API

After you sign in, select See Profile to view the user profile information returned in the response from the call to the Microsoft Graph API:



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 in the Azure 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 `Mail.Read` scope in order to list the user's email.

As you add scopes, your users might be prompted to provide additional consent for the added scopes.

If a back-end API doesn't require a scope, which isn't 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 about your support options, see [Help and support for developers](#).

Next steps

If you'd like to dive deeper into JavaScript single-page application development on the Microsoft identity platform, see our multi-part scenario series:

[Scenario: Single-page application](#)

Tutorial: Sign in users and call the Microsoft Graph API from a React single-page app (SPA) using auth code flow

4/12/2022 • 14 minutes to read • [Edit Online](#)

In this tutorial, you build a React single-page application (SPA) that signs in users and calls Microsoft Graph by using the authorization code flow with PKCE. The SPA you build uses the Microsoft Authentication Library (MSAL) for React.

In this tutorial:

- Create a React project with `npm`
- Register the application in the Azure portal
- Add code to support user sign-in and sign-out
- Add code to call Microsoft Graph API
- Test the app

MSAL React supports the authorization code flow in the browser instead of the implicit grant flow. MSAL React does **NOT** support the implicit flow.

Prerequisites

- [Node.js](#) for running a local webserver
- [Visual Studio Code](#) or another code editor

How the tutorial app works



The application you create in this tutorial enables a React SPA to query the Microsoft Graph API by acquiring security tokens from the Microsoft identity platform. It uses the Microsoft Authentication Library (MSAL) for React, a wrapper of the MSAL.js v2 library. MSAL React enables React 16+ applications to authenticate enterprise users by using Azure Active Directory (Azure AD), and also users with Microsoft accounts and social identities like Facebook, Google, and LinkedIn. The library also enables applications to get access to Microsoft cloud services and Microsoft Graph.

In this scenario, after a user signs in, an access token is requested and added to HTTP requests in the authorization header. Token acquisition and renewal are handled by the Microsoft Authentication Library for React (MSAL React).

Libraries

This tutorial uses the following libraries:

LIBRARY	DESCRIPTION
MSAL React	Microsoft Authentication Library for JavaScript React Wrapper
MSAL Browser	Microsoft Authentication Library for JavaScript v2 browser package

Get the completed code sample

Prefer to download this tutorial's completed sample project instead? To run the project by using a local web server, such as Nodejs, clone the [ms-identity-javascript-react-spa](https://github.com/Azure-Samples/ms-identity-javascript-react-spa) repository:

```
git clone https://github.com/Azure-Samples/ms-identity-javascript-react-spa
```

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

To continue with the tutorial and build the application yourself, move on to the next section, [Prerequisites](#).

Create your project

Once you have [Node.js](#) installed, open up a terminal window and then run the following commands:

```
npx create-react-app msal-react-tutorial # Create a new React app
cd msal-react-tutorial # Change to the app directory
npm install @azure/msal-browser @azure/msal-react # Install the MSAL packages
npm install react-bootstrap bootstrap # Install Bootstrap for styling
```

You have now bootstrapped a small React project using [Create React App](#). This will be the starting point the rest of this tutorial will build on. If you would like to see the changes to your app as you are working through this tutorial you can run the following command:

```
npm start
```

A browser window should be opened to your app automatically. If it does not, open your browser and navigate to <http://localhost:3000>. Each time you save a file with updated code the page will reload to reflect the changes.

Register your application

Follow the steps in [Single-page application: App registration](#) to create an app registration for your SPA by using the Azure portal.

In the [Redirect URI: MSAL.js 2.0 with auth code flow](#) step, enter `http://localhost:3000`, the default location where create-react-app will serve your application.

Configure your JavaScript SPA

1. Create a file named `authConfig.js` in the `src` folder to contain your configuration parameters for authentication, and then add the following code:

```

export const msalConfig = {
  auth: {
    clientId: "Enter_the_Application_Id_Here",
    authority: "Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Info_Here", // This is a URL (e.g.
    https://login.microsoftonline.com/{your tenant ID})
    redirectUri: "Enter_the_Redirect_Uri_Here",
  },
  cache: {
    cacheLocation: "sessionStorage", // This configures where your cache will be stored
    storeAuthStateInCookie: false, // Set this to "true" if you are having issues on IE11 or Edge
  }
};

// Add scopes here for ID token to be used at Microsoft identity platform endpoints.
export const loginRequest = {
  scopes: ["User.Read"]
};

// Add the endpoints here for Microsoft Graph API services you'd like to use.
export const graphConfig = {
  graphMeEndpoint: "Enter_the_Graph_Endpoint_Here/v1.0/me"
};

```

2. Modify the values in the `msalConfig` section as described here:

VALUE NAME	ABOUT
<code>Enter_the_Application_Id_Here</code>	The Application (client) ID of the application you registered.
<code>Enter_the_Cloud_Instance_Id_Here</code>	The Azure cloud instance in which your application is registered. For the main (or <i>global</i>) Azure cloud, enter <code>https://login.microsoftonline.com</code> . For national clouds (for example, China), you can find appropriate values in National clouds .
<code>Enter_the_Tenant_Info_Here</code>	Set to one of the following options: If your application supports <i>accounts in this organizational directory</i> , replace this value with the directory (tenant) ID or tenant name (for example, <code>contoso.microsoft.com</code>). If your application supports <i>accounts in any organizational directory</i> , replace this value with organizations . If your application supports <i>accounts in any organizational directory and personal Microsoft accounts</i> , replace this value with common . To restrict support to <i>personal Microsoft accounts only</i> , replace this value with consumers .
<code>Enter_the_Redirect_Uri_Here</code>	Replace with <code>http://localhost:3000</code> .
<code>Enter_the_Graph_Endpoint_Here</code>	The instance of the Microsoft Graph API the application should communicate with. For the global Microsoft Graph API endpoint, replace both instances of this string with <code>https://graph.microsoft.com</code> . For endpoints in national cloud deployments, see National cloud deployments in the Microsoft Graph documentation.

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

3. Open up the `src/index.js` file and add the following imports:

```
import "bootstrap/dist/css/bootstrap.min.css";
import { PublicClientApplication } from "@azure/msal-browser";
import { MsalProvider } from "@azure/msal-react";
import { msalConfig } from "./authConfig";
```

4. Underneath the imports in *src/index.js* create a `PublicClientApplication` instance using the configuration from step 1.

```
const msalInstance = new PublicClientApplication(msalConfig);
```

5. Find the `<App />` component in *src/index.js* and wrap it in the `MsalProvider` component. Your render function should look like this:

```
ReactDOM.render(
  <React.StrictMode>
    <MsalProvider instance={msalInstance}>
      <App />
    </MsalProvider>
  </React.StrictMode>,
  document.getElementById("root")
);
```

Sign in users

Create a folder in *src* called *components* and create a file inside this folder named *SignInButton.jsx*. Add the code from either of the following sections to invoke login using a pop-up window or a full-frame redirect:

Sign in using pop-ups

Add the following code to *src/components/SignInButton.jsx* to create a button component that will invoke a pop-up login when selected:

```
import React from "react";
import { useMsal } from "@azure/msal-react";
import { loginRequest } from "../authConfig";
import Button from "react-bootstrap/Button";

function handleLogin(instance) {
  instance.loginPopup(loginRequest).catch(e => {
    console.error(e);
  });
}

/**
 * Renders a button which, when selected, will open a popup for login
 */
export const SignInButton = () => {
  const { instance } = useMsal();

  return (
    <Button variant="secondary" className="ml-auto" onClick={() => handleLogin(instance)}>Sign in using
    Popup</Button>
  );
}
```

Sign in using redirects

Add the following code to *src/components/SignInButton.jsx* to create a button component that will invoke a redirect login when selected:

```

import React from "react";
import { useMsal } from "@azure/msal-react";
import { loginRequest } from "../authConfig";
import Button from "react-bootstrap/Button";

function handleLogin(instance) {
    instance.loginRedirect(loginRequest).catch(e => {
        console.error(e);
    });
}

/**
 * Renders a button which, when selected, will redirect the page to the login prompt
 */
export const SignInButton = () => {
    const { instance } = useMsal();

    return (
        <Button variant="secondary" className="ml-auto" onClick={() => handleLogin(instance)}>Sign in using
        Redirect</Button>
    );
}

```

Add the sign-in button

1. Create another file in the *components* folder named *PageLayout.jsx* and add the following code to create a navbar component that will contain the sign-in button you just created:

```

import React from "react";
import Navbar from "react-bootstrap/Navbar";
import { useIsAuthenticated } from "@azure/msal-react";
import { SignInButton } from "./SignInButton";

/**
 * Renders the navbar component with a sign-in button if a user is not authenticated
 */
export const PageLayout = (props) => {
    const isAuthenticated = useIsAuthenticated();

    return (
        <>
            <Navbar bg="primary" variant="dark">
                <a className="navbar-brand" href="/">MSAL React Tutorial</a>
                { isAuthenticated ? <span>Signed In</span> : <SignInButton /> }
            </Navbar>
            <h5><center>Welcome to the Microsoft Authentication Library For React Tutorial</center>
            </h5>
            <br />
            <br />
            {props.children}
        </>
    );
};

```

2. Now open *src/App.js* and add replace the existing content with the following code:

```

import React from "react";
import { PageLayout } from "./components/PageLayout";

function App() {
  return (
    <PageLayout>
      <p>This is the main app content!</p>
    </PageLayout>
  );
}

export default App;

```

Your app now has a sign-in button which is only displayed for unauthenticated users!

When a user selects the **Sign in using Popup** or **Sign in using Redirect** button for the first time, the `onClick` handler calls `loginPopup` (or `loginRedirect`) to sign in the user. The `loginPopup` 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, *msal.js* initiates the [authorization code flow](#).

At this point, a PKCE-protected authorization code is sent to the CORS-protected token endpoint and is exchanged for tokens. An ID token, access token, and refresh token are received by your application and processed by *msal.js*, and the information contained in the tokens is cached.

Sign users out

In *src/components* create a file named *SignInButton.jsx*. Add the code from either of the following sections to invoke logout using a pop-up window or a full-frame redirect:

Sign out using pop-ups

Add the following code to *src/components/SignInButton.jsx* to create a button component that will invoke a pop-up logout when selected:

```

import React from "react";
import { useMsal } from "@azure/msal-react";
import Button from "react-bootstrap/Button";

function handleLogout(instance) {
  instance.logoutPopup().catch(e => {
    console.error(e);
  });
}

/**
 * Renders a button which, when selected, will open a popup for logout
 */
export const SignInButton = () => {
  const { instance } = useMsal();

  return (
    <Button variant="secondary" className="ml-auto" onClick={() => handleLogout(instance)}>Sign out
    using Popup</Button>
  );
}

```

Sign out using redirects

Add the following code to *src/components/SignInButton.jsx* to create a button component that will invoke a redirect logout when selected:

```

import React from "react";
import { useMsal } from "@azure/msal-react";
import Button from "react-bootstrap/Button";

function handleLogout(instance) {
    instance.logoutRedirect().catch(e => {
        console.error(e);
    });
}

/**
 * Renders a button which, when selected, will redirect the page to the logout prompt
 */
export const SignOutButton = () => {
    const { instance } = useMsal();

    return (
        <Button variant="secondary" className="ml-auto" onClick={() => handleLogout(instance)}>Sign out using Redirect</Button>
    );
}

```

Add the sign-out button

Update your `PageLayout` component in `src/components/PageLayout.jsx` to render the new `SignOutButton` component for authenticated users. Your code should look like this:

```

import React from "react";
import Navbar from "react-bootstrap/Navbar";
import { useIsAuthenticated } from "@azure/msal-react";
import { SignInButton } from "./SignInButton";
import { SignOutButton } from "./SignOutButton";

/**
 * Renders the navbar component with a sign-in button if a user is not authenticated
 */
export const PageLayout = (props) => {
    const isAuthenticated = useIsAuthenticated();

    return (
        <>
            <Navbar bg="primary" variant="dark">
                <a className="navbar-brand" href="/">MSAL React Tutorial</a>
                { isAuthenticated ? <SignOutButton /> : <SignInButton /> }
            </Navbar>
            <h5><center>Welcome to the Microsoft Authentication Library For React Tutorial</center></h5>
            <br />
            <br />
            {props.children}
        </>
    );
}

```

Conditionally render components

In order to render certain components only for authenticated or unauthenticated users use the `AuthenticateTemplate` and/or `UnauthenticatedTemplate` as demonstrated below.

1. Add the following import to `src/App.js`:

```
import { AuthenticatedTemplate, UnauthenticatedTemplate } from "@azure/msal-react";
```

2. In order to render certain components only for authenticated users update your `App` function in `src/App.js` with the following code:

```
function App() {
  return (
    <PageLayout>
      <AuthenticatedTemplate>
        <p>You are signed in!</p>
      </AuthenticatedTemplate>
    </PageLayout>
  );
}
```

3. To render certain components only for unauthenticated users, such as a suggestion to login, update your `App` function in `src/App.js` with the following code:

```
function App() {
  return (
    <PageLayout>
      <AuthenticatedTemplate>
        <p>You are signed in!</p>
      </AuthenticatedTemplate>
      <UnauthenticatedTemplate>
        <p>You are not signed in! Please sign in.</p>
      </UnauthenticatedTemplate>
    </PageLayout>
  );
}
```

Acquire a token

1. Before calling an API, such as Microsoft Graph, you'll need to acquire an access token. Add a new component to `src/App.js` called `ProfileContent` with the following code:

```

function ProfileContent() {
    const { instance, accounts, inProgress } = useMsal();
    const [accessToken, setAccessToken] = useState(null);

    const name = accounts[0] && accounts[0].name;

    function RequestAccessToken() {
        const request = {
            ...loginRequest,
            account: accounts[0]
        };
    }

    // Silently acquires an access token which is then attached to a request for Microsoft Graph
    data
        instance.acquireTokenSilent(request).then((response) => {
            setAccessToken(response.accessToken);
        }).catch((e) => {
            instance.acquireTokenPopup(request).then((response) => {
                setAccessToken(response.accessToken);
            });
        });
    }

    return (
        <>
            <h5 className="card-title">Welcome {name}</h5>
            {accessToken ?
                <p>Access Token Acquired!</p>
                :
                <Button variant="secondary" onClick={RequestAccessToken}>Request Access
            Token</Button>
            }
        </>
    );
}

```

2. Update your imports in `src/App.js` to match the following:

```

import React, { useState } from "react";
import { PageLayout } from "./components/PageLayout";
import { AuthenticatedTemplate, UnauthenticatedTemplate, useMsal } from "@azure/msal-react";
import { loginRequest } from "./authConfig";
import Button from "react-bootstrap/Button";

```

3. Finally, add your new `ProfileContent` component as a child of the `AuthenticatedTemplate` in your `App` component in `src/App.js`. Your `App` component should look like this:

```

function App() {
    return (
        <PageLayout>
            <AuthenticatedTemplate>
                <ProfileContent />
            </AuthenticatedTemplate>
            <UnauthenticatedTemplate>
                <p>You are not signed in! Please sign in.</p>
            </UnauthenticatedTemplate>
        </PageLayout>
    );
}

```

The code above will render a button for signed in users, allowing them to request an access token for Microsoft Graph when the button is selected.

After a user signs in, your app shouldn't ask users to reauthenticate every time they need to access a protected resource (that is, to request a token). To prevent such reauthentication requests, call `acquireTokenSilent` which will first look for a cached, unexpired access token then, if needed, use the refresh token to obtain a new access token. There are some situations, however, where you might need to force users to interact with the Microsoft identity platform. For example:

- Users need to re-enter their credentials because the session has expired.
- The refresh token 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). In that window, users need to interact by confirming their credentials, giving consent to the required resource, or completing the two-factor authentication.

NOTE

If you're using Internet Explorer, we recommend that you use the `loginRedirect` and `acquireTokenRedirect` methods due to a [known issue](#) with Internet Explorer and pop-up windows.

Call the Microsoft Graph API

1. Create file named `graph.js` in the `src` folder and add the following code for making REST calls to the Microsoft Graph API:

```
import { graphConfig } from "./authConfig";

/**
 * Attaches a given access token to a Microsoft Graph API call. Returns information about the user
 */
export async function callMsGraph(accessToken) {
    const headers = new Headers();
    const bearer = `Bearer ${accessToken}`;

    headers.append("Authorization", bearer);

    const options = {
        method: "GET",
        headers: headers
    };

    return fetch(graphConfig.graphMeEndpoint, options)
        .then(response => response.json())
        .catch(error => console.log(error));
}
```

2. Next create a file named `ProfileData.jsx` in `src/components` and add the following code:

```

import React from "react";

/**
 * Renders information about the user obtained from Microsoft Graph
 */
export const ProfileData = (props) => {
  return (
    <div id="profile-div">
      <p><strong>First Name: </strong> {props.graphData.givenName}</p>
      <p><strong>Last Name: </strong> {props.graphData.surname}</p>
      <p><strong>Email: </strong> {props.graphData.userPrincipalName}</p>
      <p><strong>Id: </strong> {props.graphData.id}</p>
    </div>
  );
};

```

3. Next, open `src/App.js` and add these to the imports:

```

import { ProfileData } from "./components/ProfileData";
import { callMsGraph } from "./graph";

```

4. Finally, update your `ProfileContent` component in `src/App.js` to call Microsoft Graph and display the profile data after acquiring the token. Your `ProfileContent` component should look like this:

```

function ProfileContent() {
  const { instance, accounts } = useMsal();
  const [graphData, setGraphData] = useState(null);

  const name = accounts[0] && accounts[0].name;

  function RequestProfileData() {
    const request = {
      ...loginRequest,
      account: accounts[0]
    };

    // Silently acquires an access token which is then attached to a request for Microsoft Graph
    data
    instance.acquireTokenSilent(request).then((response) => {
      callMsGraph(response.accessToken).then(response => setGraphData(response));
    }).catch((e) => {
      instance.acquireTokenPopup(request).then((response) => {
        callMsGraph(response.accessToken).then(response => setGraphData(response));
      });
    });
  }

  return (
    <>
      <h5 className="card-title">Welcome {name}</h5>
      {graphData ?
        <ProfileData graphData={graphData} />
        :
        <Button variant="secondary" onClick={RequestProfileData}>Request Profile
        Information</Button>
      }
    </>
  );
}

```

In the changes made above, 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*. In the sample application created in this tutorial, the protected resource is the Microsoft Graph API *me* endpoint which displays the signed-in user's profile information.

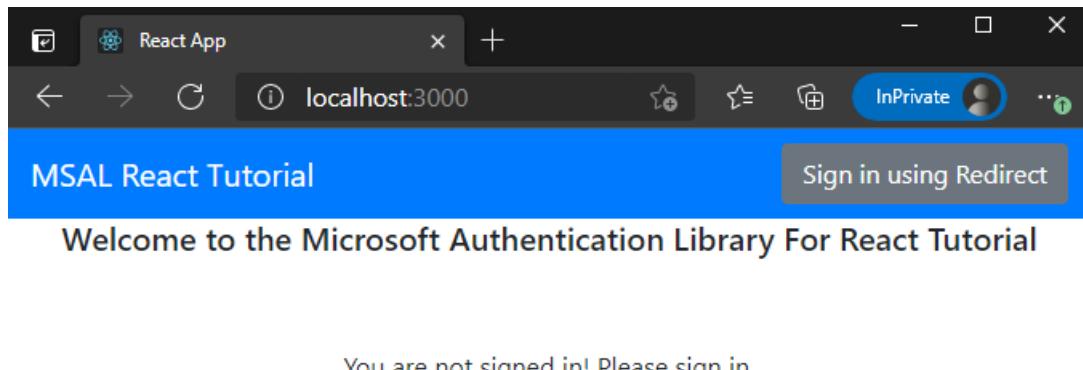
Test your application

You've completed creation of the application and are now ready to launch the web server and test the app's functionality.

1. Serve your app by running the following command from within the root of your project folder:

```
npm start
```

2. A browser window should be opened to your app automatically. If it does not, open your browser and navigate to `http://localhost:3000`. You should see a page that looks like the one below.



3. Select the sign-in button to sign in.

Provide consent for application access

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

Sign in to your account - Google Chrome

login.microsoftonline.com/common/SSPR/End

 Microsoft

kendall@contoso.com

Permissions requested

Contoso JavaScript SPA
unverified

This application is not published by Microsoft.

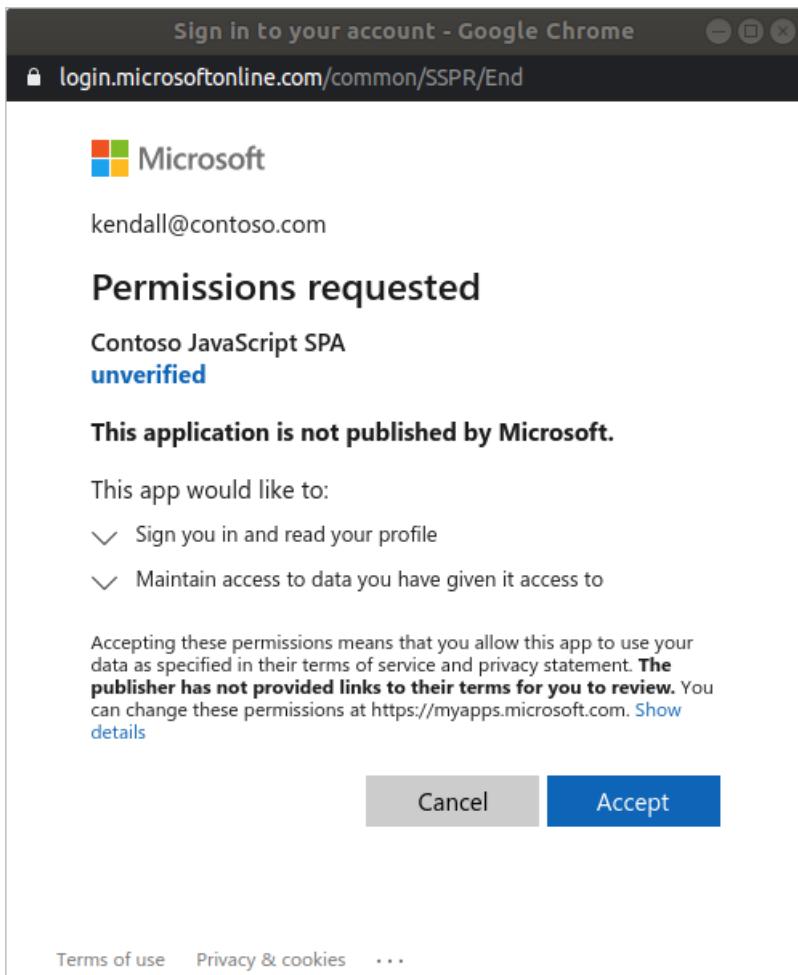
This app would like to:

- ✓ Sign you in and read your profile
- ✓ Maintain access to data you have given it access to

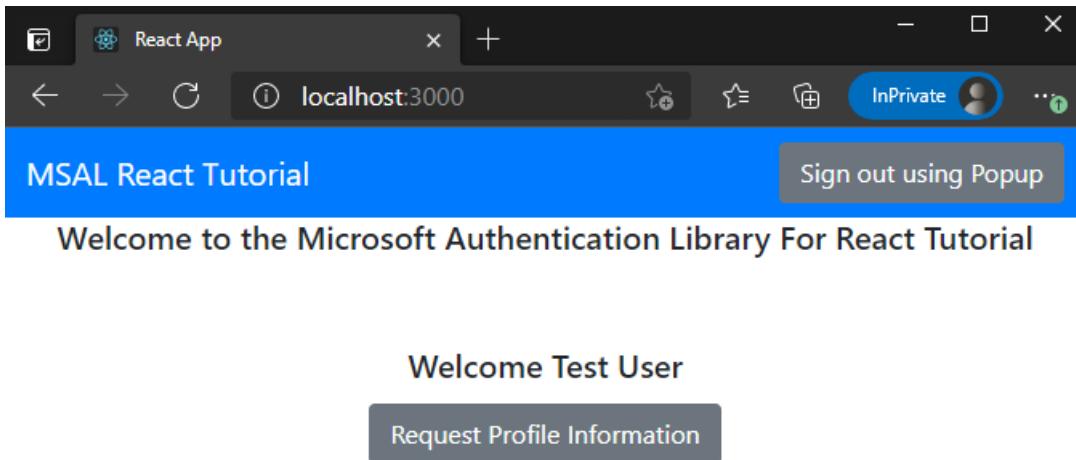
Accepting these permissions means that you allow this app to use your data as specified in their terms of service and privacy statement. **The publisher has not provided links to their terms for you to review.** You can change these permissions at <https://myapps.microsoft.com>. [Show details](#)

[Cancel](#) [Accept](#)

[Terms of use](#) [Privacy & cookies](#) ...



If you consent to the requested permissions, the web application displays your name, signifying a successful login:

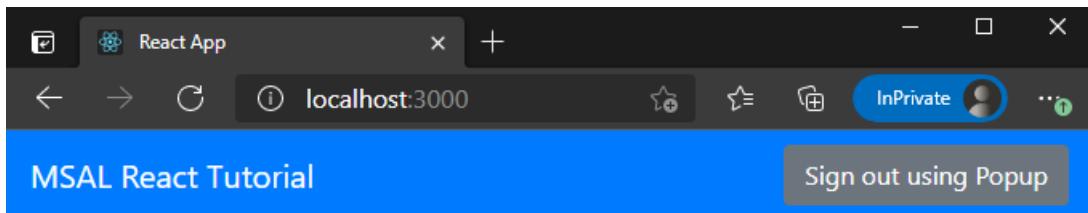


The screenshot shows a Microsoft Edge browser window with the following details:

- Title Bar:** React App
- Address Bar:** localhost:3000
- Toolbar:** InPrivate, ...
- Content Area:**
 - Header: MSAL React Tutorial
 - Header: Sign out using Popup
 - Main Content: Welcome to the Microsoft Authentication Library For React Tutorial
 - Below content: Welcome Test User
 - Button: Request Profile Information

Call the Graph API

After you sign in, select See Profile to view the user profile information returned in the response from the call to the Microsoft Graph API:



Welcome Test User

First Name: Test

Last Name: User

Email: testuser@contoso.com

Id: undefined

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 in the Azure 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 `Mail.Read` scope in order to list the user's email.

As you add scopes, your users might be prompted to provide additional consent for the added scopes.

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

If you'd like to dive deeper into JavaScript single-page application development on the Microsoft identity platform, see our multi-part scenario series:

[Scenario: Single-page application](#)

Microsoft identity platform code samples

4/12/2022 • 9 minutes to read • [Edit Online](#)

These code samples are built and maintained by Microsoft to demonstrate usage of our authentication libraries with the Microsoft identity platform. Common authentication and authorization scenarios are implemented in several [application types](#), development languages, and frameworks.

- Sign in users to web applications and provide authorized access to protected web APIs.
- Protect a web API by requiring an access token to perform API operations.

Each code sample includes a *README.md* file describing how to build the project (if applicable) and run the sample application. Comments in the code help you understand how these libraries are used in the application to perform authentication and authorization by using the identity platform.

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Angular	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Call .NET Core web API• Call .NET Core web API (B2C)• Call Microsoft Graph via OBO• Call .NET Core web API using PoP• Use App Roles for access control• Use Security Groups for access control• Deploy to Azure Storage and App Service	MSAL Angular	<ul style="list-style-type: none">• Authorization code with PKCE• On-behalf-of (OBO)• Proof of Possession (PoP)
Blazor WebAssembly	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Deploy to Azure App Service	MSAL.js	Implicit Flow

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
JavaScript	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API via OBO and CA • Deploy to Azure Storage and App Service 	MSAL.js	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA)
React	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API using PoP • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service • Deploy to Azure Static Web Apps 	MSAL React	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA) • Proof of Possession (PoP)

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Customize token cache • Call Graph (multi-tenant) • Call Azure REST APIs • Protect web API • Protect web API (B2C) • Protect multi-tenant web API • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service 	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	<ul style="list-style-type: none"> • OpenID connect • Authorization code • On-Behalf-Of

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Blazor	Blazor Server Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call web API • Call web API (B2C) 	MSAL.NET	Authorization code Grant Flow
ASP.NET Core	Advanced Token Cache Scenarios	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	On-Behalf-Of (OBO)
ASP.NET Core	Use the Conditional Access auth context to perform step-up authentication	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	Authorization code
ASP.NET Core	Active Directory FS to Azure AD migration	MSAL.NET	<ul style="list-style-type: none"> • SAML • OpenID connect
ASP.NET	<ul style="list-style-type: none"> • Microsoft Graph Training Sample • Sign in users and call Microsoft Graph • Sign in users and call Microsoft Graph with admin restricted scope • Quickstart: Sign in users 	MSAL.NET	<ul style="list-style-type: none"> • OpenID connect • Authorization code
Java Spring	Azure AD Spring Boot Starter Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Groups for access control • Deploy to Azure App Service 	<ul style="list-style-type: none"> • MSAL Java • Azure AD Boot Starter 	Authorization code
Java Servlets	Spring-less Servlet Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure App Service 	MSAL Java	Authorization code
Java	Sign in users and call Microsoft Graph	MSAL Java	Authorization code
Java Spring	Sign in users and call Microsoft Graph via OBO • Web API	MSAL Java	<ul style="list-style-type: none"> • Authorization code • On-Behalf-Of (OBO)

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js Express	Express web app series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service • Use App Roles for access control • Use Security Groups for access control • Web app that sign in users 	MSAL Node	Authorization code
Python Flask	Flask Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Sign in users and call Microsoft Graph • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Python Django	Django Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Ruby	Graph Training <ul style="list-style-type: none"> • Sign in users and call Microsoft Graph 	OmniAuth OAuth2	Authorization code

Web API

The following samples show how to protect a web API with the Microsoft identity platform, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET	Call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
ASP.NET Core	Sign in users and call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
Java	Sign in users	MSAL Java	On-Behalf-Of (OBO)
Node.js	<ul style="list-style-type: none"> • Protect a Node.js web API • Protect a Node.js Web API with Azure AD B2C 	MSAL Node	Authorization bearer

Desktop

The following samples show public client desktop applications that access the Microsoft Graph API, or your own

web API in the name of the user. Apart from the *Desktop (Console) with Web Authentication Manager (WAM)* sample, all these client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> • Call Microsoft Graph • Call Microsoft Graph with token cache • Call Microsoft Graph with custom web UI HTML • Call Microsoft Graph with custom web browser • Sign in users with device code flow 	MSAL.NET	<ul style="list-style-type: none"> • Authorization code with PKCE • Device code
.NET	<ul style="list-style-type: none"> • Call Microsoft Graph with daemon console • Call web API with daemon console 	MSAL.NET	Authorization code with PKCE
.NET	Invoke protected API with integrated Windows authentication	MSAL.NET	Integrated Windows authentication
Java	Call Microsoft Graph	MSAL Java	Integrated Windows authentication
Node.js	Sign in users	MSAL Node	Authorization code with PKCE
PowerShell	Call Microsoft Graph by signing in users using username/password	MSAL.NET	Resource owner password credentials
Python	Sign in users	MSAL Python	Resource owner password credentials
Universal Window Platform (UWP)	Call Microsoft Graph	MSAL.NET	Web account manager
Windows Presentation Foundation (WPF)	Sign in users and call Microsoft Graph	MSAL.NET	Authorization code with PKCE
XAML	<ul style="list-style-type: none"> • Sign in users and call ASP.NET core web API • Sign in users and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Mobile

The following samples show public client mobile applications that access the Microsoft Graph API, or your own web API in the name of the user. These client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
iOS	<ul style="list-style-type: none"> Call Microsoft Graph native Call Microsoft Graph with Azure AD nxauth 	MSAL iOS	Authorization code with PKCE
Java	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Kotlin	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Xamarin	<ul style="list-style-type: none"> Sign in users and call Microsoft Graph Sign in users with broker and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Service / daemon

The following samples show an application that accesses the Microsoft Graph API with its own identity (with no user).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> Call Microsoft Graph Call web API Call own web API Using managed identity and Azure key vault 	MSAL.NET	Client credentials grant
ASP.NET	Multi-tenant with Microsoft identity platform endpoint	MSAL.NET	Client credentials grant
Java	Call Microsoft Graph	MSAL Java	Client credentials grant
Node.js	Sign in users and call web API	MSAL Node	Client credentials grant
Python	<ul style="list-style-type: none"> Call Microsoft Graph with secret Call Microsoft Graph with certificate 	MSAL Python	Client credentials grant

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, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET	.NET Azure function web API secured by Azure AD	MSAL.NET	Authorization code

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Node.js Azure function web API secured by Azure AD	MSAL Node	Authorization bearer
Node.js	Call Microsoft Graph API on behalf of a user	MSAL Node	On-Behalf-Of (OBO)
Python	Python Azure function web API secured by Azure AD	MSAL Python	Authorization code

Headless

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 the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET core	Invoke protected API from text-only device	MSAL.NET	Device code
Java	Sign in users and invoke protected API	MSAL Java	Device code
Python	Call Microsoft Graph	MSAL Python	Device code

Microsoft Teams applications

The following sample illustrates Microsoft Teams Tab application that signs in users. Additionally it demonstrates how to call Microsoft Graph API with the user's identity using the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Teams Tab app: single sign-on (SSO) and call Microsoft Graph	MSAL Node	On-Behalf-Of (OBO)

Multi-tenant SaaS

The following samples show how to configure your application to accept sign-ins from any Azure Active Directory (Azure AD) tenant. Configuring your application to be *multi-tenant* means that you can offer a **Software as a Service** (SaaS) application to many organizations, allowing their users to be able to sign-in to your application after providing consent.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core MVC web application calls Microsoft Graph API	MSAL.NET	OpenID connect
ASP.NET Core	ASP.NET Core MVC web application calls ASP.NET Core Web API	MSAL.NET	Authorization code

Next steps

If you'd like to delve deeper into more sample code, see:

- [Sign in users and call the Microsoft Graph API from an Angular](#)
- [Sign in users in a Nodejs and Express web app](#)
- [Call the Microsoft Graph API from a Universal Windows Platform](#)

Scenario: Single-page application

4/12/2022 • 2 minutes to read • [Edit Online](#)

Learn all you need to build a single-page application (SPA). For instructions regarding Azure Static Web Apps, see [Authentication and Authorization for Static Web Apps](#) instead.

Getting started

If you haven't already, create your first app by completing the JavaScript SPA quickstart:

[Quickstart: Single-page application](#)

Overview

Many modern web applications are built as client-side single-page applications. Developers write them by using JavaScript or a SPA framework such as Angular, Vue, and React. These applications run on a web browser and have different authentication characteristics than traditional server-side web applications.

The Microsoft identity platform provides **two** options to enable single-page applications to sign in users and get tokens to access back-end services or web APIs:

- [OAuth 2.0 Authorization code flow \(with PKCE\)](#). The authorization code flow allows the application to exchange an authorization code for ID tokens to represent the authenticated user and Access tokens needed to call protected APIs.

Proof Key for Code Exchange, or *PKCE*, is an extension to the authorization code flow to prevent authorization code injection attacks. This IETF standard mitigates the threat of having an authorization code intercepted and enables secure OAuth exchange from public clients as documented in [RFC 7636](#). In addition, it returns **refresh** tokens that provide long-term access to resources on behalf of users without requiring interaction from those users.

Using the authorization code flow with PKCE is the more secure and **recommended** authorization approach, not only in native and browser-based JavaScript apps, but for every other type of OAuth client.



- [OAuth 2.0 implicit flow](#). The implicit grant flow allows the application to get ID and Access tokens. Unlike the authorization code flow, implicit grant flow does not return a **Refresh token**.



This authentication flow does not include application scenarios that use cross-platform JavaScript frameworks such as Electron and React-Native. They require further capabilities for interaction with the native platforms.

Specifics

To enable this scenario for your application, you need:

- Application registration with Azure Active Directory (Azure AD). The registration steps differ between the implicit grant flow and authorization code flow.
- Application configuration with the registered application properties, such as the application ID.
- Using Microsoft Authentication Library for JavaScript (MSAL.js) to do the authentication flow to sign in and acquire tokens.

Recommended reading

If you're new to identity and access management (IAM) with OAuth 2.0 and OpenID Connect, or even just new to IAM on the Microsoft identity platform, the following set of articles should be high on your reading list.

Although not required reading before completing your first quickstart or tutorial, they cover topics integral to the platform, and familiarity with them will help you on your path as you build more complex scenarios.

Authentication and authorization

- [Authentication basics](#)
- [ID tokens](#)
- [Access tokens](#)

Microsoft identity platform

- [Audiences](#)
- [Applications and service principals](#)
- [Permissions and consent](#)

Next steps

Move on to the next article in this scenario, [App registration](#).

Single-page application: App registration

4/12/2022 • 3 minutes to read • [Edit Online](#)

To register a single-page application (SPA) in the Microsoft identity platform, complete the following steps. The registration steps differ between MSAL.js 1.0, which supports the implicit grant flow, and MSAL.js 2.0, which supports the authorization code flow with PKCE.

Create the app registration

For both MSAL.js 1.0- and 2.0-based applications, start by completing the following steps to create the initial app registration.

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directory + subscription** filter  in the top menu to select the tenant in which you want to register an application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application. Users of your app might see this name, and you can change it later.
6. Choose the **Supported account types** for the application. Do NOT enter a **Redirect URI**. For a description of the different account types, see the [Register an application](#).
7. Select **Register** to create the app registration.

Next, configure the app registration with a **Redirect URI** to specify where the Microsoft identity platform should redirect the client along with any security tokens. Use the steps appropriate for the version of MSAL.js you're using in your application:

- [MSAL.js 2.0 with auth code flow](#) (recommended)
- [MSAL.js 1.0 with implicit flow](#)

Redirect URI: [MSAL.js 2.0 with auth code flow](#)

Follow these steps to add a redirect URI for an app that uses MSAL.js 2.0 or later. MSAL.js 2.0+ supports the authorization code flow with PKCE and CORS in response to [browser third party cookie restrictions](#). The implicit grant flow is not supported in MSAL.js 2.0+.

1. In the Azure portal, select the app registration you created earlier in [Create the app registration](#).
2. Under **Manage**, select **Authentication > Add a platform**.
3. Under **Web applications**, select the **Single-page application** tile.
4. Under **Redirect URIs**, enter a [redirect URI](#). Do NOT select either checkbox under **Implicit grant and hybrid flows**.
5. Select **Configure** to finish adding the redirect URI.

You've now completed the registration of your single-page application (SPA) and configured a redirect URI to which the client will be redirected and any security tokens will be sent. By configuring your redirect URI using the **Single-page application** tile in the **Add a platform** pane, your application registration is configured to support the authorization code flow with PKCE and CORS.

Follow the [tutorial](#) for further guidance.

Redirect URI: MSAL.js 1.0 with implicit flow

Follow these steps to add a redirect URI for a single-page app that uses MSAL.js 1.3 or earlier and the implicit grant flow. Applications that use MSAL.js 1.3 or earlier do not support the auth code flow.

1. In the Azure portal, select the app registration you created earlier in [Create the app registration](#).
2. Under **Manage**, select **Authentication** > **Add a platform**.
3. Under **Web applications**, select **Single-page application** tile.
4. Under **Redirect URIs**, enter a [redirect URI](#).
5. Enable the **Implicit grant and hybrid flows**:
 - If your application signs in users, select **ID tokens**.
 - If your application also needs to call a protected web API, select **Access tokens**. For more information about these token types, see [ID tokens](#) and [Access tokens](#).
6. Select **Configure** to finish adding the redirect URI.

You've now completed the registration of your single-page application (SPA) and configured a redirect URI to which the client will be redirected and any security tokens will be sent. By selecting one or both of **ID tokens** and **Access tokens**, you've enabled the implicit grant flow.

Note about authorization flows

By default, an app registration created by using single-page application platform configuration enables the authorization code flow. To take advantage of this flow, your application must use MSAL.js 2.0 or later.

As mentioned previously, single-page applications using MSAL.js 1.3 are restricted to the implicit grant flow. Current [OAuth 2.0 best practices](#) recommend using the authorization code flow rather than the implicit flow for SPAs. Having limited-lifetime refresh tokens also helps your application adapt to [modern browser cookie privacy limitations](#), like Safari ITP.

When all your production single-page applications represented by an app registration are using MSAL.js 2.0 and the authorization code flow, uncheck the implicit grant settings on the app registration's **Authentication** pane in the Azure portal. Applications using MSAL.js 1.x and the implicit flow can continue to function, however, if you leave the implicit flow enabled (checked).

Next steps

Configure your app's code to use the app registration you created in the previous steps: [App's code configuration](#).

Single-page application: Code configuration

4/12/2022 • 2 minutes to read • [Edit Online](#)

Learn how to configure the code for your single-page application (SPA).

Microsoft libraries supporting single-page apps

The following Microsoft libraries support single-page apps:

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
Angular	MSAL Angular v2²	msal-angular	Tutorial	✓	✓	GA
Angular	MSAL Angular³	msal-angular	—	✓	✓	GA
AngularJS	MSAL AngularJS³	msal-angularjs	—	✓	✓	Public preview
JavaScript	MSAL.js v2²	msal-browser	Tutorial	✓	✓	GA
JavaScript	MSAL.js 1.0³	msal-core	—	✓	✓	GA
React	MSAL React²	msal-react	Tutorial	✓	✓	GA

¹ Supplemental terms of use for Microsoft Azure Previews apply to libraries in *Public preview*.

² Auth code flow with PCKE only (Recommended).

³ Implicit grant flow only.

Application code configuration

In an MSAL library, the application registration information is passed as configuration during the library initialization.

- [JavaScript \(MSAL.js v2\)](#)
- [JavaScript \(MSAL.js v1\)](#)
- [Angular \(MSAL.js v2\)](#)
- [Angular \(MSAL.js v1\)](#)
- [React](#)

```
import * as Msal from "@azure/msal-browser"; // if using CDN, 'Msal' will be available in global scope

// Configuration object constructed.
const config = {
    auth: {
        clientId: 'your_client_id'
    }
};

// create PublicClientApplication instance
const publicClientApplication = new Msal.PublicClientApplication(config);
```

For more information on the configurable options, see [Initializing application with MSAL.js](#).

Next steps

Move on to the next article in this scenario, [Sign-in and sign-out](#).

Single-page application: Sign-in and Sign-out

4/12/2022 • 10 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 need an authenticated user context. You can sign in users to your application in MSAL.js in two ways:

- [Pop-up window](#), by using the `loginPopup` method
- [Redirect](#), by using the `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 details, see [SSO without MSAL.js login](#).

Choosing between a pop-up or redirect experience

The choice between a pop-up or redirect experience depends on your application flow:

- If you don't want users to move away from your main application page during authentication, we recommend the pop-up method. Because the authentication redirect happens in a pop-up window, the state of the main application is preserved.
- If users have browser constraints or policies where pop-up windows are disabled, you can use the redirect method. Use the redirect method with the Internet Explorer browser, because there are [known issues with pop-up windows on Internet Explorer](#).

Sign-in with a pop-up window

- [JavaScript \(MSAL.js v2\)](#)
- [JavaScript \(MSAL.js v1\)](#)
- [Angular \(MSAL.js v2\)](#)
- [Angular \(MSAL.js v1\)](#)
- [React](#)

```
const config = {
  auth: {
    clientId: 'your_app_id',
    redirectUri: "your_app_redirect_uri", //defaults to application start page
    postLogoutRedirectUri: "your_app_logout_redirect_uri"
  }
}

const loginRequest = {
  scopes: ["User.ReadWrite"]
}

let accountId = "";

const myMsal = new PublicClientApplication(config);

myMsal.loginPopup(loginRequest)
  .then(function (loginResponse) {
    accountId = loginResponse.account.homeAccountId;
    // Display signed-in user content, call API, etc.
  }).catch(function (error) {
    //login failure
    console.log(error);
  });
}
```

Sign-in with redirect

- [JavaScript \(MSAL.js v2\)](#)
- [JavaScript \(MSAL.js v1\)](#)
- [Angular \(MSAL.js v2\)](#)
- [Angular \(MSAL.js v1\)](#)
- [React](#)

```

const config = {
  auth: {
    clientId: 'your_app_id',
    redirectUri: "your_app_redirect_uri", //defaults to application start page
    postLogoutRedirectUri: "your_app_logout_redirect_uri"
  }
}

const loginRequest = {
  scopes: ["User.ReadWrite"]
}

let accountId = "";

const myMsal = new PublicClientApplication(config);

function handleResponse(response) {
  if (response !== null) {
    accountId = response.account.homeAccountId;
    // Display signed-in user content, call API, etc.
  } else {
    // In case multiple accounts exist, you can select
    const currentAccounts = myMsal.getAllAccounts();

    if (currentAccounts.length === 0) {
      // no accounts signed-in, attempt to sign a user in
      myMsal.loginRedirect(loginRequest);
    } else if (currentAccounts.length > 1) {
      // Add choose account code here
    } else if (currentAccounts.length === 1) {
      accountId = currentAccounts[0].homeAccountId;
    }
  }
}

myMsal.handleRedirectPromise().then(handleResponse);

```

Sign-out with a pop-up window

MSAL.js v2 provides a `logoutPopup` method that clears the cache in browser storage and opens a pop-up window to the Azure Active Directory (Azure AD) sign-out page. After sign-out, Azure AD redirects the pop-up back to your application and MSAL.js will close the pop-up.

You can configure the URI to which Azure AD should redirect after sign-out by setting `postLogoutRedirectUri`. This URI should be registered as a redirect URI in your application registration.

You can also configure `logoutPopup` to redirect the main window to a different page, such as the home page or sign-in page, after logout is complete by passing `mainWindowRedirectUri` as part of the request.

- [JavaScript \(MSAL.js v2\)](#)
- [JavaScript \(MSAL.js v1\)](#)
- [Angular \(MSAL.js v2\)](#)
- [Angular \(MSAL.js v1\)](#)
- [React](#)

```

const config = {
  auth: {
    clientId: 'your_app_id',
    redirectUri: "your_app_redirect_uri", // defaults to application start page
    postLogoutRedirectUri: "your_app_logout_redirect_uri"
  }
}

const myMsal = new PublicClientApplication(config);

// you can select which account application should sign out
const logoutRequest = {
  account: myMsal.getAccountByHomeId(homeAccountId),
  mainWindowRedirectUri: "your_app_main_window_redirect_uri"
}

await myMsal.logoutPopup(logoutRequest);

```

Sign-out with a redirect

MSAL.js provides a `logout` method in v1, and `logoutRedirect` method in v2, that clears the cache in browser storage and redirects the window to the Azure Active Directory (Azure AD) sign-out page. After sign-out, Azure AD redirects back to the page that invoked logout by default.

You can configure the URI to which it should redirect after sign-out by setting `postLogoutRedirectUri`. This URI should be registered as a redirect URI in your application registration.

- [JavaScript \(MSAL.js v2\)](#)
- [JavaScript \(MSAL.js v1\)](#)
- [Angular \(MSAL.js v2\)](#)
- [Angular \(MSAL.js v1\)](#)
- [React](#)

```

const config = {
  auth: {
    clientId: 'your_app_id',
    redirectUri: "your_app_redirect_uri", //defaults to application start page
    postLogoutRedirectUri: "your_app_logout_redirect_uri"
  }
}

const myMsal = new PublicClientApplication(config);

// you can select which account application should sign out
const logoutRequest = {
  account: myMsal.getAccountByHomeId(homeAccountId)
}

myMsal.logoutRedirect(logoutRequest);

```

Next steps

Move on to the next article in this scenario, [Acquiring a token for the app](#).

Single-page application: Acquire a token to call an API

4/12/2022 • 9 minutes to read • [Edit Online](#)

The pattern for acquiring tokens for APIs with [MSAL.js](#) is to first attempt a silent token request by using the `acquireTokenSilent` method. When this method is called, the library first checks the cache in browser storage to see if a valid token exists and returns it. When no valid token is in the cache, it attempts to use its refresh token to get the token. If the refresh token's 24-hour lifetime has expired, MSAL.js will open a hidden iframe to silently request a new authorization code, which it will exchange for a new, valid refresh token. 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 might fail for reasons like a password change or updated conditional access policies. More often, failures are due to the refresh token's 24-hour lifetime expiring and [the browser blocking 3rd party cookies](#), which prevents the use of hidden iframes to continue authenticating the user. In these cases, you should invoke one of the interactive methods (which may prompt the user) to acquire tokens:

- [Pop-up window](#), by using `acquireTokenPopup`
- [Redirect](#), by using `acquireTokenRedirect`

Choose between a pop-up or redirect experience

The choice between a pop-up or redirect experience depends on your application flow:

- If you don't want users to move away from your main application page during authentication, we recommend the pop-up method. Because the authentication redirect happens in a pop-up window, the state of the main application is preserved.
- If users have browser constraints or policies where pop-up windows are disabled, you can use the redirect method. Use the redirect method with the Internet Explorer browser, because there are [known issues with pop-up windows on Internet Explorer](#).

You can set the API scopes that you want the access token to include when it's building the access token request. Note that all requested scopes might not be granted in the access token. That depends on the user's consent.

Acquire a token with a pop-up window

- [JavaScript \(MSAL.js v2\)](#)
- [JavaScript \(MSAL.js v1\)](#)
- [Angular \(MSAL.js v2\)](#)
- [Angular \(MSAL.js v1\)](#)
- [React](#)

The following code combines the previously described pattern with the methods for a pop-up experience:

```

// MSAL.js v2 exposes several account APIs, logic to determine which account to use is the responsibility of
// the developer
const account = publicClientApplication.getAllAccounts()[0];

const accessTokenRequest = {
    scopes: ["user.read"],
    account: account
}

publicClientApplication.acquireTokenSilent(accessTokenRequest).then(function(accessTokenResponse) {
    // Acquire token silent success
    let accessToken = accessTokenResponse.accessToken;
    // Call your API with token
    callApi(accessToken);
}).catch(function (error) {
    //Acquire token silent failure, and send an interactive request
    if (error instanceof InteractionRequiredAuthError) {
        publicClientApplication.acquireTokenPopup(accessTokenRequest).then(function(accessTokenResponse) {
            // Acquire token interactive success
            let accessToken = accessTokenResponse.accessToken;
            // Call your API with token
            callApi(accessToken);
        }).catch(function(error) {
            // Acquire token interactive failure
            console.log(error);
        });
    }
    console.log(error);
});

```

Acquire a token with a redirect

- [JavaScript \(MSAL.js v2\)](#)
- [JavaScript \(MSAL.js v1\)](#)
- [Angular \(MSAL.js v2\)](#)
- [Angular \(MSAL.js v1\)](#)
- [React](#)

The following pattern is as described earlier but shown with a redirect method to acquire tokens interactively. You'll need to call and await `handleRedirectPromise` on page load.

```
const redirectResponse = await publicClientApplication.handleRedirectPromise();
if (redirectResponse !== null) {
    // Acquire token silent success
    let accessToken = redirectResponse.accessToken;
    // Call your API with token
    callApi(accessToken);
} else {
    // MSAL.js v2 exposes several account APIs, logic to determine which account to use is the
    responsibility of the developer
    const account = publicClientApplication.getAllAccounts()[0];

    const accessTokenRequest = {
        scopes: ["user.read"],
        account: account
    }

    publicClientApplication.acquireTokenSilent(accessTokenRequest).then(function(accessTokenResponse) {
        // Acquire token silent success
        // Call API with token
        let accessToken = accessTokenResponse.accessToken;
        // Call your API with token
        callApi(accessToken);
    }).catch(function (error) {
        //Acquire token silent failure, and send an interactive request
        console.log(error);
        if (error instanceof InteractionRequiredAuthError) {
            publicClientApplication.acquireTokenRedirect(accessTokenRequest);
        }
    });
}
}
```

Next steps

Move on to the next article in this scenario, [Calling a web API](#).

Single-page application: Call a web API

4/12/2022 • 2 minutes to read • [Edit Online](#)

We recommend that you call the `acquireTokenSilent` method to acquire or renew an access token before you call a web API. After you have a token, you can call a protected web API.

Call a web API

- [JavaScript](#)
- [Angular](#)

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
  })
```

Next steps

Move on to the next article in this scenario, [Move to production](#).

Single-page application: Move to production

4/12/2022 • 2 minutes to read • [Edit Online](#)

Now that you know how to acquire a token to call web APIs, here are some things to consider when moving your application to production.

Enable logging

To help in debugging and authentication failure troubleshooting scenarios, the Microsoft Authentication Library provides built-in logging support. Logging is each library is covered in the following articles:

- [Logging in MSAL.NET](#)
- [Logging in MSAL for Android](#)
- [Logging in MSAL.js](#)
- [Logging in MSAL for iOS/macOS](#)
- [Logging in MSAL for Java](#)
- [Logging in MSAL for Python](#)

Here are some suggestions for data collection:

- Users might ask for help when they have problems. A best practice is to capture and temporarily store logs. Provide a location where users can upload the logs. MSAL provides logging extensions to capture detailed information about authentication.
- If telemetry is available, enable it through MSAL to gather data about how users sign in to your app.

Validate your integration

Test your integration by following the [Microsoft identity platform integration checklist](#).

Build for resilience

Learn how to increase resiliency in your app. For details, see [Increase resilience of authentication and authorization applications you develop](#)

Deploy your app

Check out a [deployment sample](#) for learning how to deploy your SPA and Web API projects with Azure Storage and Azure App Services, respectively.

Code samples

These code samples demonstrate several key operations for a single-page app.

- [SPA with an ASP.NET back-end](#): How to get tokens for your own back-end web API (ASP.NET Core) by using [MSAL.js](#).
- [Node.js Web API \(Azure AD\)](#): How to validate access tokens for your back-end web API (Node.js) by using [passport-azure-ad](#).
- [SPA with Azure AD B2C](#): How to use [MSAL.js](#) to sign in users in an app that's registered with [Azure](#)

Active Directory B2C (Azure AD B2C).

- [Node.js web API \(Azure AD B2C\)](#): How to use `passport-azure-ad` to validate access tokens for apps registered with **Azure Active Directory B2C** (Azure AD B2C).

Next steps

- [JavaScript SPA tutorial](#): Deep dive to how to sign in users and get an access token to call the **Microsoft Graph API** by using `MSAL.js`.

Initialize client applications using MSAL.js

4/12/2022 • 3 minutes to read • [Edit Online](#)

This article describes initializing the 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. Such clients don't store secrets because the browser context is openly accessible.

To learn more about the client application types and application configuration options, see [Public and confidential client apps in MSAL](#).

Prerequisites

Before initializing an application, you first need to [register it with the Azure portal](#), establishing a trust relationship between your application and the Microsoft identity platform.

After registering your app, you'll need some or all of the following values that can be found in the Azure portal.

VALUE	REQUIRED	DESCRIPTION
Application (client) ID	Required	A GUID that uniquely identifies your application within the Microsoft identity platform.
Authority	Optional	The identity provider URL (the <i>instance</i>) and the <i>sign-in audience</i> for your application. The instance and sign-in audience, when concatenated, make up the <i>authority</i> .
Directory (tenant) ID	Optional	Specify Directory (tenant) ID if you're building a line-of-business application solely for your organization, often referred to as a <i>single-tenant application</i> .
Redirect URI	Optional	If you're building a web app, the <code>redirectUri</code> specifies where the identity provider (the Microsoft identity platform) should return the security tokens it has issued.

Initialize MSAL.js 2.x apps

Initialize the MSAL.js authentication context by instantiating a [PublicClientApplication](#) with a [Configuration](#) object. The minimum required configuration property is the `clientId` of your application, shown as the **Application (client) ID** on the [Overview](#) page of the app registration in the Azure portal.

Here's an example configuration object and instantiation of a [PublicClientApplication](#):

```

const msalConfig = {
  auth: {
    clientId: "11111111-1111-1111-111111111111",
    authority: "https://login.microsoftonline.com/common",
    knownAuthorities: [],
    redirectUri: "https://localhost:3001",
    postLogoutRedirectUri: "https://localhost:3001/logout",
    navigateToLoginRequestUrl: true,
  },
  cache: {
    cacheLocation: "sessionStorage",
    storeAuthStateInCookie: false,
  },
  system: {
    loggerOptions: {
      loggerCallback: (
        level: LogLevel,
        message: string,
        containsPii: boolean
      ): void => {
        if (containsPii) {
          return;
        }
        switch (level) {
          case LogLevel.Error:
            console.error(message);
            return;
          case LogLevel.Info:
            console.info(message);
            return;
          case LogLevel.Verbose:
            console.debug(message);
            return;
          case LogLevel.Warning:
            console.warn(message);
            return;
        }
      },
      piiLoggingEnabled: false,
    },
    windowHashTimeout: 60000,
    iframeHashTimeout: 6000,
    loadFrameTimeout: 0,
  },
};

// Create an instance of PublicClientApplication
const msalInstance = new PublicClientApplication(msalConfig);

// Handle the redirect flows
msalInstance
  .handleRedirectPromise()
  .then((tokenResponse) => {
    // Handle redirect response
  })
  .catch((error) => {
    // Handle redirect error
  });

```

`handleRedirectPromise`

Invoke `handleRedirectPromise` when your application uses the redirect flows. When using the redirect flows, `handleRedirectPromise` should be run on every page load.

There are three possible outcomes from the promise:

- `.then` is invoked and `tokenResponse` is truthy: The application is returning from a redirect operation that was successful.
- `.then` is invoked and `tokenResponse` is falsy (`null`): The application isn't returning from a redirect operation.
- `.catch` is invoked: The application is returning from a redirect operation and there was an error.

Initialize MSAL.js 1.x apps

Initialize the MSAL 1.x authentication context by instantiating a [UserAgentApplication](#) with a configuration object. The minimum required configuration property is the `clientId` of your application, shown as the **Application (client) ID** on the [Overview](#) page of the app registration in the Azure portal.

For authentication methods with redirect flows ([loginRedirect](#) and [acquireTokenRedirect](#)) in MSAL.js 1.2.x or earlier, you must explicitly register a callback for success or error through the `handleRedirectCallback()` method. Explicitly registering the callback is required in MSAL.js 1.2.x and earlier because redirect flows don't return promises like the methods with a pop-up experience do. Registering the callback is *optional* in MSAL.js version 1.3.x and later.

```
// Configuration object constructed
const msalConfig = {
  auth: {
    clientId: "11111111-1111-1111-111111111111",
  },
};

// Create UserAgentApplication instance
const msalInstance = new UserAgentApplication(msalConfig);

function authCallback(error, response) {
  // Handle redirect response
}

// Register a redirect callback for Success or Error (when using redirect methods)
// **REQUIRED** in MSAL.js 1.2.x and earlier
// **OPTIONAL** in MSAL.js 1.3.x and later
msalInstance.handleRedirectCallback(authCallback);
```

Single instance and configuration

Both MSAL.js 1.x and 2.x are designed to have a single instance and configuration of the [UserAgentApplication](#) or [PublicClientApplication](#), respectively, to represent a single authentication context.

Multiple instances of [UserAgentApplication](#) or [PublicClientApplication](#) aren't recommended as they cause conflicting cache entries and behavior in the browser.

Next steps

The MSAL.js 2.x code sample on GitHub demonstrates instantiation of a [PublicClientApplication](#) with a [Configuration](#) object:

[Azure-Samples/ms-identity-javascript-v2](#)

Known issues on Internet Explorer browsers (MSAL.js)

4/12/2022 • 2 minutes to read • [Edit Online](#)

For better compatibility with Internet Explorer, we generate the Microsoft Authentication Library for JavaScript (MSAL.js) for [JavaScript ES5](#), but there are other things to consider as you develop your application.

Run an app in Internet Explorer

Internet Explorer lacks native support for JavaScript Promises, required by MSAL.js.

To support JavaScript Promises in an Internet Explorer app, reference a Promise polyfill before you reference MSAL.js.

```
<script
  src="https://cdnjs.cloudflare.com/ajax/libs/bluebird/3.3.4/bluebird.min.js"
  class="pre"
></script>
```

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

To debug your application locally, temporarily disable Internet Explorer's *Protected Mode* during your debugging session.

1. In Internet Explorer, select **Tools > Internet Options > Security tab > Internet zone**.
2. Clear the **Enable Protected Mode (requires restarting Internet Explorer)** checkbox.
3. Select **OK** to restart Internet Explorer.

When you're done debugging, follow the previous steps and select (instead of clear) the **Enable Protected Mode (requires restarting Internet Explorer)** checkbox.

Next steps

Learn more about [Known issues when using MSAL.js in Internet Explorer](#).

Use the Microsoft Authentication Library for JavaScript to work with Azure AD B2C

4/12/2022 • 2 minutes to read • [Edit Online](#)

The [Microsoft Authentication Library for JavaScript \(MSAL.js\)](#) enables JavaScript developers to authenticate users with social and local identities using [Azure Active Directory B2C](#) (Azure AD B2C).

By using Azure AD B2C as an identity management service, you can customize and control how your 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 that your application displays during the authentication process.

Supported app types and scenarios

MSAL.js enables [single-page applications](#) to sign-in users with Azure AD B2C using the [authorization code flow with PKCE](#) grant. With MSAL.js and Azure AD B2C:

- Users **can** authenticate with their social and local identities.
- Users **can** be authorized to access Azure AD B2C protected resources (but not Azure AD protected resources).
- Users **cannot** obtain tokens for Microsoft APIs (for example, MS Graph API) using [delegated permissions](#).
- Users with administrator privileges **can** obtain tokens for Microsoft APIs (for example, MS Graph API) using [delegated permissions](#).

For more information, see: [Working with Azure AD B2C](#)

Next steps

Follow the tutorial on how to:

- [Sign in users with Azure AD B2C in a single-page application](#)
- [Call an Azure AD B2C protected web API](#)

Single sign-on with MSAL.js

4/12/2022 • 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. The session provides a seamless experience to the user and reduces the repeated prompts for credentials.

Azure Active Directory (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 apply a session cookie 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 doesn't 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-ashdjhlsdg",
  },
  cache: {
    cacheLocation: "localStorage",
  },
};

const msalInstance = new msal.PublicClientApplication(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 uses 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.

When a user is signed in on domain A navigate to an application on domain B, the user will be redirected or prompted with the sign-in page. Since Azure AD still has the user session cookie, it will sign in the user and no prompt for 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 there's a need to bypass the Azure AD account selection prompt when multiple accounts are signed in. Bypassing the Azure AD account selection prompt can be done in a few different ways:

Using Session ID

Session ID (SID) is an [optional claim](#) that can be configured in the ID tokens. A 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. The `acquireTokenSilent` in the request parameters allow Azure AD to bypass the account selection. SID is bound to the session cookie and won't cross browser contexts.

```
var request = {
  scopes: ["user.read"],
  sid: sid,
};

msalInstance.acquireTokenSilent(request)
  .then(function (response) {
    const token = response.accessToken;
  })
  .catch(function (error) {
    //handle error
  });

```

SID can be used only with silent authentication requests made by `acquireTokenSilent` call in MSAL.js. To find the steps to configure optional claims in your application manifest, see [Provide optional claims to your app](#).

Using Login Hint

If you don't 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" },
};

msalInstance.loginRedirect(request);
```

To 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's consumers. Otherwise, it's organizations.

For more information about `login_hint` and `domain_hint`, see [auth code grant](#).

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 use SSO to acquire tokens without first signing in through MSAL.js.

An example: A user is signed in to Microsoft account in a browser that hosts another JavaScript application running as an add-on or plugin, which requires a Microsoft account sign-in.

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" },
};

msalInstance.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 SSO behavior when updating from ADAL.js, you'll 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-ashdjhlhsdg",
  },
  cache: {
    cacheLocation: "localStorage",
  },
};

const msalInstance = new msal.PublicClientApplication(config);
```

Once the `cacheLocation` is configured, MSAL.js can read the cached state of the authenticated user in ADAL.js and use that to provide SSO in MSAL.js.

Next steps

For more information about SSO, see:

- [Single Sign-On SAML protocol](#)
- [Configurable token lifetimes](#)

Pass custom state in authentication requests using MSAL.js

4/12/2022 • 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, the 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;
    forceRefresh?: boolean;
};
```

NOTE

If you would like to skip a cached token and go to the server, please pass in the boolean `forceRefresh` into the `AuthenticationParameters` object used to make a login/token request. `forceRefresh` should not be used by default, because of the performance impact on your application. Relying on the cache will give your users a better experience. Skipping the cache should only be used in scenarios where you know the currently cached data does not have up-to-date information. Such as an Admin tool that adds roles to a user that needs to get a new token with updated roles.

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

4/12/2022 • 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.

Avoid page reloads when acquiring and renewing tokens silently using MSAL.js

4/12/2022 • 2 minutes to read • [Edit Online](#)

The 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.

How to migrate a JavaScript app from ADAL.js to MSAL.js

4/12/2022 • 11 minutes to read • [Edit Online](#)

Microsoft Authentication Library for JavaScript (MSAL.js, also known as *msal-browser*) 2.x is the authentication library we recommend using with JavaScript applications on the Microsoft identity platform. This article highlights the changes you need to make to migrate an app that uses the ADAL.js to use MSAL.js 2.x

NOTE

We strongly recommend MSAL.js 2.x over MSAL.js 1.x. The auth code grant flow is more secure and allows single-page applications to maintain a good user experience despite the privacy measures browsers like Safari have implemented to block 3rd party cookies, among other benefits.

Prerequisites

- You must set the **Platform / Reply URL Type** to **Single-page application** on App Registration portal (if you have other platforms added in your app registration, such as **Web**, you need to make sure the redirect URLs do not overlap. See: [Redirect URI restrictions](#))
- You must provide [polyfills](#) for ES6 features that MSAL.js relies on (e.g. promises) in order to run your apps on **Internet Explorer**
- Make sure you have migrated your Azure AD apps to [v2 endpoint](#) if you haven't already

Install and import MSAL

There are two ways to install the MSAL.js 2.x library:

Via NPM:

```
npm install @azure/msal-browser
```

Then, depending on your module system, import it as shown below:

```
import * as msal from "@azure/msal-browser"; // ESM

const msal = require('@azure/msal-browser'); // CommonJS
```

Via CDN:

Load the script in the header section of your HTML document:

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="https://alcdn.msauth.net/browser/2.14.2/js/msal-browser.min.js">
  </script>
  </head>
</html>
```

For alternative CDN links and best practices when using CDN, see: [CDN Usage](#)

Initialize MSAL

In ADAL.js, you instantiate the [AuthenticationContext](#) class, which then exposes the methods you can use to achieve authentication (`login`, `acquireTokenPopup` etc.). This object serves as the representation of your application's connection to the authorization server or identity provider. When initializing, the only mandatory parameter is the `clientId`:

```
window.config = {
  clientId: "YOUR_CLIENT_ID"
};

var authContext = new AuthenticationContext(config);
```

In MSAL.js, you instantiate the [PublicClientApplication](#) class instead. Like ADAL.js, the constructor expects a [configuration object](#) that contains the `clientId` parameter at minimum. See for more: [Initialize MSAL.js](#)

```
const msalConfig = {
  auth: {
    clientId: 'YOUR_CLIENT_ID'
  }
};

const msalInstance = new msal.PublicClientApplication(msalConfig);
```

In both ADAL.js and MSAL.js, the authority URI defaults to `https://login.microsoftonline.com/common` if you do not specify it.

NOTE

If you use the `https://login.microsoftonline.com/common` authority in v2.0, you will allow users to sign in with any Azure AD organization or a personal Microsoft account (MSA). In MSAL.js, if you want to restrict login to any Azure AD account (same behavior as with ADAL.js), use `https://login.microsoftonline.com/organizations` instead.

Configure MSAL

Some of the [configuration options in ADAL.js](#) that are used when initializing [AuthenticationContext](#) are deprecated in MSAL.js, while some new ones are introduced. See the [full list of available options](#). Importantly, many of these options, except for `clientId`, can be overridden during token acquisition, allowing you to set them on a *per-request* basis. For instance, you can use a different **authority URI** or **redirect URI** than the one you set during initialization when acquiring tokens.

Additionally, you no longer need to specify the login experience (that is, whether using pop-up windows or redirecting the page) via the configuration options. Instead, `MSAL.js` exposes `loginPopup` and `loginRedirect` methods through the `PublicClientApplication` instance.

Enable logging

In ADAL.js, you configure logging separately at any place in your code:

```

window.config = {
  clientId: "YOUR_CLIENT_ID"
};

var authContext = new AuthenticationContext(config);

var Logging = {
  level: 3,
  log: function (message) {
    console.log(message);
  },
  piiLoggingEnabled: false
};

authContext.log(Logging)

```

In MSAL.js, logging is part of the configuration options and is created during the initialization of `PublicClientApplication`:

```

const msalConfig = {
  auth: {
    // authentication related parameters
  },
  cache: {
    // cache related parameters
  },
  system: {
    loggerOptions: {
      loggerCallback(loglevel, message, containsPii) {
        console.log(message);
      },
      piiLoggingEnabled: false,
      logLevel: msal.LogLevel.Verbose,
    }
  }
}

const msalInstance = new msal.PublicClientApplication(msalConfig);

```

Switch to MSAL API

Some of the public methods in ADAL.js have equivalents in MSAL.js:

ADAL	MSAL	NOTES
<code>acquireToken</code>	<code>acquireTokenSilent</code>	Renamed and now expects an account object
<code>acquireTokenPopup</code>	<code>acquireTokenPopup</code>	Now async and returns a promise
<code>acquireTokenRedirect</code>	<code>acquireTokenRedirect</code>	Now async and returns a promise
<code>handleWindowCallback</code>	<code>handleRedirectPromise</code>	Needed if using redirect experience
<code>getCachedUser</code>	<code>getAllAccounts</code>	Renamed and now returns an array of accounts.

Others were deprecated, while MSAL.js offers new methods:

ADAL	MSAL	NOTES
<code>login</code>	N/A	Deprecated. Use <code>loginPopup</code> or <code>loginRedirect</code>
<code>logOut</code>	N/A	Deprecated. Use <code>logoutPopup</code> or <code>logoutRedirect</code>
N/A	<code>loginPopup</code>	
N/A	<code>loginRedirect</code>	
N/A	<code>logoutPopup</code>	
N/A	<code>logoutRedirect</code>	
N/A	<code>getAccountByHomeId</code>	Filters accounts by home ID (oid + tenant ID)
N/A	<code>getAccountLocalId</code>	Filters accounts by local ID (useful for ADFS)
N/A	<code>getAccountUsername</code>	Filters accounts by username (if exists)

In addition, as MSAL.js is implemented in TypeScript unlike ADAL.js, it exposes various types and interfaces that you can make use of in your projects. See the [MSAL.js API reference](#) for more.

Use scopes instead of resources

An important difference between the Azure AD v1.0 vs. v2.0 endpoints is about how the resources are accessed. When using ADAL.js with the v1.0 endpoint, you would first register a permission on app registration portal, and then request an access token for a resource (such as Microsoft Graph) as shown below:

```
authContext.acquireTokenRedirect("https://graph.microsoft.com", function (error, token) {
  // do something with the access token
});
```

MSAL.js supports both v1.0 and v2.0 endpoints. The v2.0 endpoint employs a *scope-centric* model to access resources. Thus, when you request an access token for a resource, you also need to specify the scope for that resource:

```
msalInstance.acquireTokenRedirect({
  scopes: ["https://graph.microsoft.com/User.Read"]
});
```

One advantage of the scope-centric model is the ability to use *dynamic scopes*. When building applications using the v1.0 endpoint, you needed to register the full set of permissions (called *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 (hence, *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. See for more:

Use promises instead of callbacks

In ADAL.js, callbacks are used for any operation after the authentication succeeds and a response is obtained:

```
authContext.acquireTokenPopup(resource, extraQueryParameter, claims, function (error, token) {
    // do something with the access token
});
```

In MSAL.js, promises are used instead:

```
msalInstance.acquireTokenPopup({
    scopes: ["User.Read"] // shorthand for https://graph.microsoft.com/User.Read
}).then((response) => {
    // do something with the auth response
}).catch((error) => {
    // handle errors
});
```

You can also use the **async/await** syntax that comes with ES8:

```
const getAccessToken = async() => {
    try {
        const authResponse = await msalInstance.acquireTokenPopup({
            scopes: ["User.Read"]
        });
    } catch (error) {
        // handle errors
    }
}
```

Cache and retrieve tokens

Like ADAL.js, MSAL.js caches tokens and other authentication artifacts in browser storage, using the [Web Storage API](#). You are recommended to use `sessionStorage` option (see: [configuration](#)) because it is more secure in storing tokens that are acquired by your users, but `localStorage` will give you [Single Sign On](#) across tabs and user sessions.

Importantly, you are not supposed to access the cache directly. Instead, you should use an appropriate MSAL.js API for retrieving authentication artifacts like access tokens or user accounts.

Renew tokens with refresh tokens

ADAL.js uses the [OAuth 2.0 implicit flow](#), which does not return refresh tokens for security reasons (refresh tokens have longer lifetime than access tokens and are therefore more dangerous in the hands of malicious actors). Hence, ADAL.js performs token renewal using a hidden Iframe so that the user is not repeatedly prompted to authenticate.

With the auth code flow with PKCE support, apps using MSAL.js 2.x obtain refresh tokens along with ID and access tokens, which can be used to renew them. The usage of refresh tokens is abstracted away, and the developers are not supposed to build logic around them. Instead, MSAL manages token renewal using refresh tokens by itself. Your previous token cache with ADAL.js will not be transferable to MSAL.js, as the token cache schema has changed and incompatible with the schema used in ADAL.js.

Handle errors and exceptions

When using MSAL.js, the most common type of error you might face is the `interaction_in_progress` error. This error is thrown when an interactive API (`loginPopup`, `loginRedirect`, `acquireTokenPopup`, `acquireTokenRedirect`) is invoked while another interactive API is still in progress. The `login*` and `acquireToken*` APIs are *async* so you will need to ensure that the resulting promises have resolved before invoking another one.

Another common error is `interaction_required`. This error is often resolved by simply initiating an interactive token acquisition prompt. For instance, the web API you are trying to access might have a [conditional access](#) policy in place, requiring the user to perform [multifactor authentication](#) (MFA). In that case, handling `interaction_required` error by triggering `acquireTokenPopup` or `acquireTokenRedirect` will prompt the user for MFA, allowing them to fulfil it.

Yet another common error you might face is `consent_required`, which occurs when permissions required for obtaining an access token for a protected resource are not consented by the user. As in `interaction_required`, the solution for `consent_required` error is often initiating an interactive token acquisition prompt, using either `acquireTokenPopup` OR `acquireTokenRedirect`.

See for more: [Common MSAL.js errors and how to handle them](#)

Use the Events API

MSAL.js (>=v2.4) introduces an events API that you can make use of in your apps. These events are related to the authentication process and what MSAL is doing at any moment, and can be used to update UI, show error messages, check if any interaction is in progress and so on. For instance, below is an event callback that will be called when login process fails for any reason:

```
const callbackId = msalInstance.addEventCallback((message) => {
    // Update UI or interact with EventMessage here
    if (message.eventType === EventType.LOGIN_FAILURE) {
        if (message.error instanceof AuthError) {
            // Do something with the error
        }
    }
});
```

For performance, it is important to unregister event callbacks when they are no longer needed. See for more: [MSAL.js Events API](#)

Handle multiple accounts

ADAL.js has the concept of a *user* to represent the currently authenticated entity. MSAL.js replaces *users* with *accounts*, given the fact that a user can have more than one account associated with her. This also means that you now need to control for multiple accounts and choose the appropriate one to work with. The snippet below illustrates this process:

```

let homeAccountId = null; // Initialize global accountId (can also be localAccountId or username) used for
account lookup later, ideally stored in app state

// This callback is passed into `acquireTokenPopup` and `acquireTokenRedirect` to handle the interactive
auth response
function handleResponse(resp) {
    if (resp !== null) {
        homeAccountId = resp.account.homeAccountId; // alternatively: resp.account.homeAccountId or
        resp.account.username
    } else {
        const currentAccounts = myMSALObj.getAllAccounts();
        if (currentAccounts.length < 1) { // No cached accounts
            return;
        } else if (currentAccounts.length > 1) { // Multiple account scenario
            // Add account selection logic here
        } else if (currentAccounts.length === 1) {
            homeAccountId = currentAccounts[0].homeAccountId; // Single account scenario
        }
    }
}

```

For more information, see: [Accounts in MSAL.js](#)

Use the wrappers libraries

If you are developing for Angular and React frameworks, you can use [MSAL Angular v2](#) and [MSAL React](#), respectively. These wrappers expose the same public API as MSAL.js while offering framework-specific methods and components that can streamline the authentication and token acquisition processes.

Run the app

Once your changes are done, run the app and test your authentication scenario:

```
npm start
```

Example: Securing web apps with ADAL Node vs. MSAL Node

The snippets below demonstrates the minimal code required for a single-page application authenticating users with the Microsoft identity platform and getting an access token for Microsoft Graph using first ADAL.js and then MSAL.js:

Using ADAL.js	Using MSAL.js
<pre> <head> <meta charset="UTF-8"> <meta http-equiv="X-UA-Compatible" content="IE=edge"> <meta name="viewport" content="width=device- width, initial-scale=1.0"> <script type="text/javascript" src="https://secure.aadcdn.microsoftonline- p.com/lib/1.0.18/js/adal.min.js"> </script> </head> <div> <button id="loginButton">Login</button> </pre>	<pre> <head> <meta charset="UTF-8"> <meta http-equiv="X-UA-Compatible" content="IE=edge"> <meta name="viewport" content="width=device- width, initial-scale=1.0"> <script type="text/javascript" src="https://alcdn.msauth.net/browser/2.14.2/ja /msal-browser.min.js"> </script> </head> <div> </pre>

```

        <button id="logoutButton" style="visibility: hidden;">LogoutGet Token

```

```

        <button id="loginButton">Login</button>
        <button id="logoutButton" style="visibility: hidden;">LogoutGet Token

```

Next steps

- [MSAL.js API reference](#)
- [MSAL.js code samples](#)

Migrate a JavaScript single-page app from implicit grant to auth code flow

4/12/2022 • 2 minutes to read • [Edit Online](#)

The Microsoft Authentication Library for JavaScript (MSAL.js) v2.0 brings support for the authorization code flow with PKCE and CORS to single-page applications on the Microsoft identity platform. Follow the steps in the sections below to migrate your MSAL.js 1.x application using the implicit grant to MSAL.js 2.0+ (hereafter *2.x*) and the auth code flow.

MSAL.js 2.x improves on MSAL.js 1.x by supporting the authorization code flow in the browser instead of the implicit grant flow. MSAL.js 2.x does **NOT** support the implicit flow.

Migration steps

To update your application to MSAL.js 2.x and the auth code flow, there are three primary steps:

1. Switch your [app registration](#) redirect URI(s) from **Web** platform to **Single-page application** platform.
2. Update your [code](#) from MSAL.js 1.x to 2.x.
3. Disable the [implicit grant](#) in your app registration when all applications sharing the registration have been updated to MSAL.js 2.x and the auth code flow.

The following sections describe each step in additional detail.

Switch redirect URIs to SPA platform

If you'd like to continue using your existing app registration for your applications, use the Azure portal to update the registration's redirect URIs to the SPA platform. Doing so enables the authorization code flow with PKCE and CORS support for apps that use the registration (you still need to update your application's code to MSAL.js v2.x).

Follow these steps for app registrations that are currently configured with **Web** platform redirect URIs:

1. Sign in to the [Azure portal](#) and select your **Azure Active Directory** tenant.
2. In **App registrations**, select your application, and then **Authentication**.
3. In the **Web** platform tile under **Redirect URIs**, select the warning banner indicating that you should migrate your URIs.

^ Web

Quickstart Docs 📕

Redirect URIs

The URIs we will accept as destinations when returning authentication responses (tokens) after successfully authenticating users. Also referred as reply URLs. [Learn more about redirect URIs and the restrictions](#)

⚠️ This app has implicit grant settings enabled. If you are using any of these URIs in a SPA with MSAL.js 2.0, you should migrate URIs.

https://jwt.ms

https://localhost:3000

Add URI

4. Select *only* those redirect URIs whose applications will use MSAL.js 2.x, and then select **Configure**.

Migrate URIs

X

i The latest version of MSAL.js uses the authorization code flow with PKCE and CORS. [Learn more](#)

Select URIs to migrate to the single-page application (SPA) platform configuration. Migrated URIs will have auth code flow enabled. Implicit grant is also still enabled so long as the settings are checked.

Redirect URIs ↑↓

<https://jwt.ms>

<https://localhost:3000>

[Configure](#)

[Cancel](#)

These redirect URIs should now appear in the **Single-page application** platform tile, showing that CORS support with the authorization code flow and PKCE is enabled for these URIs.

^ Single-page application

Redirect URIs

The URIs we will accept as destinations when returning authentication responses (tokens) after successfully authenticating users. Also referred as reply URLs. [Learn more about redirect URIs and the restrictions](#)

<https://localhost:3000> 

[Add URI](#)

Grant types

MSAL.js 2.0 does not support implicit grant. Enable implicit grant settings only if your app is using MSAL.js 1.0 [Learn more](#)

 Your Redirect URI is eligible for the Authorization Code Flow with PKCE.

You can also [create a new app registration](#) instead of updating the redirect URIs in your existing registration.

Update your code to MSAL.js 2.x

In MSAL 1.x, you created a application instance by initializing a [UserAgentApplication](#) as follows:

```
// MSAL 1.x
import * as msal from "msal";

const msalInstance = new msal.UserAgentApplication(config);
```

In MSAL 2.x, initialize instead a [PublicClientApplication](#):

```
// MSAL 2.x
import * as msal from "@azure/msal-browser";

const msalInstance = new msal.PublicClientApplication(config);
```

For a full walk-through of adding MSAL 2.x to your application, see [Tutorial: Sign in users and call the Microsoft](#)

[Graph API from a JavaScript single-page app \(SPA\) using auth code flow.](#)

For additional changes you might need to make to your code, see the [migration guide](#) on GitHub.

Disable implicit grant settings

Once you've updated all your production applications that use this app registration and its client ID to MSAL 2.x and the authorization code flow, you should uncheck the implicit grant settings under the **Authentication** menu of the app registration.

When you uncheck the implicit grant settings in the app registration, the implicit flow is disabled for all applications using registration and its client ID.

Do not disable the implicit grant flow before you've updated all your applications to MSAL.js 2.x and the [PublicClientApplication](#).

Next steps

To learn more about the authorization code flow, including the differences between the implicit and auth code flows, see the [Microsoft identity platform and OAuth 2.0 authorization code flow](#).

If you'd like to dive deeper into JavaScript single-page application development on the Microsoft identity platform, the multi-part [Scenario: Single-page application](#) series of articles can help you get started.

Logging in MSAL.js

4/12/2022 • 2 minutes to read • [Edit Online](#)

The 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. Used 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.

The following sections provide more details about MSAL error logging for your application.

Configure logging in MSAL.js

Enable logging in MSAL.js (JavaScript) by passing a loggerOptions object during the configuration for creating a `PublicClientApplication` instance. The only required config parameter is the client ID of the application.

Everything else is optional, but may be required depending on your tenant and application model.

The loggerOptions object has the following properties:

- `loggerCallback`: a Callback function that can be provided by the developer to handle the logging of MSAL statements in a custom manner. Implement the `loggerCallback` function depending on how you want to redirect logs. The loggerCallback function has the following format

```
(level: LogLevel, message: string, containsPii: boolean): void
```

 - 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.

```

const msalConfig = {
  auth: {
    clientId: "enter_client_id_here",
    authority: "https://login.microsoftonline.com/common",
    knownAuthorities: [],
    cloudDiscoveryMetadata: "",
    redirectUri: "enter_redirect_uri_here",
    postLogoutRedirectUri: "enter_postlogout_uri_here",
    navigateToLoginRequestUrl: true,
    clientCapabilities: ["CP1"]
  },
  cache: {
    cacheLocation: "sessionStorage",
    storeAuthStateInCookie: false,
    secureCookies: false
  },
  system: {
    loggerOptions: {
      loggerCallback: (level: LogLevel, message: string, containsPii: boolean): void => {
        if (containsPii) {
          return;
        }
        switch (level) {
          case LogLevel.Error:
            console.error(message);
            return;
          case LogLevel.Info:
            console.info(message);
            return;
          case LogLevel.Verbose:
            console.debug(message);
            return;
          case LogLevel.Warning:
            console.warn(message);
            return;
        }
      },
      piiLoggingEnabled: false
    },
    windowHashTimeout: 60000,
    iframeHashTimeout: 6000,
    loadFrameTimeout: 0,
    asyncPopups: false
  };
}

const msalInstance = new PublicClientApplication(msalConfig);

```

Next steps

For more code samples, refer to [Microsoft identity platform code samples](#).

Handle errors and exceptions in MSAL.js

4/12/2022 • 5 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, 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 [Azure AD Authentication and authorization error codes](#).

During the sign-in experience, you may encounter errors about consents, Conditional Access (MFA, Device Management, Location-based restrictions), token issuance and redemption, and user properties.

The following section provides more details about error handling for your app.

Error handling in MSAL.js

MSAL.js provides error objects that abstract and classify the different types of common errors. It also provides an 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.

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);
});
```

Errors that require interaction

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. This 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.

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, if exists, claims from error message
        if (error.ErrorMessage.claims) {
            accessTokenRequest.claimsRequest = JSON.stringify(error(ErrorMessage.claims));
        }

        // 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.

Retrying after errors and exceptions

You're expected to implement your own retry policies when calling MSAL. MSAL makes HTTP calls to the Azure AD service, and occasionally failures can occur. For example the network can go down or the server is overloaded.

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.

Next steps

Consider enabling [Logging in MSAL.js](#) to help you diagnose and debug issues.

Known issues on Internet Explorer and Microsoft Edge browsers (MSAL.js)

4/12/2022 • 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)
```

- **Pop-up window doesn't close or is stuck when using login through pop-up window to authenticate.** When authenticating through a pop-up 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 pop-up window doesn't close because `MSAL.js` loses the handle to the pop-up window.

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 pop-up 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 pop-up window once or always. If you choose to allow, the browser opens the pop-up 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 pop-up window. The same issue does not happen in Chrome when it prompts you to allow pop-up windows because it does not automatically open a pop-up 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](#).

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

4/12/2022 • 27 minutes to read • [Edit Online](#)

In this quickstart, you download and run a code sample that demonstrates an ASP.NET web application that can sign in users with Azure Active Directory (Azure AD) accounts.

See [How the sample works](#) for an illustration.

Prerequisites

- An Azure account with an active subscription. [Create an account for free.](#)
- [Visual Studio 2019](#)
- [.NET Framework 4.7.2+](#)

Register and download the app

You have two options to start building your application: automatic or manual configuration.

Automatic configuration

If you want to automatically configure your app and then download the code sample, follow these steps:

1. Go to the [Azure portal page for app registration](#).
2. Enter a name for your application and select **Register**.
3. Follow the instructions to download and automatically configure your new application in one click.

Manual configuration

If you want to manually configure your application and code sample, use the following procedures.

Step 1: Register your application

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directory + subscription** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. For **Name**, enter a name for your application. For example, enter **ASPNET-Quickstart**. Users of your app will see this name, and you can change it later.
6. Add **https://localhost:44368/** in **Redirect URI**, and select **Register**.
7. Under **Manage**, select **Authentication**.
8. In the **Implicit grant and hybrid flows** section, select **ID tokens**.
9. Select **Save**.

Step 2: Download the project

[Download the ASP.NET code sample](#)

TIP

To avoid errors caused by path length limitations in Windows, we recommend extracting the archive or cloning the repository into a directory near the root of your drive.

Step 3: Run the project

1. Extract the .zip file to a local folder that's close to the root folder. For example, extract to `C:\Azure-Samples`.

We recommend extracting the archive into a directory near the root of your drive to avoid errors caused by path length limitations on Windows.

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 the project `AppModelv2-WebApp-OpenIDConnect-DotNet` and then select **Restore NuGet packages**.
4. Open the Package Manager Console by selecting **View > Other Windows > Package Manager Console**. Then run `Update-Package Microsoft.CodeDom.Providers.DotNetCompilerPlatform -r`.
5. Edit `Web.config` and replace the parameters `ClientId`, `Tenant`, and `redirectUri` with:

```
<add key="ClientId" value="Enter_the_Application_Id_here" />
<add key="Tenant" value="Enter_the_Tenant_Info_Here" />
<add key="redirectUri" value="https://localhost:44368/" />
```

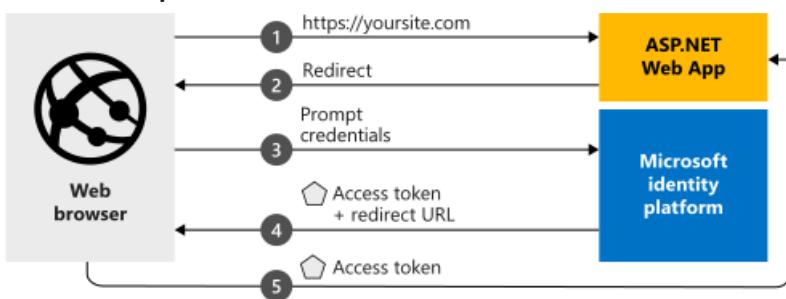
In that code:

- `Enter_the_Application_Id_here` is the application (client) ID of the app registration that you created earlier. Find the application (client) ID on the app's **Overview** page in **App registrations** in the Azure portal.
- `Enter_the_Tenant_Info_Here` is one of the following options:
 - If your application supports **My organization only**, replace this value with the directory (tenant) ID or tenant name (for example, `contoso.onmicrosoft.com`). Find the directory (tenant) ID on the app's **Overview** page in **App registrations** in the Azure portal.
 - 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`.
- `redirectUri` is the **Redirect URI** you entered earlier in **App registrations** 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, what the main arguments are, and how to add sign-in to an existing ASP.NET application.

How the sample works



OWIN middleware NuGet packages

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

```

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 starts. In this quickstart, the `startup.cs` file is in the root folder. The following code shows the parameters that this quickstart uses:

```

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

    app.UseCookieAuthentication(new CookieAuthenticationOptions());
    app.UseOpenIdConnectAuthentication(
        new OpenIdConnectAuthenticationOptions
        {
            // Sets the client ID, authority, and redirect URI 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's using the home page
            PostLogoutRedirectUri = redirectUri,
            Scope = OpenIdConnectScope.OpenIdProfile,
            // ResponseType is set to request the code id_token, which contains basic information about the
            signed-in user
            ResponseType = OpenIdConnectResponseType.CodeIdToken,
            // 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 organization, set ValidateIssuer to true and the '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
            the ValidIssuers parameter
            TokenValidationParameters = new TokenValidationParameters()
            {
                ValidateIssuer = false // Simplification (see note below)
            },
            // OpenIdConnectAuthenticationNotifications configures OWIN to send notification of failed
            authentications to the OnAuthenticationFailed method
            Notifications = new OpenIdConnectAuthenticationNotifications
            {
                AuthenticationFailed = OnAuthenticationFailed
            }
        }
    );
}

```

WHERE	DESCRIPTION
<code>ClientId</code>	The application ID from the application registered in the Azure portal.
<code>Authority</code>	The security token service (STS) endpoint for the user to authenticate. It's usually https://login.microsoftonline.com/{tenant}/v2.0 for the public cloud. In that URL, <code>{tenant}</code> is the name of your tenant, your tenant ID, or <code>common</code> for a reference to the common endpoint. (The common endpoint is used for multitenant applications.)

WHERE	DESCRIPTION
RedirectUri	The URL where users are sent after authentication against the Microsoft identity platform.
PostLogoutRedirectUri	The URL where users are sent after signing off.
Scope	The list of scopes being requested, separated by spaces.
ResponseType	The request that the response from authentication contains an authorization code and an ID token.
TokenValidationParameters	A list of parameters for token validation. In this case, ValidateIssuer is set to false to indicate that it can accept sign-ins from any personal, work, or school account type.
Notifications	A list of delegates that can be run on OpenIdConnect messages.

NOTE

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

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 by using this method is optional. You'd normally use it when you want a view to be accessible from both authenticated and unauthenticated users. Alternatively, you can protect controllers by using the method described in the next section.

Attribute for protecting a controller or a controller actions

You can protect a controller or controller actions by 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. An authentication challenge will then happen automatically when an unauthenticated user tries to access one of the actions or controllers decorated by the `[Authorize]` attribute.

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for](#)

developers.

Next steps

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

[Add sign-in to an ASP.NET web app](#)

In this quickstart, you download and run a code sample that demonstrates how an ASP.NET Core web app can sign in users from any Azure Active Directory (Azure AD) organization.

See [How the sample works](#) for an illustration.

Prerequisites

- [Visual Studio](#) or [Visual Studio Code](#)
- [.NET Core SDK 3.1+](#)

Register and download your quickstart application

Step 1: Register your application

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. For **Name**, enter a name for your application. For example, enter **AspNetCore-Quickstart**. Users of your app will see this name, and you can change it later.
6. For **Redirect URI**, enter <https://localhost:44321/signin-oidc>.
7. Select **Register**.
8. Under **Manage**, select **Authentication**.
9. For **Front-channel logout URL**, enter <https://localhost:44321/signout-oidc>.
10. Under **Implicit grant and hybrid flows**, select **ID tokens**.
11. Select **Save**.
12. Under **Manage**, select **Certificates & secrets > Client secrets > New client secret**.
13. Enter a **Description**, for example `clientsecret1`.
14. Select **In 1 year** for the secret's expiration.
15. Select **Add** and immediately record the secret's **Value** for use in a later step. The secret value is *never displayed again* and is irretrievable by any other means. Record it in a secure location as you would any password.

Step 2: Download the ASP.NET Core project

[Download the ASP.NET Core solution](#)

TIP

To avoid errors caused by path length limitations in Windows, we recommend extracting the archive or cloning the repository into a directory near the root of your drive.

Step 3: Configure your ASP.NET Core project

1. Extract the .zip archive into a local folder near the root of your drive. For example, extract into `C:\Azure-Samples`.

We recommend extracting the archive into a directory near the root of your drive to avoid errors caused by path length limitations on Windows.

2. Open the solution in Visual Studio 2019.
3. Open the *appsettings.json* file and modify the following code:

```
"Domain": "[Enter the domain of your tenant, e.g. contoso.onmicrosoft.com]",  
"ClientId": "Enter_the_Application_Id_here",  
"TenantId": "common",
```

- Replace `Enter_the_Application_Id_here` with the application (client) ID of the application that you registered in the Azure portal. You can find the **Application (client) ID** value on the app's **Overview** page.
- Replace `common` with one of the following:
 - If your application supports **Accounts in this organizational directory only**, replace this value with the directory (tenant) ID (a GUID) or the tenant name (for example, `contoso.onmicrosoft.com`). You can find the **Directory (tenant) ID** value on the app's **Overview** page.
 - If your application supports **Accounts in any organizational directory**, replace this value with `organizations`.
 - If your application supports **All Microsoft account users**, leave this value as `common`.
- Replace `Enter_the_Client_Secret_Here` with the **Client secret** you created and recorded in an earlier step.

For this quickstart, don't change any other values in the *appsettings.json* file.

Step 4: Build and run the application

Build and run the app in Visual Studio by selecting the **Debug** menu > **Start Debugging**, or by pressing the F5 key.

You're prompted for your credentials, and then asked to consent to the permissions that your app requires. Select **Accept** on the consent prompt.



kendall@contoso.onmicrosoft.com

Permissions requested

AspNetCore-Quickstart
[App info](#)

This application is not published by Microsoft.

This app would like to:

- ▽ View your basic profile
- ▽ Maintain access to data you have given it access to

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](#)

Does this app look suspicious? [Report it here](#)

[Cancel](#)

[Accept](#)

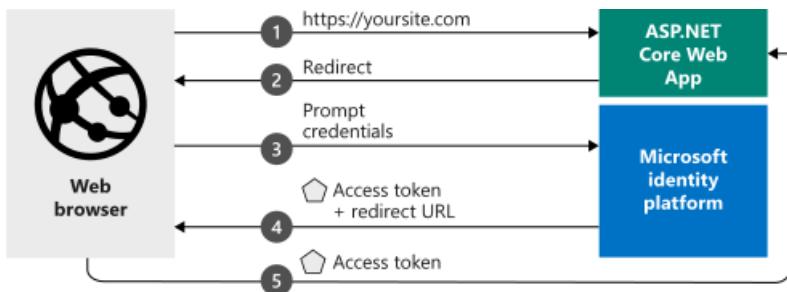
After consenting to the requested permissions, the app displays that you've successfully logged in using your Azure Active Directory credentials, and you'll see your email address in the "Api result" section of the page. This was extracted using Microsoft Graph.

The screenshot shows a web browser window with the title "Home Page - WebApp_OpenIDC". The URL bar shows "https://localhost:44321". The page content includes a "Welcome" message, a link to "Learn about building Web apps with ASP.NET Core.", and a sign-in message "Hello kendall@contoso.onmicrosoft.com! Sign out".

More information

This section gives an overview of the code required to sign in users and call the Microsoft Graph API on their behalf. 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 and call Microsoft Graph. It uses [Microsoft.Identity.Web](#), which is a wrapper around [MSAL.NET](#).

How the sample works



Startup class

The `Microsoft.AspNetCore.Authentication` middleware uses a `Startup` class that's executed when the hosting process starts:

```

// Get the scopes from the configuration (appsettings.json)
var initialScopes = Configuration.GetValue<string>("DownstreamApi:Scopes")?.Split(' ');

public void ConfigureServices(IServiceCollection services)
{
    // Add sign-in with Microsoft
    services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
        .AddMicrosoftIdentityWebApp(Configuration.GetSection("AzureAd"))

    // Add the possibility of acquiring a token to call a protected web API
    .EnableTokenAcquisitionToCallDownstreamApi(initialScopes)

    // Enables controllers and pages to get GraphServiceClient by dependency injection
    // And use an in memory token cache
    .AddMicrosoftGraph(Configuration.GetSection("DownstreamApi"))
    .AddInMemoryTokenCaches();

    services.AddControllersWithViews(options =>
    {
        var policy = new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .Build();
        options.Filters.Add(new AuthorizeFilter(policy));
    });

    // Enables a UI and controller for sign in and sign out.
    services.AddRazorPages()
        .AddMicrosoftIdentityUI();
}

```

The `AddAuthentication()` method configures the service to add cookie-based authentication. This authentication is used in browser scenarios and to set the challenge to OpenID Connect.

The line that contains `.AddMicrosoftIdentityWebApp` adds Microsoft identity platform authentication to your application. The application is then configured to sign in users based on the following information in the `AzureAD` section of the `appsettings.json` configuration file:

APPSETTINGS.JSON KEY	DESCRIPTION
<code>ClientId</code>	Application (client) ID of the application registered in the Azure portal.
<code>Instance</code>	Security token service (STS) endpoint for the user to authenticate. This value is typically <code>https://login.microsoftonline.com/</code> , indicating the Azure public cloud.
<code>TenantId</code>	Name of your tenant or the tenant ID (a GUID), or <code>common</code> to sign in users with work or school accounts or Microsoft personal accounts.

The `EnableTokenAcquisitionToCallDownstreamApi` method enables your application to acquire a token to call protected web APIs. `AddMicrosoftGraph` enables your controllers or Razor pages to benefit directly the `GraphServiceClient` (by dependency injection) and the `AddInMemoryTokenCaches` methods enables your app to benefit from a token cache.

The `Configure()` method contains two important methods, `app.UseAuthentication()` and `app.UseAuthorization()`, that enable their named functionality. Also in the `Configure()` method, you must register Microsoft Identity Web routes with at least one call to `endpoints.MapControllerRoute()` or a call to `endpoints.MapControllers()`:

```
app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
    endpoints.MapRazorPages();
});
```

Protect a controller or a controller's method

You can protect a controller or its methods by applying the `[Authorize]` attribute to the controller's class or one or more of its methods. This `[Authorize]` attribute restricts access by allowing only authenticated users. If the user isn't already authenticated, an authentication challenge can be started to access the controller. In this quickstart, the scopes are read from the configuration file:

```
[AuthorizeForScopes(ScopeKeySection = "DownstreamApi:Scopes")]
public async Task<IActionResult> Index()
{
    var user = await _graphServiceClient.Me.Request().GetAsync();
    ViewData["ApiResult"] = user.DisplayName;

    return View();
}
```

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

The GitHub repo that contains the ASP.NET Core code sample referenced in this quickstart includes instructions and more code samples that show you how to:

- Add authentication to a new ASP.NET Core web application.
- Call Microsoft Graph, other Microsoft APIs, or your own web APIs.
- Add authorization.
- Sign in users in national clouds or with social identities.

[ASP.NET Core web app tutorials on GitHub](#)

In this quickstart, you download and run a code sample that demonstrates how a Node.js web app can sign in users by using the authorization code flow. The code sample also demonstrates how to get an access token to call Microsoft Graph API.

See [How the sample works](#) for an illustration.

This quickstart uses the Microsoft Authentication Library for Node.js (MSAL Node) with the authorization code flow.

Prerequisites

- An Azure subscription. [Create an Azure subscription for free](#).
- [Node.js](#)

- [Visual Studio Code](#) or another code editor

Register and download your quickstart application

Step 1: Register your application

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Under **Manage**, select **App registrations > New registration**.
4. Enter a **Name** for your application. Users of your app might see this name, and you can change it later.
5. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
6. Set the **Redirect URI** value to `http://localhost:3000/redirect`.
7. Select **Register**.
8. On the app **Overview** page, note the **Application (client) ID** value for later use.
9. Under **Manage**, select **Certificates & secrets > Client secrets > New client secret**. Leave the description blank and default expiration, and then select **Add**.
10. Note the value of **Client secret** for later use.

Step 2: Download the project

To run the project with a web server by using Node.js, [download the core project files](#).

Step 3: Configure your Node app

Extract the project, open the *ms-identity-node-main* folder, and then open the *index.js* file.

Set the `clientId` value with the application (client) ID, and then set the `clientSecret` value with the client secret.

```
const config = {
  auth: {
    clientId: "Enter_the_Application_Id_Here",
    authority: "https://login.microsoftonline.com/common",
    clientSecret: "Enter_the_Client_Secret_Here"
  },
  system: {
    loggerOptions: {
      loggerCallback(loglevel, message, containsPii) {
        console.log(message);
      },
      piiLoggingEnabled: false,
      logLevel: msal.LogLevel.Verbose,
    }
  }
};
```

Modify the values in the `config` section:

- `Enter_the_Application_Id_Here` is the application (client) ID for the application you registered.

To find the application (client) ID, go to the app registration's **Overview** page in the Azure portal.

- `Enter_the_Client_Secret_Here` is the client secret for the application you registered.

To retrieve or generate a new client secret, under **Manage**, select **Certificates & secrets**.

The default `authority` value represents the main (global) Azure cloud:

```
authority: "https://login.microsoftonline.com/common",
```

Step 4: Run the project

Run the project by using Node.js.

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

```
npm install  
npm start
```

2. Go to <http://localhost:3000/>.

3. Select **Sign In** to start the sign-in process.

The first time you sign in, you're prompted to provide your consent to allow the application to access your profile and sign you in. After you're signed in successfully, you will see a log message in the command line.

More information

How the sample works

The sample hosts a web server on localhost, port 3000. When a web browser accesses this site, the sample immediately redirects the user to a Microsoft authentication page. Because of this, the sample does not contain any HTML or display elements. Authentication success displays the message "OK".

MSAL Node

The MSAL Node library signs in users and requests the tokens that are used to access an API that's protected by Microsoft identity platform. You can download the latest version by using the Node.js Package Manager (npm):

```
npm install @azure/msal-node
```

Next steps

[Adding Auth to an existing web app - GitHub code sample >](#)

In this quickstart, you download and run a code sample that demonstrates 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

- An Azure account with an active subscription. [Create an account for free.](#)
- [Node.js](#).

Register your application

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.

5. Enter a **Name** for your application, for example `MyWebApp`. Users of your app might see this name, and you can change it later.
6. 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, add these from the **Authentication** tab later after the app has been successfully created.
7. Select **Register** to create the app.
8. 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.
9. Under **Manage**, select **Authentication**.
10. Select **Add a platform > Web**.
11. In the **Redirect URIs** section, enter `http://localhost:3000/auth/openid/return`.
12. Enter a **Front-channel logout URL** `https://localhost:3000`.
13. In the **Implicit grant and hybrid flows** section, select **ID tokens** as this sample requires the [Implicit grant flow](#) to be enabled to sign-in the user.
14. Select **Configure**.
15. Under **Manage**, select **Certificates & secrets > Client secrets > New client secret**.
16. Enter a key description (for instance app secret).
17. Select a key duration of either **In 1 year**, **In 2 years**, or **Never Expires**.
18. Select **Add**. The key value will be displayed. Copy the key value and save it in a safe location for later use.

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 isn't 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're using mongoDB session store in this app, you have to [install mongoDB](#) and start the service first. If you're 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'll 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 about your support options, see [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](#)

In this quickstart, you download and run a code sample that demonstrates how a Java web application can sign in users and call the Microsoft Graph API. Users from any Azure Active Directory (Azure AD) organization can sign in to the application.

See [How the sample works](#) for an illustration.

Prerequisites

To run this sample, you need:

- [Java Development Kit \(JDK\)](#) 8 or later.
- [Maven](#).

Register and download your quickstart app

There are two ways to start your quickstart application: express (option 1) and manual (option 2).

Option 1: Register and automatically configure your app, and then download the code sample

1. Go to the [Azure portal - App registrations](#) quickstart experience.
2. Enter a name for your application, and then select **Register**.
3. Follow the instructions in the portal's quickstart experience to download the automatically configured application code.

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 it, follow these steps:

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations**.
5. Select **New registration**.
6. Enter a **Name** for your application, for example **java-webapp**. Users of your app might see this name. You can change it later.
7. Select **Register**.
8. On the **Overview** page, note the **Application (client) ID** and the **Directory (tenant) ID**. You'll need these values later.
9. Under **Manage**, select **Authentication**.
10. Select **Add a platform > Web**.
11. In the **Redirect URIs** section, enter `https://localhost:8443/msa14jsample/secure/aad`.
12. Select **Configure**.
13. In the **Web** section, under **Redirect URIs**, enter `https://localhost:8443/msa14jsample/graph/me` as a second redirect URI.
14. Under **Manage**, select **Certificates & secrets**. In the **Client secrets** section, select **New client secret**.
15. Enter a key description (for example, *app secret*), leave the default expiration, and select **Add**.
16. Note the **Value** of the client secret. You'll need it later.

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. *Optional.* If you use an integrated development environment, open the sample in that environment.
3. Open the *application.properties* file. You can find it in the *src/main/resources*/folder. Replace the values in the fields `aad.clientId`, `aad.authority`, and `aad.secretKey` with the application ID, tenant ID, and client secret values, respectively. Here's what it should look like:

```
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=https://localhost:8443/msal4jsample/secure/aad  
aad.redirectUriGraph=https://localhost:8443/msal4jsample/graph/me  
aad.msGraphEndpointHost="https://graph.microsoft.com/"
```

In the previous code:

- `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.
4. To use HTTPS with localhost, provide the `server.ssl.key` properties. To generate a self-signed certificate, use the keytool utility (included in JRE).

Here's an example:

```
keytool -genkeypair -alias testCert -keyalg RSA -storetype PKCS12 -keystore keystore.p12 -storepass password  
  
server.ssl.key-store-type=PKCS12  
server.ssl.key-store=classpath:keystore.p12  
server.ssl.key-store-password=password  
server.ssl.key-alias=testCert
```

5. Put the generated keystore file in the *resources* folder.

Step 4: Run the code sample

To run the project, take one of these steps:

- Run it directly from your IDE by using the embedded Spring Boot server.
- Package it to a WAR file by using [Maven](#), and then deploy it to a J2EE container solution like [Apache Tomcat](#).

Running the project from an IDE

To run the web application from an IDE, select run, and then go to the home page of the project. For this sample, the standard home page URL is <https://localhost:8443>.

1. On the front page, select the **Login** button to redirect users to Azure Active Directory and prompt them for credentials.
2. After users are authenticated, they're redirected to <https://localhost:8443/msal4jsample/secure/aad>. They're now signed in, and the page will show information about the user account. The sample UI has these buttons:
 - **Sign Out**: Signs the current user out of the application and redirects that user to the home page.
 - **Show User Info**: Acquires a token for Microsoft Graph and calls Microsoft Graph with a request that contains the token, which returns basic information about the signed-in user.

Running the project from Tomcat

If you want to deploy the web sample to Tomcat, make a couple changes to the source code.

1. Open *ms-identity-java-webapp/src/main/java/com.microsoft.azure.msalwebsample/MsalWebSampleApplication*.
 - Delete all source code and replace it with this code:

```

package com.microsoft.azure.msalwebsample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication
public class MsalWebSampleApplication extends SpringBootServletInitializer {

    public static void main(String[] args) {
        SpringApplication.run(MsalWebSampleApplication.class, args);
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(MsalWebSampleApplication.class);
    }
}

```

2. Tomcat's default HTTP port is 8080, but you need an HTTPS connection over port 8443. To configure this setting:

- Go to `tomcat/conf/server.xml`.
- Search for the `<connector>` tag, and replace the existing connector with this connector:

```

<Connector
    protocol="org.apache.coyote.http11.Http11NioProtocol"
    port="8443" maxThreads="200"
    scheme="https" secure="true" SSLEnabled="true"
    keystoreFile="C:/Path/To/Keystore/File/keystore.p12" keystorePass="KeystorePassword"
    clientAuth="false" sslProtocol="TLS"/>

```

3. Open a Command Prompt window. Go to the root folder of this sample (where the `pom.xml` file is located), and run `mvn package` to build the project.

- This command will generate a `msal-web-sample-0.1.0.war` file in your `/targets` directory.
- Rename this file to `msal4jsample.war`.
- Deploy the WAR file by using Tomcat or any other J2EE container solution.
 - To deploy the `msal4jsample.war` file, copy it to the `/webapps/` directory in your Tomcat installation, and then start the Tomcat server.

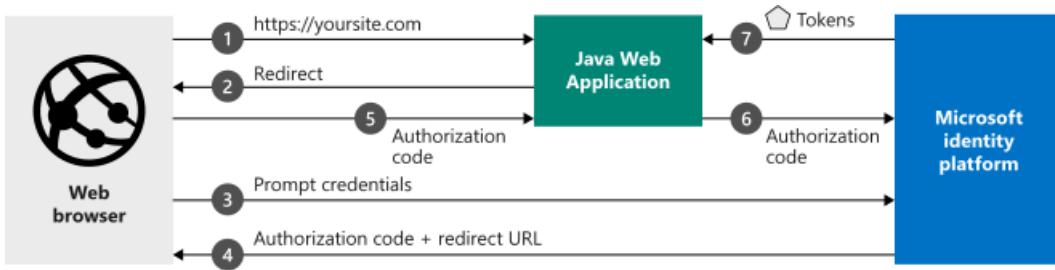
4. After the file is deployed, go to `https://localhost:8443/msal4jsample` by using a browser.

IMPORTANT

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

More information

How the sample works



Get MSAL

MSAL for Java (MSAL4J) is the Java library used to sign in users and request tokens that are used to access an API that's 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.

In pom.xml:

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

In build.gradle:

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

Initialize MSAL

Add a reference to MSAL for Java by adding the following code at the start of the file where you'll be using MSAL4J:

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

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

For a more in-depth discussion of building web apps that sign in users on the Microsoft identity platform, see the multipart scenario series:

Scenario: Web app that signs in users

In this quickstart, you download and run a code sample that demonstrates how a Python web application can sign in users and get an access token to call the Microsoft Graph API. Users with a personal Microsoft Account or an account in any Azure Active Directory (Azure AD) organization can sign into the application.

See [How the sample works](#) for an illustration.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).

- 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

1. Go to the [Azure portal - App registrations](#) quickstart experience.
2. Enter a name for your application and select **Register**.
3. 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:

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directory + subscription** filter  in the top menu to select the tenant in which you want to register an application.
3. Under **Manage**, select **App registrations > New registration**.
4. Enter a **Name** for your application, for example `python-webapp`. Users of your app might see this name, and you can change it later.
5. Under **Supported account types**, select **Accounts in any organizational directory and personal Microsoft accounts**.
6. Select **Register**.
7. On the app **Overview** page, note the **Application (client) ID** value for later use.
8. Under **Manage**, select **Authentication**.
9. Select **Add a platform > Web**.
10. Add `http://localhost:5000/getAToken` as **Redirect URIs**.
11. Select **Configure**.
12. Under **Manage**, select the **Certificates & secrets** and from the **Client secrets** section, select **New client secret**.
13. Type a key description (for instance app secret), leave the default expiration, and select **Add**.
14. Note the **Value** of the **Client Secret** for later use.
15. Under **Manage**, select **API permissions > Add a permission**.
16. Ensure that the **Microsoft APIs** tab is selected.
17. From the *Commonly used Microsoft APIs* section, select **Microsoft Graph**.
18. From the **Delegated permissions** section, ensure that the right permissions are checked:
`User.ReadBasic.All`. Use the search box if necessary.
19. Select the **Add permissions** button.

Step 2: Download your project

[Download the Code Sample](#)

Step 3: Configure the Application

1. Extract the zip file to a local folder closer to the root folder - for example, C:\Azure-Samples
2. If you use an integrated development environment, open the sample in your favorite IDE (optional).
3. 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

1. 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
```

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

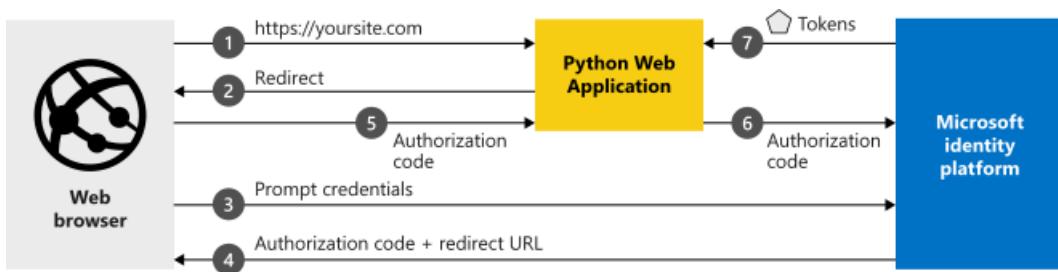
```
python app.py
```

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

How the sample works



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
```

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

Learn more about web apps that sign in users in our multi-part scenario series.

[Scenario: Web app that signs in users](#)

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

4/12/2022 • 14 minutes to read • [Edit Online](#)

In this tutorial, you build an ASP.NET MVC web app that signs in users by using the Open Web Interface for .NET (OWIN) middleware and the Microsoft identity platform.

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 the Microsoft identity platform will be able to sign in to your app.

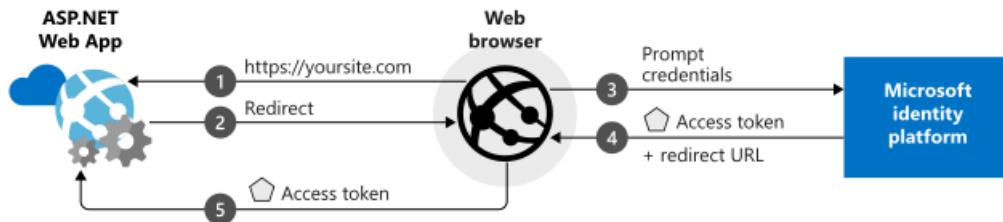
In this tutorial:

- Create an *ASP.NET Web Application* project in Visual Studio
- Add the Open Web Interface for .NET (OWIN) middleware components
- Add code to support user sign-in and sign-out
- Register the app in the Azure portal
- Test the app

Prerequisites

- [Visual Studio 2019](#) with the **ASP.NET and web development** workload installed

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
Microsoft.Owin.Security.OpenIdConnect	Middleware that enables an application to use OpenIdConnect for authentication
Microsoft.Owin.Security.Cookies	Middleware that enables an application to maintain a user session by using cookies

LIBRARY	DESCRIPTION
Microsoft.Owin.Host.SystemWeb	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 Microsoft identity
    // platform.
    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 of the Microsoft identity platform 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 code id_token - which contains basic
                // information about the signed-in user
                ResponseType = OpenIdConnectResponseType.CodeIdToken,
                // 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 Microsoft identity platform. 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 --&gt;
    &lt;a href="@Url.Action("SignIn", "Home)" style="text-decoration: none;"&gt;
        &lt;svg xmlns="http://www.w3.org/2000/svg" xml:space="preserve" width="300px" height="50px"
viewBox="0 0 3278 522" class="SignInButton"&gt;
            &lt;style type="text/css"&gt;.fil0:hover {fill: #4B4B4B;} .fnt0 {font-size: 260px;font-family:
'Segoe UI Semibold', 'Segoe UI'; text-decoration: none;}&lt;/style&gt;
            &lt;rect class="fil0" x="2" y="2" width="3174" height="517" fill="black" /&gt;
            &lt;rect x="150" y="129" width="122" height="122" fill="#F35325" /&gt;
            &lt;rect x="284" y="129" width="122" height="122" fill="#81BC06" /&gt;
            &lt;rect x="150" y="263" width="122" height="122" fill="#05A6F0" /&gt;
            &lt;rect x="284" y="263" width="122" height="122" fill="#FFBA08" /&gt;
            &lt;text x="470" y="357" fill="white" class="fnt0"&gt;Sign in with Microsoft&lt;/text&gt;
        &lt;/svg&gt;
    &lt;/a&gt;
}
else
{
    &lt;span&gt;&lt;br/&gt;Hello @System.Security.Claims.ClaimsPrincipal.Current.FindFirst("name").Value;&lt;/span&gt;
    &lt;br /&gt;&lt;br /&gt;
    @Html.ActionLink("See Your Claims", "Index", "Claims")
    &lt;br /&gt;&lt;br /&gt;
    @Html.ActionLink("Sign out", "SignOut", "Home")
}
@if (!string.IsNullOrWhiteSpace(Request.QueryString["errormessage"]))
{
    &lt;div style="background-color:red;color:white;font-weight: bold;"&gt;Error:
    @Request.QueryString["errormessage"]&lt;/div&gt;
}
&lt;/body&gt;
&lt;/html&gt;
</pre>

```

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>
    <br />
    <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>
    <br />
    <br />
    @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

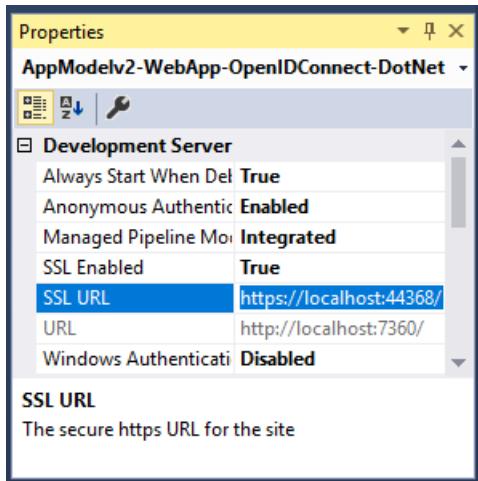
To quickly register your application, follow these steps:

1. Go to the [Azure portal - App registrations](#) quickstart experience.
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 URIs in the Registration portal's list of Redirect URIs in the next step.



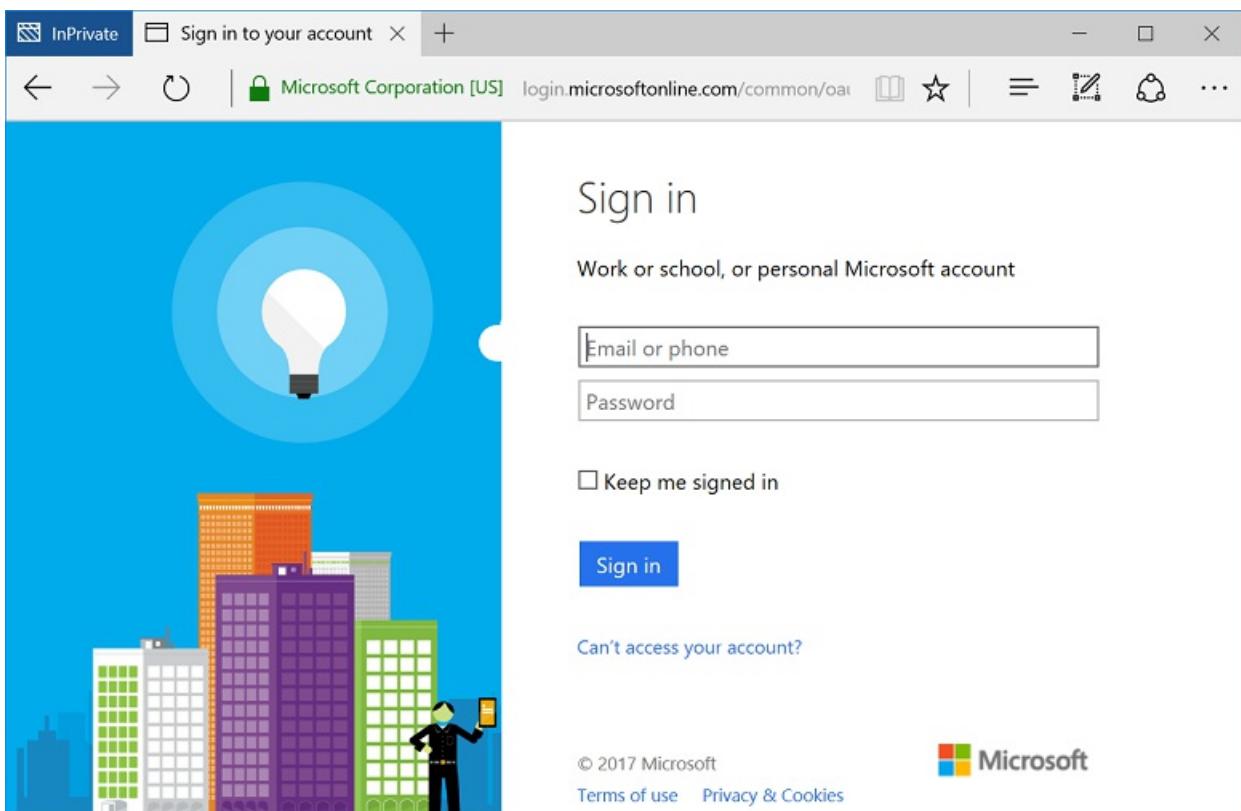
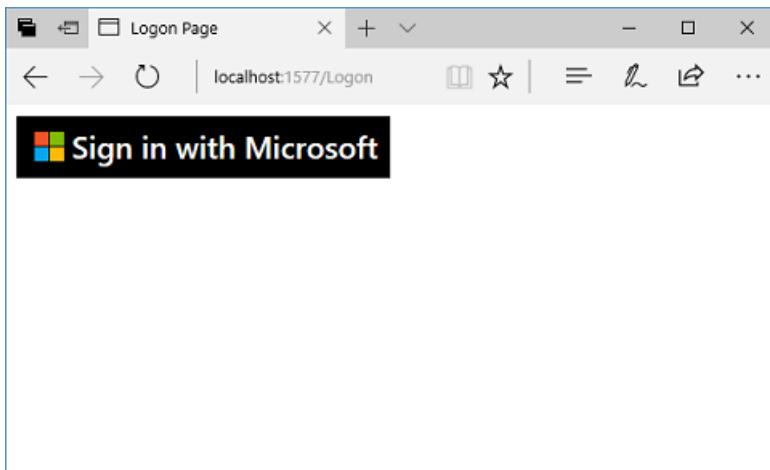
2. Sign in to the [Azure portal](#).
3. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
4. Search for and select **Azure Active Directory**.
5. Under **Manage**, select **App registrations > New registration**.
6. Enter a **Name** for your application, for example `ASPNET-Tutorial1`. Users of your app might see this name, and you can change it later.
7. Add the SSL URL you copied from Visual Studio in step 1 (for example, `https://localhost:44368/`) in **Redirect URI**.
8. Select **Register**.
9. Under **Manage**, select **Authentication**.
10. In the **Implicit grant and hybrid flows** section, select **ID tokens**, and then select **Save**.
11. 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" />
```
12. Replace `ClientId` with the Application ID you just registered.
13. 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

Applications that integrate with the Microsoft identity platform follow an authorization model that gives users and administrators control over how data can be accessed. After a user authenticates with the Microsoft identity platform 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](#).

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 guid 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 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 Microsoft identity platform. 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.

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

Learn about calling protected web APIs from web apps with the Microsoft identity platform:

[Web apps calling web APIs](#)

Tutorial: Create a Blazor Server app that uses the Microsoft identity platform for authentication

4/12/2022 • 5 minutes to read • [Edit Online](#)

In this tutorial, you build a Blazor Server app that signs in users and gets data from Microsoft Graph by using the Microsoft identity platform and registering your app in Azure Active Directory (Azure AD).

We also have a tutorial for [Blazor WASM](#).

In this tutorial:

- Create a new Blazor Server app configured to use Azure Active Directory (Azure AD) for authentication
- Handle both authentication and authorization using Microsoft.Identity.Web
- Retrieve data from a protected web API, Microsoft Graph

Prerequisites

- [.NET Core 3.1 SDK](#)
- An Azure AD tenant where you can register an app. If you don't have access to an Azure AD tenant, you can get one by registering with the [Microsoft 365 Developer Program](#) or by creating an [Azure free account](#).

Register the app in the Azure portal

Every app that uses Azure Active Directory (Azure AD) for authentication must be registered with Azure AD.

Follow the instructions in [Register an application](#) with these additions:

- For **Supported account types**, select **Accounts in this organizational directory only**.
- Leave the **Redirect URI** drop down set to **Web** and enter `https://localhost:5001/signin-oidc`. The default port for an app running on Kestrel is 5001. If the app is available on a different port, specify that port number instead of `5001`.

Under **Manage**, select **Authentication > Implicit grant and hybrid flows**. Select **ID tokens**, and then select **Save**.

Finally, because the app calls a protected API (in this case Microsoft Graph), it needs a client secret in order to verify its identity when it requests an access token to call that API.

1. Within the same app registration, under **Manage**, select **Certificates & secrets** and then **Client secrets**.
2. Create a **New client secret** that never expires.
3. Make note of the secret's **Value** as you will use it in the next step. You can't access it again once you navigate away from this pane. However, you can recreate it as needed.

Create the app using the .NET CLI

Run the following command to download the templates for Microsoft.Identity.Web, which we will make use of in this tutorial.

```
dotnet new --install Microsoft.Identity.Web.ProjectTemplates
```

Then, run the following command to create the application. Replace the placeholders in the command with the

proper information from your app's overview page and execute the command in a command shell. The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the app's name.

```
dotnet new blazorserver2 --auth SingleOrg --calls-graph -o {APP NAME} --client-id "{CLIENT ID}" --tenant-id "{TENANT ID}" --domain "{DOMAIN}"
```

PLACEHOLDER	AZURE PORTAL NAME	EXAMPLE
{APP NAME}	—	BlazorSample
{CLIENT ID}	Application (client) ID	41451fa7-0000-0000-0000-69eff5a761fd
{TENANT ID}	Directory (tenant) ID	e86c78e2-0000-0000-0000-918e0565a45e
{DOMAIN}	Primary domain	tenantname.onmicrosoft.com

Now, navigate to your new Blazor app in your editor and add the client secret to the `appsettings.json` file, replacing the text "secret-from-app-registration".

```
"ClientSecret": "secret-from-app-registration",
```

Test the app

You can now build and run the app. When you run this template app, you must specify the framework to run using `--framework`. This tutorial uses the .NET Core 3.1 SDK.

```
dotnet run --framework netcoreapp3.1
```

In your browser, navigate to `https://localhost:5001`, and log in using an Azure AD user account to see the app running.

Retrieving data from Microsoft Graph

[Microsoft Graph](#) offers a range of APIs that provide access to your users' Microsoft 365 data. By using the Microsoft identity platform as the identity provider for your app, you have easier access to this information since Microsoft Graph directly supports the tokens issued by the Microsoft identity platform. In this section, you add code to display the signed in user's emails on the application's "fetch data" page.

Before you start, log out of your app since you'll be making changes to the required permissions, and your current token won't work. If you haven't already, run your app again and select **Log out** before updating the code below.

Now you will update your app's registration and code to pull a user's email and display the messages within the app. To achieve this, first extend the app registration permissions in Azure AD to enable access to the email data. Then, add code to the Blazor app to retrieve and display this data in one of the pages.

1. In the Azure portal, select your app in **App registrations**.
2. Under **Manage**, select **API permissions**.
3. Select **Add a permission > Microsoft Graph**.

4. Select **Delegated Permissions**, then search for and select the **Mail.Read** permission.

5. Select **Add permissions**.

In the *appsettings.json* file, update your code so it fetches the appropriate token with the right permissions. Add "mail.read" after the "user.read" scope under "DownstreamAPI". This is specifying which scopes (or permissions) the app will request access to.

```
"Scopes": "user.read mail.read"
```

Next, update the code in the *FetchData.razor* file to retrieve email data instead of the default (random) weather details. Replace the code in that file with the following:

```
@page "/fetchdata"

@inject IHttpClientFactory HttpClientFactory
@inject Microsoft.Identity.Web.ITokenAcquisition TokenAcquisitionService

<p>This component demonstrates fetching data from a service.</p>

@if (messages == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <h1>Hello @userDisplayName !!!!</h1>
    <table class="table">
        <thead>
            <tr>
                <th>Subject</th>
                <th>Sender</th>
                <th>Received Time</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var mail in messages)
            {
                <tr>
                    <td>@mail.Subject</td>
                    <td>@mail.Sender</td>
                    <td>@mail.ReceivedTime</td>
                </tr>
            }
        </tbody>
    </table>
}
```

@code {

```
private string userDisplayName;
private List<MailMessage> messages = new List<MailMessage>();

private HttpClient _httpClient;

protected override async Task OnInitializedAsync()
{
    _httpClient = HttpClientFactory.CreateClient();

    // get a token
    var token = await TokenAcquisitionService.GetAccessTokenForUserAsync(new string[] { "User.Read",
    "Mail.Read" });

    // make API call
    HttpResponseMessage response = await _httpClient.GetAsync("https://graph.microsoft.com/v1.0/me/messages");
    if (response.IsSuccessStatusCode)
    {
        var content = await response.Content.ReadAsStringAsync();
        messages = JsonConvert.DeserializeObject<List<MailMessage>>(content);
    }
}
```

```

        _httpClient.DefaultRequestHeaders.Authorization = new
System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
        var dataRequest = await _httpClient.GetAsync("https://graph.microsoft.com/beta/me");

        if (dataRequest.IsSuccessStatusCode)
        {
            var userData = System.Text.Json.JsonDocument.Parse(await
dataRequest.Content.ReadAsStreamAsync());
            userDisplayName = userData.RootElement.GetProperty("displayName").GetString();
        }

        var mailRequest = await _httpClient.GetAsync("https://graph.microsoft.com/beta/me/messages?
$select=subject,receivedDateTime,sender&$top=10");

        if (mailRequest.IsSuccessStatusCode)
        {
            var mailData = System.Text.Json.JsonDocument.Parse(await
mailRequest.Content.ReadAsStreamAsync());
            var messagesArray = mailData.RootElement.GetProperty("value").EnumerateArray();

            foreach (var m in messagesArray)
            {
                var message = new MailMessage();
                message.Subject = m.GetProperty("subject").GetString();
                message.Sender =
m.GetProperty("sender").GetProperty("emailAddress").GetProperty("address").GetString();
                message.ReceivedTime = m.GetProperty("receivedDateTime").GetDateTime();
                messages.Add(message);
            }
        }
    }

    public class MailMessage
    {
        public string Subject;
        public string Sender;
        public DateTime ReceivedTime;
    }
}

```

Launch the app. You'll notice that you're prompted for the newly added permissions, indicating that everything is working as expected. Now, beyond basic user profile data, the app is requesting access to email data.

After granting consent, navigate to the "Fetch data" page to read some email.

[Home](#) [Counter](#) [Fetch data](#) [Show profile](#)

This component demonstrates fetching data from a service.

Hello Nicholas

!!!!

Subject	Sender	Received Time
Azure AD Identity Protection Weekly Digest		9/22/2020 11:17:40 AM
Weekly digest: Microsoft service updates		9/21/2020 9:04:33 AM
Your password has been reset		9/15/2020 1:56:46 AM

Next steps

Learn about calling building web apps that sign in users in our multi-part scenario series:

[Scenario: Web app that signs in users](#)

Tutorial: Sign in users in a Node.js & Express web app

4/12/2022 • 3 minutes to read • [Edit Online](#)

In this tutorial, you build a web app that signs-in users. The web app you build uses the [Microsoft Authentication Library \(MSAL\) for Node](#).

Follow the steps in this tutorial to:

- Register the application in the Azure portal
- Create an Express web app project
- Install the authentication library packages
- Add app registration details
- Add code for user login
- Test the app

Prerequisites

- [Node.js](#)
- [Visual Studio Code](#) or another code editor

Register the application

First, complete the steps in [Register an application with the Microsoft identity platform](#) to register your app.

Use the following settings for your app registration:

- Name: `ExpressWebApp` (suggested)
- Supported account types: **Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**
- Platform type: **Web**
- Redirect URI: `http://localhost:3000/redirect`
- Client secret: `*****` (record this value for use in a later step - it's shown only once)

Create the project

Create a folder to host your application, for example `ExpressWebApp`.

1. First, change to your project directory in your terminal and then run the following `npm` commands:

```
npm init -y
npm install --save express
```

2. Next, create file named `index.js` and add the following code:

```
const express = require("express");
const msal = require('@azure/msal-node');

const SERVER_PORT = process.env.PORT || 3000;

// Create Express App and Routes
const app = express();

app.listen(SERVER_PORT, () => console.log(`Msal Node Auth Code Sample app listening on port ${SERVER_PORT}!`))
```

You now have a simple web server running on port 3000. The file and folder structure of your project should look similar to the following:

```
ExpressWebApp/
├── index.js
└── package.json
```

Install the auth library

Locate the root of your project directory in a terminal and install the MSAL Node package via NPM.

```
npm install --save @azure/msal-node
```

Add app registration details

In the *index.js* file you've created earlier, add the following code:

```
// Before running the sample, you will need to replace the values in the config,
// including the clientSecret
const config = {
    auth: {
        clientId: "Enter_the_Application_Id",
        authority: "Enter_the_Cloud_Instance_Id_Here/Enter_the_Tenant_Id_here",
        clientSecret: "Enter_the_Client_secret"
    },
    system: {
        loggerOptions: {
            loggerCallback(loglevel, message, containsPii) {
                console.log(message);
            },
            piiLoggingEnabled: false,
            logLevel: msal.LogLevel.Verbose,
        }
    }
};
```

Fill in these details with the values you obtain from Azure app registration portal:

- `Enter_the_Tenant_Id_here` should be one of the following:
 - 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`.
- `Enter_the_Application_Id_Here` : The Application (client) ID of the application you registered.
- `Enter_the_Cloud_Instance_Id_Here` : The Azure cloud instance in which your application is registered.
 - For the main (or *global*) Azure cloud, enter `https://login.microsoftonline.com`.
 - For **national** clouds (for example, China), you can find appropriate values in [National clouds](#).
- `Enter_the_Client_secret` : Replace this value with the client secret you created earlier. To generate a new key, use [Certificates & secrets](#) in the app registration settings in the Azure portal.

WARNING

Any plaintext secret in source code poses an increased security risk. This article uses a plaintext client secret for simplicity only. Use [certificate credentials](#) instead of client secrets in your confidential client applications, especially those apps you intend to deploy to production.

Add code for user login

In the `index.js` file you've created earlier, add the following code:

```
// Create msal application object
const cca = new msal.ConfidentialClientApplication(config);

app.get('/', (req, res) => {
    const authCodeUrlParameters = {
        scopes: ["user.read"],
        redirectUri: "http://localhost:3000/redirect",
    };

    // get url to sign user in and consent to scopes needed for application
    cca.getAuthCodeUrl(authCodeUrlParameters).then((response) => {
        res.redirect(response);
    }).catch((error) => console.log(JSON.stringify(error)));
});

app.get('/redirect', (req, res) => {
    const tokenRequest = {
        code: req.query.code,
        scopes: ["user.read"],
        redirectUri: "http://localhost:3000/redirect",
    };

    cca.acquireTokenByCode(tokenRequest).then((response) => {
        console.log("\nResponse: \n:", response);
        res.sendStatus(200);
    }).catch((error) => {
        console.log(error);
        res.status(500).send(error);
    });
});
```

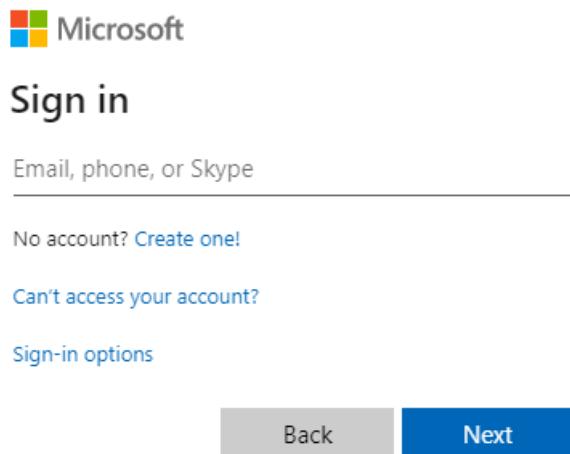
Test sign in

You've completed creation of the application and are now ready to test the app's functionality.

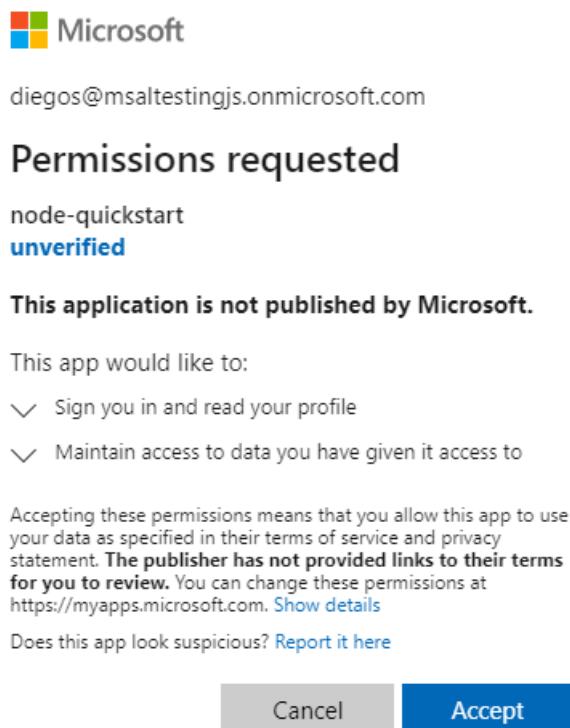
1. Start the Node.js console app by running the following command from within the root of your project folder:

```
node index.js
```

2. Open a browser window and navigate to <http://localhost:3000>. You should see a sign-in screen:



3. Once you enter your credentials, you should see a consent screen asking you to approve the permissions for the app.



How the application works

In this tutorial, you initialized an MSAL Node [ConfidentialClientApplication](#) object by passing it a configuration object (*msalConfig*) that contains parameters obtained from your Azure AD app registration on Azure portal. The web app you created uses the [OAuth 2.0 Authorization code grant flow](#) to sign-in users and obtain ID and access tokens.

Next steps

If you'd like to dive deeper into Node.js & Express web application development on the Microsoft identity

platform, see our multi-part scenario series:

[Scenario: Web app that signs in users](#)

Microsoft identity platform code samples

4/12/2022 • 9 minutes to read • [Edit Online](#)

These code samples are built and maintained by Microsoft to demonstrate usage of our authentication libraries with the Microsoft identity platform. Common authentication and authorization scenarios are implemented in several [application types](#), development languages, and frameworks.

- Sign in users to web applications and provide authorized access to protected web APIs.
- Protect a web API by requiring an access token to perform API operations.

Each code sample includes a *README.md* file describing how to build the project (if applicable) and run the sample application. Comments in the code help you understand how these libraries are used in the application to perform authentication and authorization by using the identity platform.

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Angular	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Call .NET Core web API• Call .NET Core web API (B2C)• Call Microsoft Graph via OBO• Call .NET Core web API using PoP• Use App Roles for access control• Use Security Groups for access control• Deploy to Azure Storage and App Service	MSAL Angular	<ul style="list-style-type: none">• Authorization code with PKCE• On-behalf-of (OBO)• Proof of Possession (PoP)
Blazor WebAssembly	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Deploy to Azure App Service	MSAL.js	Implicit Flow

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
JavaScript	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API via OBO and CA • Deploy to Azure Storage and App Service 	MSAL.js	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA)
React	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API using PoP • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service • Deploy to Azure Static Web Apps 	MSAL React	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA) • Proof of Possession (PoP)

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Customize token cache • Call Graph (multi-tenant) • Call Azure REST APIs • Protect web API • Protect web API (B2C) • Protect multi-tenant web API • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service 	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	<ul style="list-style-type: none"> • OpenID connect • Authorization code • On-Behalf-Of

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Blazor	Blazor Server Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call web API • Call web API (B2C) 	MSAL.NET	Authorization code Grant Flow
ASP.NET Core	Advanced Token Cache Scenarios	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	On-Behalf-Of (OBO)
ASP.NET Core	Use the Conditional Access auth context to perform step-up authentication	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	Authorization code
ASP.NET Core	Active Directory FS to Azure AD migration	MSAL.NET	<ul style="list-style-type: none"> • SAML • OpenID connect
ASP.NET	<ul style="list-style-type: none"> • Microsoft Graph Training Sample • Sign in users and call Microsoft Graph • Sign in users and call Microsoft Graph with admin restricted scope • Quickstart: Sign in users 	MSAL.NET	<ul style="list-style-type: none"> • OpenID connect • Authorization code
Java Spring	Azure AD Spring Boot Starter Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Groups for access control • Deploy to Azure App Service 	<ul style="list-style-type: none"> • MSAL Java • Azure AD Boot Starter 	Authorization code
Java Servlets	Spring-less Servlet Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure App Service 	MSAL Java	Authorization code
Java	Sign in users and call Microsoft Graph	MSAL Java	Authorization code
Java Spring	Sign in users and call Microsoft Graph via OBO • Web API	MSAL Java	<ul style="list-style-type: none"> • Authorization code • On-Behalf-Of (OBO)

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js Express	Express web app series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service • Use App Roles for access control • Use Security Groups for access control • Web app that sign in users 	MSAL Node	Authorization code
Python Flask	Flask Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Sign in users and call Microsoft Graph • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Python Django	Django Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Ruby	Graph Training <ul style="list-style-type: none"> • Sign in users and call Microsoft Graph 	OmniAuth OAuth2	Authorization code

Web API

The following samples show how to protect a web API with the Microsoft identity platform, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET	Call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
ASP.NET Core	Sign in users and call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
Java	Sign in users	MSAL Java	On-Behalf-Of (OBO)
Node.js	<ul style="list-style-type: none"> • Protect a Node.js web API • Protect a Node.js Web API with Azure AD B2C 	MSAL Node	Authorization bearer

Desktop

The following samples show public client desktop applications that access the Microsoft Graph API, or your own

web API in the name of the user. Apart from the *Desktop (Console) with Web Authentication Manager (WAM)* sample, all these client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> • Call Microsoft Graph • Call Microsoft Graph with token cache • Call Microsoft Graph with custom web UI HTML • Call Microsoft Graph with custom web browser • Sign in users with device code flow 	MSAL.NET	<ul style="list-style-type: none"> • Authorization code with PKCE • Device code
.NET	<ul style="list-style-type: none"> • Call Microsoft Graph with daemon console • Call web API with daemon console 	MSAL.NET	Authorization code with PKCE
.NET	Invoke protected API with integrated Windows authentication	MSAL.NET	Integrated Windows authentication
Java	Call Microsoft Graph	MSAL Java	Integrated Windows authentication
Node.js	Sign in users	MSAL Node	Authorization code with PKCE
PowerShell	Call Microsoft Graph by signing in users using username/password	MSAL.NET	Resource owner password credentials
Python	Sign in users	MSAL Python	Resource owner password credentials
Universal Window Platform (UWP)	Call Microsoft Graph	MSAL.NET	Web account manager
Windows Presentation Foundation (WPF)	Sign in users and call Microsoft Graph	MSAL.NET	Authorization code with PKCE
XAML	<ul style="list-style-type: none"> • Sign in users and call ASP.NET core web API • Sign in users and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Mobile

The following samples show public client mobile applications that access the Microsoft Graph API, or your own web API in the name of the user. These client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
iOS	<ul style="list-style-type: none"> Call Microsoft Graph native Call Microsoft Graph with Azure AD nxauth 	MSAL iOS	Authorization code with PKCE
Java	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Kotlin	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Xamarin	<ul style="list-style-type: none"> Sign in users and call Microsoft Graph Sign in users with broker and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Service / daemon

The following samples show an application that accesses the Microsoft Graph API with its own identity (with no user).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> Call Microsoft Graph Call web API Call own web API Using managed identity and Azure key vault 	MSAL.NET	Client credentials grant
ASP.NET	Multi-tenant with Microsoft identity platform endpoint	MSAL.NET	Client credentials grant
Java	Call Microsoft Graph	MSAL Java	Client credentials grant
Node.js	Sign in users and call web API	MSAL Node	Client credentials grant
Python	<ul style="list-style-type: none"> Call Microsoft Graph with secret Call Microsoft Graph with certificate 	MSAL Python	Client credentials grant

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, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET	.NET Azure function web API secured by Azure AD	MSAL.NET	Authorization code

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Node.js Azure function web API secured by Azure AD	MSAL Node	Authorization bearer
Node.js	Call Microsoft Graph API on behalf of a user	MSAL Node	On-Behalf-Of (OBO)
Python	Python Azure function web API secured by Azure AD	MSAL Python	Authorization code

Headless

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 the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET core	Invoke protected API from text-only device	MSAL.NET	Device code
Java	Sign in users and invoke protected API	MSAL Java	Device code
Python	Call Microsoft Graph	MSAL Python	Device code

Microsoft Teams applications

The following sample illustrates Microsoft Teams Tab application that signs in users. Additionally it demonstrates how to call Microsoft Graph API with the user's identity using the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Teams Tab app: single sign-on (SSO) and call Microsoft Graph	MSAL Node	On-Behalf-Of (OBO)

Multi-tenant SaaS

The following samples show how to configure your application to accept sign-ins from any Azure Active Directory (Azure AD) tenant. Configuring your application to be *multi-tenant* means that you can offer a **Software as a Service** (SaaS) application to many organizations, allowing their users to be able to sign-in to your application after providing consent.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core MVC web application calls Microsoft Graph API	MSAL.NET	OpenID connect
ASP.NET Core	ASP.NET Core MVC web application calls ASP.NET Core Web API	MSAL.NET	Authorization code

Next steps

If you'd like to delve deeper into more sample code, see:

- [Sign in users and call the Microsoft Graph API from an Angular](#)
- [Sign in users in a Nodejs and Express web app](#)
- [Call the Microsoft Graph API from a Universal Windows Platform](#)

Scenario: Web app that signs in users

4/12/2022 • 3 minutes to read • [Edit Online](#)

Learn all you need to build a web app that uses the Microsoft identity platform to sign in users.

Getting started

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

If you want to create your first portable (ASP.NET Core) web app that signs 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 in order to customize the experience for users.

Web apps authenticate a user in a web browser. In this scenario, the web app directs the user's browser to sign them in to Azure Active Directory (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 takes advantage of the [Open ID Connect](#) standard protocol, simplified by the use of middleware [libraries](#).



Web app

As a second phase, you can 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 that 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 yet require the Microsoft Authentication Library (MSAL), which is about acquiring a token to call protected APIs. Authentication libraries for .NET will 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 URIs. In some cases (ASP.NET and 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 your web app delegates sign-in. You might want to customize token validation (in particular, in partner scenarios).
- Web applications support any account types. For more information, see [Supported account types](#).

Recommended reading

If you're new to identity and access management (IAM) with OAuth 2.0 and OpenID Connect, or even just new to IAM on the Microsoft identity platform, the following set of articles should be high on your reading list.

Although not required reading before completing your first quickstart or tutorial, they cover topics integral to the platform, and familiarity with them will help you on your path as you build more complex scenarios.

Authentication and authorization

- [Authentication basics](#)
- [ID tokens](#)
- [Access tokens](#)

Microsoft identity platform

- [Audiences](#)
- [Applications and service principals](#)
- [Permissions and consent](#)

Next steps

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

Move on to the next article in this scenario, [App registration](#).

Web app that signs in users: App registration

4/12/2022 • 5 minutes to read • [Edit Online](#)

This article explains the app registration steps 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. 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 affect the sign-out URI.
- The Azure portal to [register your application manually](#).
- PowerShell and command-line tools.

Register an app by using the quickstarts

You can use these links to bootstrap the creation of your web application:

- [ASP.NET Core](#)
- [ASP.NET](#)

Register an app by using the Azure portal

NOTE

The portal to use is different depending on whether 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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

1. When the **Register an application** page appears, enter your application's registration information:
 - a. Enter a **Name** for your application, for example `AspNetCore-WebApp`. Users of your app might see this name, and you can change it later.
 - b. Choose the supported account types for your application. (See [Supported account types](#).)
 - c. For **Redirect URI**, add the type of application and the URI destination that will accept returned token responses after successful authentication. For example, enter `https://localhost:44321`.
 - d. Select **Register**.

2. Under **Manage**, select **Authentication** and then add the following information:

- a. In the **Web** section, add `https://localhost:44321/signin-oidc` as a **Redirect URI**.
- b. In **Front-channel logout URL**, enter `https://localhost:44321/signout-oidc`.
- c. Under **Implicit grant and hybrid flows**, select **ID tokens**.
- d. Select **Save**.

Register an app by using PowerShell

NOTE

Currently, Azure AD PowerShell creates applications with only the following supported account types:

- MyOrg (accounts in this organizational directory only)
- AnyOrg (accounts in any organizational directory)

You can create an application that signs in users with their personal Microsoft accounts (for example, Skype, Xbox, or Outlook.com). First, create a multitenant application. Supported account types are accounts in any organizational directory. Then, change the `accessTokenAcceptedVersion` property to 2 and the `signInAudience` property to `AzureADandPersonalMicrosoftAccount` in the [application manifest](#) from the Azure portal. For more information, see [step 1.3](#) in the ASP.NET Core tutorial. You can generalize this step to web apps in any language.

Next steps

Move on to the next article in this scenario, [App's code configuration](#).

Web app that signs in users: Code configuration

4/12/2022 • 10 minutes to read • [Edit Online](#)

Learn how to configure the code for your web app that signs in users.

Microsoft libraries supporting web apps

The following Microsoft libraries are used to protect a web app (and a web API):

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
.NET	MSAL.NET	Microsoft.Identity.Client	—	✗	✓	GA
.NET	Microsoft.IdentityModel	Microsoft.IdentityModel	—	✗ ²	✗ ²	GA
ASP.NET Core	ASP.NET Core	Microsoft.AspNetCore.Authentication	Quickstart	✓	✗	GA
ASP.NET Core	Microsoft.Identity.Web	Microsoft.Identity.Web	Quickstart	✓	✓	GA
Java	MSAL4J	msal4j	Quickstart	✓	✓	GA
Node.js	MSAL Node	msal-node	Quickstart	✓	✓	GA
Python	MSAL Python	msal	Quickstart	✓	✓	GA

¹ Supplemental terms of use for [Microsoft Azure Previews](#) apply to libraries in *Public preview*.

² The [Microsoft.IdentityModel](#) library only *validates* tokens - it cannot request ID or access tokens.

Select the tab that corresponds to the platform you're interested in:

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [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 by using the Microsoft identity platform are configured through configuration files. These are the values you're required to specify in the configuration:

- The cloud instance (`Instance`) if you want your app to run in national clouds, for example
- The audience in the tenant ID (`TenantId`)
- The client ID (`ClientId`) for your application, as copied from the Azure portal

You might also see references to the `Authority`. The `Authority` value is the concatenation of the `Instance` and `TenantId` values.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [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.partner.microsoftonline.cn/common" for Azure AD China operated by 21Vianet  
    "Instance": "https://login.microsoftonline.com/",  
  
    // Azure AD audience among:  
    // - "TenantId" 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 account  
    // - "common" to sign in users with any work or school account or Microsoft personal account  
    // - "consumers" to sign in users with a 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-oidc"  
  }  
}
```

In ASP.NET Core, another file (`properties\launchSettings.json`) contains the URL (`applicationUrl`) and the TLS/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 redirect URLs that you register on the **Authentication** page for your application need to match these URLs. For the two preceding configuration files, they would be

`https://localhost:44321/signin-oidc`. The reason is that `applicationUrl` is `http://localhost:3110`, but `sslPort` is specified (44321). `CallbackPath` is `/signin-oidc`, as defined in `appsettings.json`.

In the same way, the sign-out URI would be set to `https://localhost:44321/signout-oidc`.

NOTE

`SignedOutCallbackPath` should set either to `portal` or the application to avoid conflict while handling the event.

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 OpenID Connect middleware. The ASP.NET or ASP.NET Core template generates web applications for the Azure Active Directory (Azure AD) v1.0 endpoint. Some configuration is required to adapt them to the Microsoft identity platform. In the case of Java, it's handled by Spring with the cooperation of the application.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [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 that's initializing the application is in the `Startup.cs` file.

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 want to start directly with the new ASP.NET Core templates for Microsoft identity platform, that leverage Microsoft.Identity.Web, you can download a preview NuGet package containing project templates for .NET Core 3.1 and .NET 5.0. Then, once installed, you can directly instantiate ASP.NET Core web applications (MVC or Blazor). See [Microsoft.Identity.Web web app project templates](#) for details. This is the simplest approach as it will do all the steps below for you.

If you prefer to start your project with the current default ASP.NET Core web project within Visual Studio or by using `dotnet new mvc --auth SingleOrg` or `dotnet new webapp --auth SingleOrg`, you'll see code like the following:

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

This code uses the legacy **Microsoft.AspNetCore.Authentication.AzureAD.UI** NuGet package which is used to create an Azure AD v1.0 application. This article explains how to create a Microsoft identity platform (Azure AD v2.0) application which replaces that code.

1. Add the [Microsoft.Identity.Web](#) and [Microsoft.Identity.Web.UI](#) NuGet packages to your project. Remove the `Microsoft.AspNetCore.Authentication.AzureAD.UI` NuGet package if it is present.
2. Update the code in `ConfigureServices` so that it uses the `AddMicrosoftIdentityWebAppAuthentication` and `AddMicrosoftIdentityUI` methods.

```
public class Startup
{
    ...
    // This method gets called by the runtime. Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMicrosoftIdentityWebAppAuthentication(Configuration, "AzureAd");

        services.AddRazorPages().AddMvcOptions(options =>
        {
            var policy = new AuthorizationPolicyBuilder()
                .RequireAuthenticatedUser()
                .Build();
            options.Filters.Add(new AuthorizeFilter(policy));
        }).AddMicrosoftIdentityUI();
    }
}
```

3. In the `Configure` method in `Startup.cs`, enable authentication with a call to `app.UseAuthentication();`

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // more code here
    app.UseAuthentication();
    app.UseAuthorization();
    // more code here
}
```

In the code above:

- The `AddMicrosoftIdentityWebAppAuthentication` extension method is defined in **Microsoft.Identity.Web**. It:
 - Adds the authentication service.

- Configures options to read the configuration file (here from the "AzureAD" section)
 - Configures the OpenID Connect options so that the authority is the Microsoft identity platform.
 - Validates the issuer of the token.
 - Ensures that the claims corresponding to name are mapped from the `preferred_username` claim in the ID token.
- In addition to the configuration object, you can specify the name of the configuration section when calling `AddMicrosoftIdentityWebAppAuthentication`. By default, it's `AzureAd`.
 - `AddMicrosoftIdentityWebAppAuthentication` has other parameters for advanced scenarios. For example, tracing OpenID Connect middleware events can help you troubleshoot your web application if authentication doesn't work. Setting the optional parameter `subscribeToOpenIdConnectMiddlewareDiagnosticsEvents` to `true` will show you how information is processed by the set of ASP.NET Core middleware as it progresses from the HTTP response to the identity of the user in `HttpContext.User`.
 - The `AddMicrosoftIdentityUI` extension method is defined in **Microsoft.Identity.Web.UI**. It provides a default controller to handle sign-in and sign-out.

You can find more details about how Microsoft.Identity.Web enables you to create web apps in <https://aka.ms/ms-id-web/webapp>

Next steps

In the next article, you'll learn how to trigger sign-in and sign-out.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

Move on to the next article in this scenario, [Sign in and sign out](#).

Web app that signs in users: Sign-in and sign-out

4/12/2022 • 8 minutes to read • [Edit Online](#)

Learn how to add sign-in to the code for your web app that signs in users. Then, learn how to let them sign out.

Sign-in

Sign-in consists of two parts:

- The sign-in button on the HTML page
- The sign-in action in the code-behind in the controller

Sign-in button

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

In ASP.NET Core, for Microsoft identity platform applications, the **Sign in** button is exposed in

`Views\Shared_LoginPartial.cshtml` (for an MVC app) or `Pages\Shared_LoginPartial.cshtm` (for a Razor app). It's displayed only when the user isn't authenticated. That is, it's displayed when the user hasn't yet signed in or has signed out. On the contrary, The **Sign out** button is displayed when the user is already signed-in. Note that the Account controller is defined in the **Microsoft.Identity.Web.UI** NuGet package, in the Area named

MicrosoftIdentity

```
<ul class="navbar-nav">
    @if (User.Identity.IsAuthenticated)
    {
        <li class="nav-item">
            <span class="navbar-text text-dark">Hello @User.Identity.Name!</span>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="MicrosoftIdentity" asp-controller="Account" asp-action="SignOut">Sign out</a>
        </li>
    }
    else
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="MicrosoftIdentity" asp-controller="Account" asp-action="SignIn">Sign in</a>
        </li>
    }
</ul>
```

SignIn action of the controller

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

In ASP.NET, selecting the **Sign-in** button in 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. That's no longer the case because the controller is now part of the `Microsoft.Identity.Web.UI` NuGet package. See [AccountController.cs](#) for details.

This controller also handles the Azure AD B2C applications.

After the user has signed in to your app, you'll want to enable them to sign out.

Sign-out

Siging out from a web app involves 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 will reauthenticate to your app without entering their credentials again. The reason is that they'll have a valid single sign-in session with the Microsoft identity platform.

To learn more, see the [Send a sign-out request](#) section in the [Microsoft identity platform and the OpenID Connect protocol](#) documentation.

Application registration

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

During the application registration, you register a front-channel logout URL. In our tutorial, you registered `https://localhost:44321/signout-oidc` in the **Front-channel logout URL** field on the **Authentication** page. For details, see [Register the webApp app](#).

Sign-out button

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

In ASP.NET, selecting the **Sign out** button in the web app triggers the `Signout` action on the `AccountController` controller (see below)

```

<ul class="navbar-nav">
    @if (User.Identity.IsAuthenticated)
    {
        <li class="nav-item">
            <span class="navbar-text text-dark">Hello @User.Identity.Name!</span>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="MicrosoftIdentity" asp-controller="Account" asp-action="SignOut">Sign out</a>
        </li>
    }
    else
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="MicrosoftIdentity" asp-controller="Account" asp-action="SignIn">Sign in</a>
        </li>
    }
</ul>

```

`SignOut` action of the controller

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

In previous versions of the ASP.NET core templates, the `Account` controller was embedded with the web app. That's no longer the case because the controller is now part of the `Microsoft.Identity.Web.UI` NuGet package. See [AccountController.cs](#) for details.

- 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 OpenID Connect middleware contact the Microsoft identity platform `logout` endpoint. The endpoint then:
 - Clears the session cookie from the browser.
 - Calls back the post-logout redirect URI. By default, the post-logout redirect URI displays the signed-out view page `SignedOut.cshtml.cs`. This page is also provided as part of Microsoft.Identity.Web.

Intercepting the call to the `logout` endpoint

The post-logout URI enables applications to participate in the global sign-out.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

The ASP.NET Core OpenID Connect middleware enables your app to intercept the call to the Microsoft identity platform `logout` endpoint by providing an OpenID Connect event named `OnRedirectToIdentityProviderForSignout`. This is handled automatically by Microsoft.Identity.Web (which clears accounts in the case where your web app calls web apis)

Protocol

If you want to learn more about sign-out, read the protocol documentation that's available from [Open ID Connect](#).

Next steps

Move on to the next article in this scenario, [Move to production](#).

Web app that signs in users: Move to production

4/12/2022 • 2 minutes to read • [Edit Online](#)

Now that you know how to get a token to call web APIs, here are some things to consider when moving your application to production.

Enable logging

To help in debugging and authentication failure troubleshooting scenarios, the Microsoft Authentication Library provides built-in logging support. Logging for each library is covered in the following articles:

- [Logging in MSAL.NET](#)
- [Logging in MSAL for Android](#)
- [Logging in MSAL.js](#)
- [Logging in MSAL for iOS/macOS](#)
- [Logging in MSAL for Java](#)
- [Logging in MSAL for Python](#)

Here are some suggestions for data collection:

- Users might ask for help when they have problems. A best practice is to capture and temporarily store logs. Provide a location where users can upload the logs. MSAL provides logging extensions to capture detailed information about authentication.
- If telemetry is available, enable it through MSAL to gather data about how users sign in to your app.

Validate your integration

Test your integration by following the [Microsoft identity platform integration checklist](#).

Build for resilience

Learn how to increase resiliency in your app. For details, see [Increase resilience of authentication and authorization applications you develop](#)

Troubleshooting

When users sign-in to the web application for the first time, they will need to consent. However, in some organizations, users can see a message like the following: *AppName needs permissions 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.* This is because your tenant administrator has **disabled** the ability for users to consent. In that case, contact your tenant administrators so that they do an admin-consent for the scopes required by the application.

Same site

Make sure you understand possible issues with new versions of the Chrome browser: [How to handle SameSite cookie changes in Chrome browser](#).

The Microsoft.Identity.Web NuGet package handles the most common SameSite issues.

Deep dive: ASP.NET Core web app tutorial

Learn about other ways to sign in users with this ASP.NET Core tutorial:

[Enable your web apps to sign in users and call APIs with the Microsoft identity platform for developers](#)

This progressive tutorial has 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 accounts
- [Azure AD B2C](#)
- National clouds

Tutorial: Node.js web app

Learn more about the Node.js web in this tutorial:

[Tutorial: Sign-in users in a Node.js & Express web app](#)

Sample code: Java web app

Learn more about the Java web app from this sample on GitHub:

[A Java Web application that signs in users with the Microsoft identity platform and calls Microsoft Graph](#)

Next Steps

After 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](#).

Scenario: A web app that authenticates users and calls web APIs

4/12/2022 • 2 minutes to read • [Edit Online](#)

Learn how to build a web app that signs users in to the Microsoft identity platform, and then calls web APIs on behalf of the signed-in user.

Prerequisites

This scenario assumes you've already completed [Scenario: Web app that signs in users](#).

Overview

You add authentication to your web app so that it can sign users in and call a web API on behalf of the signed-in user.



Web apps that call web APIs are confidential client applications. That's why they register a secret (an application password or certificate) with Azure Active Directory (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 is about protecting the web app itself. That protection is achieved by using *middleware* libraries, not the Microsoft Authentication Library (MSAL). The preceding scenario, [Web app that signs in users](#), covered that subject.

This scenario covers how to call web APIs from a web app. You must get access tokens for those web APIs. You use MSAL libraries to acquire these tokens.

Development for this scenario involves these specific tasks:

- During [application registration](#), you must provide a reply URI, secret, or certificate to be shared with Azure AD. If you deploy your app to several locations, you'll provide a reply URI for each location.
- The [application configuration](#) must provide the client credentials that were shared with Azure AD during application registration.

Recommended reading

If you're new to identity and access management (IAM) with OAuth 2.0 and OpenID Connect, or even just new to IAM on the Microsoft identity platform, the following set of articles should be high on your reading list.

Although not required reading before completing your first quickstart or tutorial, they cover topics integral to the platform, and familiarity with them will help you on your path as you build more complex scenarios.

Authentication and authorization

- [Authentication basics](#)
- [ID tokens](#)
- [Access tokens](#)

Microsoft identity platform

- [Audiences](#)
- [Applications and service principals](#)
- [Permissions and consent](#)

Next steps

Move on to the next article in this scenario, [App registration](#).

A web app that calls web APIs: App registration

4/12/2022 • 2 minutes to read • [Edit Online](#)

A web app that calls web APIs has the same registration as a web app that signs users in. So, follow the instructions in [A web app that signs in users: App registration](#).

However, because the web app now also calls web APIs, it becomes a confidential client application. That's why some extra registration is required. The app must share client credentials, or *secrets*, with the Microsoft identity platform.

Add a client secret or certificate

As with any confidential client application, you need to add a secret or certificate to act as that application's *credentials* so it can authenticate as itself, without user interaction.

You can add credentials to your client app's registration by using the [Azure portal](#) or by using a command-line tool like [PowerShell](#).

Add client credentials by using the Azure portal

To add credentials to your confidential client application's app registration, follow the steps in [Quickstart: Register an application with the Microsoft identity platform](#) for the type of credential you want to add:

- [Add a client secret](#)
- [Add a certificate](#)

Add client credentials by using PowerShell

Alternatively, you can add credentials when you register your application with the Microsoft identity platform by using PowerShell.

The [active-directory-dotnetcore-daemon-v2](#) code sample on GitHub shows how to add an application secret or certificate when registering an application:

- For details on how to add a **client secret** with PowerShell, see [AppCreationScripts/Configure.ps1](#).
- For details on how to add a **certificate** with PowerShell, see [AppCreationScripts-withCert/Configure.ps1](#).

API permissions

Web apps call APIs on behalf of the signed-in user. To do that, they must request *delegated permissions*. For details, see [Add permissions to access your web API](#).

Next steps

Move on to the next article in this scenario, [Code configuration](#).

A web app that calls web APIs: Code configuration

4/12/2022 • 11 minutes to read • [Edit Online](#)

As shown in the [Web app that signs in users](#) scenario, the web app uses the [OAuth 2.0 authorization code flow](#) to sign the user in. This flow has two steps:

1. Request an authorization code. This part delegates a private dialogue with the user to the Microsoft identity platform. During that dialogue, the user signs in and consents to the use of web APIs. When the private dialogue ends successfully, the web app receives an authorization code on its redirect URI.
2. Request an access token for the API by redeeming the authorization code.

The [Web app that signs in users](#) scenarios covered only the first step. Here you learn how to modify your web app so that it not only signs users in but also now calls web APIs.

Microsoft libraries supporting web apps

The following Microsoft libraries support web apps:

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
.NET	MSAL.NET	Microsoft.Identity.Client	—	✗	✓	GA
.NET	Microsoft.IdentityModel	Microsoft.IdentityModel	—	✗ ²	✗ ²	GA
ASP.NET Core	ASP.NET Core	Microsoft.AspNetCore.Authentication	Quickstart	✓	✗	GA
ASP.NET Core	Microsoft.Identity.Web	Microsoft.Identity.Web	Quickstart	✓	✓	GA
Java	MSAL4J	msal4j	Quickstart	✓	✓	GA
Node.js	MSAL Node	msal-node	Quickstart	✓	✓	GA
Python	MSAL Python	msal	Quickstart	✓	✓	GA

¹ Supplemental terms of use for [Microsoft Azure Previews](#) apply to libraries in *Public preview*.

² The [Microsoft.IdentityModel](#) library only *validates* tokens - it cannot request ID or access tokens.

Select the tab for the platform you're interested in:

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

Client secrets or client certificates

Given that your web app now calls a downstream web API, provide a client secret or client certificate in the `appsettings.json` file. You can also add a section that specifies:

- The URL of the downstream web API
- The scopes required for calling the API

In the following example, the `GraphBeta` section specifies these settings.

```
{  
  "AzureAd": {  
    "Instance": "https://login.microsoftonline.com/",  
    "ClientId": "[Client_id-of-web-app-eg-2ec40e65-ba09-4853-bcde-bcb60029e596]",  
    "TenantId": "common",  
  
    // To call an API  
    "ClientSecret": "[Copy the client secret added to the app from the Azure portal]",  
    "ClientCertificates": [  
    ]  
  },  
  "GraphBeta": {  
    "BaseUrl": "https://graph.microsoft.com/beta",  
    "Scopes": "user.read"  
  }  
}
```

Instead of a client secret, you can provide a client certificate. The following code snippet shows using a certificate stored in Azure Key Vault.

```
{  
  "AzureAd": {  
    "Instance": "https://login.microsoftonline.com/",  
    "ClientId": "[Client_id-of-web-app-eg-2ec40e65-ba09-4853-bcde-bcb60029e596]",  
    "TenantId": "common",  
  
    // To call an API  
    "ClientCertificates": [  
      {  
        "SourceType": "KeyVault",  
        "KeyVaultUrl": "https://msidentitywebsamples.vault.azure.net",  
        "KeyVaultCertificateName": "MicrosoftIdentitySamplesCert"  
      }  
    ]  
  },  
  "GraphBeta": {  
    "BaseUrl": "https://graph.microsoft.com/beta",  
    "Scopes": "user.read"  
  }  
}
```

`Microsoft.Identity.Web` provides several ways to describe certificates, both by configuration or by code. For details, see [Microsoft.Identity.Web - Using certificates](#) on GitHub.

Startup.cs

Your web app will need to acquire a token for the downstream API. You specify it by adding the

`.EnableTokenAcquisitionToCallDownstreamApi()` line after `.AddMicrosoftIdentityWebApp(Configuration)`. This line exposes the `ITokenAcquisition` service that you can use in your controller and page actions. However, as you'll see in the following two options, it can be done more simply. You'll also need to choose a token cache

implementation, for example `.AddInMemoryTokenCaches()`, in *Startup.cs*:

```
using Microsoft.Identity.Web;

public class Startup
{
    // ...
    public void ConfigureServices(IServiceCollection services)
    {
        // ...
        services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
            .AddMicrosoftIdentityWebApp(Configuration, "AzureAd")
            .EnableTokenAcquisitionToCallDownstreamApi(new string[]{"user.read"})
            .AddInMemoryTokenCaches();
        // ...
    }
    // ...
}
```

The scopes passed to `EnableTokenAcquisitionToCallDownstreamApi` are optional, and enable your web app to request the scopes and the user's consent to those scopes when they log in. If you don't specify the scopes, *Microsoft.Identity.Web* will enable an incremental consent experience.

If you don't want to acquire the token yourself, *Microsoft.Identity.Web* provides two mechanisms for calling a web API from a web app. The option you choose depends on whether you want to call Microsoft Graph or another API.

Option 1: Call Microsoft Graph

If you want to call Microsoft Graph, *Microsoft.Identity.Web* enables you to directly use the `GraphServiceClient` (exposed by the Microsoft Graph SDK) in your API actions. To expose Microsoft Graph:

1. Add the [Microsoft.Identity.Web.MicrosoftGraph](#) NuGet package to your project.
2. Add `.AddMicrosoftGraph()` after `.EnableTokenAcquisitionToCallDownstreamApi()` in the *Startup.cs* file.
`.AddMicrosoftGraph()` has several overrides. Using the override that takes a configuration section as a parameter, the code becomes:

```
using Microsoft.Identity.Web;

public class Startup
{
    // ...
    public void ConfigureServices(IServiceCollection services)
    {
        // ...
        services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
            .AddMicrosoftIdentityWebApp(Configuration, "AzureAd")
            .EnableTokenAcquisitionToCallDownstreamApi(new string[]{"user.read"})
            .AddMicrosoftGraph(Configuration.GetSection("GraphBeta"))
            .AddInMemoryTokenCaches();
        // ...
    }
    // ...
}
```

Option 2: Call a downstream web API other than Microsoft Graph

To call a web API other than Microsoft Graph, *Microsoft.Identity.Web* provides `.AddDownstreamWebApi()`, which requests tokens and calls the downstream web API.

```

using Microsoft.Identity.Web;

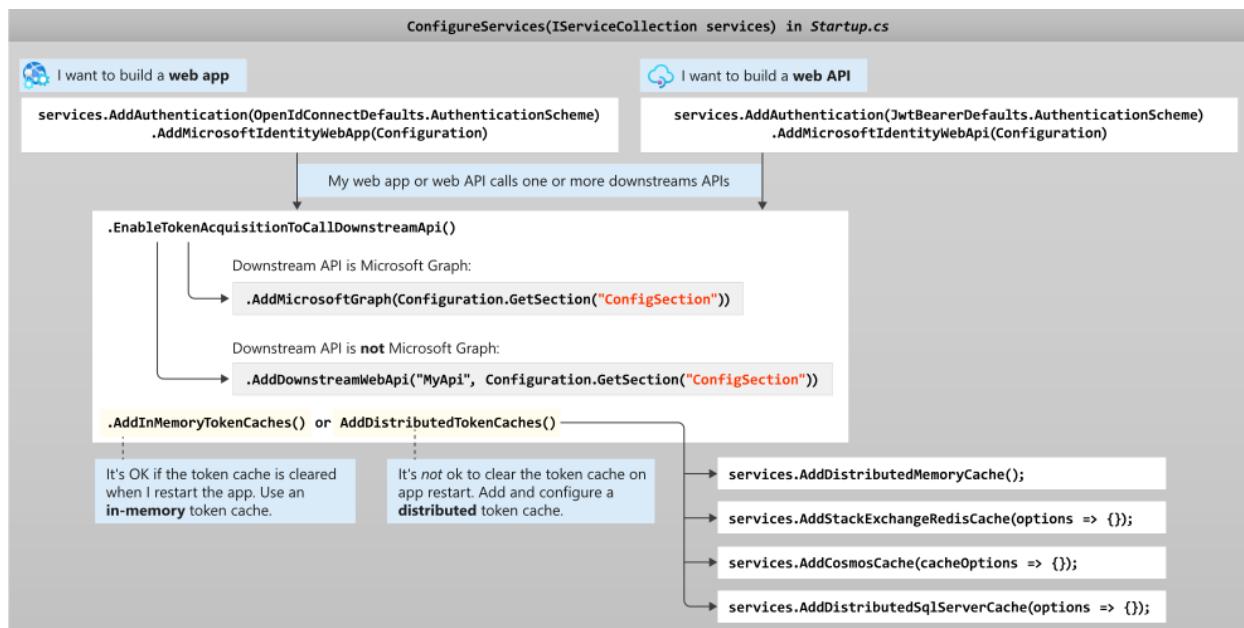
public class Startup
{
    // ...
    public void ConfigureServices(IServiceCollection services)
    {
        // ...
        services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
            .AddMicrosoftIdentityWebApp(Configuration, "AzureAd")
            .EnableTokenAcquisitionToCallDownstreamApi(new string[]{"user.read" })
                .AddDownstreamWebApi("MyApi", Configuration.GetSection("GraphBeta"))
            .AddInMemoryTokenCaches();
        // ...
    }
    // ...
}

```

Summary

As with web APIs, you can choose various token cache implementations. For details, see [Microsoft.Identity.Web - Token cache serialization](#) on GitHub.

The following image shows the various possibilities of *Microsoft.Identity.Web* and their impact on the *Startup.cs* file:



NOTE

To fully understand the code examples here, be familiar with [ASP.NET Core fundamentals](#), and in particular with [dependency injection](#) and [options](#).

Code that redeems the authorization code

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

Microsoft.Identity.Web simplifies your code by setting the correct OpenID Connect settings, subscribing to the

code received event, and redeeming the code. No extra code is required to redeem the authorization code. See [Microsoft.Identity.Web source code](#) for details on how this works.

Instead of a client secret, the confidential client application can also prove its identity by using a client certificate, or a client assertion. The use of client assertions is an advanced scenario, detailed in [Client assertions](#).

Token cache

IMPORTANT

The token-cache implementation for web apps or web APIs is different from the implementation for desktop applications, which is often [file based](#). For security and performance reasons, it's important to ensure that for web apps and web APIs there is one token cache per user account. You must 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 pre-built token-cache serializers described in [Token cache serialization](#). An interesting possibility is to choose ASP.NET Core [distributed memory caches](#):

```
// Use a distributed token cache by adding:  
services.AddMicrosoftIdentityWebAppAuthentication(Configuration, "AzureAd")  
    .EnableTokenAcquisitionToCallDownstreamApi(  
        initialScopes: new string[] { "user.read" })  
    .AddDistributedTokenCaches();  
  
// 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 Microsoft.Identity.Web's [Token cache serialization](#) article, and the [ASP.NET Core Web app tutorials | Token caches](#) phase of the web apps 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.

[Remove accounts from the cache on global sign-out](#)

A web app that calls web APIs: Remove accounts from the token cache on global sign-out

4/12/2022 • 2 minutes to read • [Edit Online](#)

You learned how to add sign-in to your web app in [Web app that signs in users: Sign-in and sign-out](#).

Sign-out is different for a web app that calls web apis. When the user signs out from your application, or from any application, you must remove the tokens associated with that user from the token cache.

Intercept the callback after single sign-out

To clear the token-cache entry associated with the account that signed out, your application can intercept the after `logout` event. Web apps store access tokens for each user in a token cache. By intercepting the after `logout` callback, your web application can remove the user from the cache.

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

`Microsoft.Identity.Web` takes care of implementing sign-out for you. For details see [Microsoft.Identity.Web source code](#)

Next steps

- [ASP.NET Core](#)
- [ASP.NET](#)
- [Java](#)
- [Python](#)

Move on to the next article in this scenario, [Acquire a token for the web app](#).

A web app that calls web APIs: Acquire a token for the app

4/12/2022 • 4 minutes to read • [Edit Online](#)

You've built your client application object. Now, 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 done in the controller:

- Get a token for the web API by using the token cache. To get this token, you call the MSAL `AcquireTokenSilent` method (or the equivalent in `Microsoft.Identity.Web`).
 - Call the protected API, passing the access token to it as a parameter.
- [ASP.NET Core](#)
 - [ASP.NET](#)
 - [Java](#)
 - [Python](#)

`Microsoft.Identity.Web` adds extension methods that provide convenience services for calling Microsoft Graph or a downstream web API. These methods are explained in detail in [A web app that calls web APIs: Call an API](#). With these helper methods, you don't need to manually acquire a token.

If, however, you do want to manually acquire a token, the following code shows an example of using `Microsoft.Identity.Web` to do so in a home controller. It calls Microsoft Graph using the REST API (instead of the Microsoft Graph SDK). To get a token to call the downstream API, you inject the `ITokenAcquisition` service by dependency injection in your controller's constructor (or your page constructor if you use Blazor), and you use it in your controller actions, getting a token for the user (`GetAccessTokenForUserAsync`) or for the application itself (`GetAccessTokenForAppAsync`) in a daemon scenario.

The controller methods are protected by an `[Authorize]` attribute that ensures only authenticated users can use the web app.

```
[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 by using dependency injection.

Here's simplified code for the action of the `HomeController`, which gets a token to call Microsoft Graph:

```
[AuthorizeForScopes(SCopes = new[] { "user.read" })]  
public async Task<IActionResult> Profile()  
{  
    // Acquire the access token.  
    string[] scopes = new string[]{"user.read"};  
    string accessToken = await tokenAcquisition.GetAccessTokenForUserAsync(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 better understand 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.

The `AuthorizeForScopes` attribute on top of the controller action (or of the Razor page if you use a Razor template) is provided by `Microsoft.Identity.Web`. It ensures that the user is asked for consent if needed, and incrementally.

There are other complex variations, such as:

- Calling several APIs.
- Processing incremental consent and conditional access.

These advanced steps are covered in chapter 3 of the [3-WebApp-multi-APIs](#) tutorial.

Next steps

Move on to the next article in this scenario, [Call a web API](#).

A web app that calls web APIs: Call a web API

4/12/2022 • 3 minutes to read • [Edit Online](#)

Now that you have a token, you can call a protected web API. You usually call a downstream API from the controller or pages of your web app.

Call a protected web API

Calling a protected web API depends on your language and framework of choice:

- [ASP.NET Core](#)
- [Java](#)
- [Python](#)

When you use *Microsoft.Identity.Web*, you have three usage options for calling an API:

- [Option 1: Call Microsoft Graph with the Microsoft Graph SDK](#)
- [Option 2: Call a downstream web API with the helper class](#)
- [Option 3: Call a downstream web API without the helper class](#)

Option 1: Call Microsoft Graph with the SDK

You want to call Microsoft Graph. In this scenario, you've added `AddMicrosoftGraph` in *Startup.cs* as specified in [Code configuration](#), and you can directly inject the `GraphServiceClient` in your controller or page constructor for use in the actions. The following example Razor page displays the photo of the signed-in user.

```
[Authorize]
[AuthorizeScopes(Scopes = new[] { "user.read" })]
public class IndexModel : PageModel
{
    private readonly GraphServiceClient _graphServiceClient;

    public IndexModel(GraphServiceClient graphServiceClient)
    {
        _graphServiceClient = graphServiceClient;
    }

    public async Task OnGet()
    {
        var user = await _graphServiceClient.Me.Request().GetAsync();
        try
        {
            using (var photoStream = await _graphServiceClient.Me.Photo.Content.Request().GetAsync())
            {
                byte[] photoByte = ((MemoryStream)photoStream).ToArray();
                ViewData["photo"] = Convert.ToBase64String(photoByte);
            }
            ViewData["name"] = user.DisplayName;
        }
        catch (Exception)
        {
            ViewData["photo"] = null;
        }
    }
}
```

Option 2: Call a downstream web API with the helper class

You want to call a web API other than Microsoft Graph. In that case, you've added `AddDownstreamWebApi` in `Startup.cs` as specified in [Code configuration](#), and you can directly inject an `IDownstreamWebApi` service in your controller or page constructor and use it in the actions:

```
[Authorize]
[AuthorizeForScopes(ScopeKeySection = "TodoList:Scopes")]
public class TodoListController : Controller
{
    private IDownstreamWebApi _downstreamWebApi;
    private const string ServiceName = "TodoList";

    public TodoListController(IDownstreamWebApi downstreamWebApi)
    {
        _downstreamWebApi = downstreamWebApi;
    }

    public async Task<ActionResult> Details(int id)
    {
        var value = await _downstreamWebApi.CallWebApiForUserAsync(
            ServiceName,
            options =>
            {
                options.RelativePath = $"me";
            });
        return View(value);
    }
}
```

The `CallWebApiForUserAsync` also has strongly typed generic overrides that enable you to directly receive an object. For example, the following method receives a `Todo` instance, which is a strongly typed representation of the JSON returned by the web API.

```
// GET: TodoList/Details/5
public async Task<ActionResult> Details(int id)
{
    var value = await _downstreamWebApi.CallWebApiForUserAsync<object, Todo>(
        ServiceName,
        null,
        options =>
        {
            options.HttpMethod = HttpMethod.Get;
            options.RelativePath = $"api/todolist/{id}";
        });
    return View(value);
}
```

Option 3: Call a downstream web API without the helper class

You've decided to acquire a token manually using the `ITokenAcquisition` service, and you now need to use the token. In that case, the following code continues the example code shown in [A web app that calls web APIs: Acquire a token for the app](#). The code is called in the actions of the web app controllers.

After you've acquired the token, use it as a bearer token to call the downstream API, in this case Microsoft Graph.

```
public async Task<IActionResult> Profile()
{
    // Acquire the access token.
    string[] scopes = new string[]{"user.read"};
    string accessToken = await tokenAcquisition.GetAccessTokenForUserAsync(scopes);

    // Use the access token to call a protected web API.
    HttpClient httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", 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);
        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 the API as is the case for Microsoft Graph. See, for instance, [Create a web application that authorizes access to Blob storage with Azure AD](#) for an example of a web app using Microsoft.Identity.Web and using the Azure Storage SDK.

Next steps

Move on to the next article in this scenario, [Move to production](#).

A web app that calls web APIs: Move to production

4/12/2022 • 2 minutes to read • [Edit Online](#)

Now that you know how to acquire a token to call web APIs, here are some things to consider when moving your application to production.

Enable logging

To help in debugging and authentication failure troubleshooting scenarios, the Microsoft Authentication Library provides built-in logging support. Logging is each library is covered in the following articles:

- [Logging in MSAL.NET](#)
- [Logging in MSAL for Android](#)
- [Logging in MSAL.js](#)
- [Logging in MSAL for iOS/macOS](#)
- [Logging in MSAL for Java](#)
- [Logging in MSAL for Python](#)

Here are some suggestions for data collection:

- Users might ask for help when they have problems. A best practice is to capture and temporarily store logs. Provide a location where users can upload the logs. MSAL provides logging extensions to capture detailed information about authentication.
- If telemetry is available, enable it through MSAL to gather data about how users sign in to your app.

Validate your integration

Test your integration by following the [Microsoft identity platform integration checklist](#).

Build for resilience

Learn how to increase resiliency in your app. For details, see [Increase resilience of authentication and authorization applications you develop](#)

Next steps

Learn more by trying out the full, progressive tutorial for ASP.NET Core web apps. The tutorial:

- Shows how to sign users in to multiple audiences or to national clouds, or by using social identities.
- Calls Microsoft Graph.
- Calls several Microsoft APIs.
- Handles incremental consent.
- Calls your own web API.

[ASP.NET Core web app tutorial](#)

Quickstart: Protect a web API with the Microsoft identity platform

4/12/2022 • 12 minutes to read • [Edit Online](#)

In this quickstart, you download and run a code sample that demonstrates how to protect an ASP.NET web API by restricting access to its resources to authorized accounts only. The sample supports authorization of personal Microsoft accounts and accounts in any Azure Active Directory (Azure AD) organization.

The article also uses a Windows Presentation Foundation (WPF) app to demonstrate how you can request an access token to access a web API.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- Visual Studio 2017 or 2019. Download [Visual Studio for free](#).

Clone or download the sample

You can obtain the sample in either of two ways:

- Clone it from your shell or command line:

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

- [Download it as a ZIP file](#).

TIP

To avoid errors caused by path length limitations in Windows, we recommend extracting the archive or cloning the repository into a directory near the root of your drive.

Register the web API (TodoListService)

Register your web API in [App registrations](#) in the Azure portal.

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directory + subscription** filter  in the top menu to select the tenant in which you want to register an application.
3. Find and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application, for example `AppModelv2-NativeClient-DotNet-TodoListService`. Users of your app might see this name, and you can change it later.
6. For **Supported account types**, select **Accounts in any organizational directory**.
7. Select **Register** to create the application.
8. On the app **Overview** page, look for the **Application (client) ID** value, and then record it for later use.

You'll need it to configure the Visual Studio configuration file for this project (that is, `<ClientId>` in the `TodoListService\Web.config` file).

9. Under **Manage**, select **Expose an API > Add a scope**. Accept the proposed Application ID URI (`api://<clientId>`) by selecting **Save and continue**, and then enter the following information:

- a. For **Scope name**, enter `access_as_user`.
- b. For **Who can consent**, ensure that the **Admins and users** option is selected.
- c. In the **Admin consent display name** box, enter `Access TodoListService as a user`.
- d. In the **Admin consent description** box, enter `Accesses the TodoListService web API as a user`.
- e. In the **User consent display name** box, enter `Access TodoListService as a user`.
- f. In the **User consent description** box, enter `Accesses the TodoListService web API as a user`.
- g. For **State**, keep **Enabled**.

10. Select **Add scope**.

Configure the service project

Configure the service project to match the registered web API.

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

Add the new scope to the app.config file

To add the new scope to the `TodoListClient` `app.config` file, follow these steps:

1. In the `TodoListClient` project root folder, open the `app.config` file.
2. Paste the Application ID from the application that you registered for your `TodoListService` project in the `TodoListServiceScope` parameter, replacing the `{Enter the Application ID of your TodoListService from the app registration portal}` string.

NOTE

Make sure that the Application ID 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` app).

Register the web app (TodoListClient)

Register your `TodoListClient` app in **App registrations** in the Azure portal, and then configure the code in the `TodoListClient` project. If the client and server are considered the same application, you can reuse the application that's registered in step 2. Use the same application if you want users to sign in with a personal Microsoft account.

Register the app

To register the `TodoListClient` app, follow these steps:

1. Go to the Microsoft identity platform for developers [App registrations](#) portal.
2. Select **New registration**.
3. When the **Register an application** page opens, 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 (for example, **NativeClient-DotNet-TodoListClient**).
- b. For **Supported account types**, select **Accounts in any organizational directory**.
- c. Select **Register** to create the application.

NOTE

In the TodoListClient project *app.config* file, the default value of `ida:Tenant` is set to `common`. The possible values are:

- `common` : You can sign in by using a work or school account or a personal Microsoft account (because you selected **Accounts in any organizational directory** in a previous step).
- `organizations` : You can sign in by using a work or school account.
- `consumers` : You can sign in only by using a Microsoft personal account.

4. On the app **Overview** page, select **Authentication**, and then complete these steps to add a platform:
 - a. Under **Platform configurations**, select the **Add a platform** button.
 - b. For **Mobile and desktop applications**, select **Mobile and desktop applications**.
 - c. For **Redirect URIs**, select the `https://login.microsoftonline.com/common/oauth2/nativeclient` check box.
 - d. Select **Configure**.
5. Select **API permissions**, and then complete these steps to add permissions:
 - a. Select the **Add a permission** button.
 - b. Select the **My APIs** tab.
 - c. In the list of APIs, select **AppModelv2-NativeClient-DotNet-TodoListService API** or the name you entered for the web API.
 - d. Select the `access_as_user` permission check box if it's not already selected. Use the Search box if necessary.
 - e. Select the **Add permissions** button.

Configure your project

Configure your TodoListClient project by adding the Application ID to the *app.config* file.

1. In the **App registrations** portal, on the **Overview** page, copy the value of the **Application (client) ID**.
2. From the TodoListClient project root folder, open the *app.config* file, and then paste the Application ID value in the `ida:ClientId` parameter.

Run your projects

Start both projects. If you are using Visual Studio:

1. Right click on the Visual Studio solution and select **Properties**
2. In the **Common Properties** select **Startup Project** and then **Multiple startup projects**.
3. For both projects choose **Start** as the action
4. Ensure the TodoListService service starts first by moving it to the fist position in the list, using the up arrow.

Sign in to run your TodoListClient project.

1. Press F5 to start the projects. The service page opens, as well as the desktop application.

2. In the TodoListClient, at the upper right, select **Sign in**, and then sign in with the same credentials you used to register your application, or sign in as a user in the same directory.

If you're signing in for the first time, you might be prompted to consent to the TodoListService web API.

To help you access the TodoListService web API and manipulate the *To-Do* list, the sign-in also requests an access token to the *access_as_user* scope.

Pre-authorize your client application

You can allow users from other directories to access your web API by pre-authorizing the client application to access your web API. You do this by adding the Application ID from the client app to the list of pre-authorized applications for your web API. By adding a pre-authorized client, you're allowing users to access your web API without having to provide consent.

1. In the **App registrations** portal, open the properties of your TodoListService app.
2. In the **Expose an API** section, under **Authorized client applications**, select **Add a client application**.
3. In the **Client ID** box, paste the Application ID of the TodoListClient app.
4. In the **Authorized scopes** section, select the scope for the `api://<Application ID>/access_as_user` web API.
5. Select **Add application**.

Run your project

1. Press F5 to run your project. Your TodoListClient app opens.
2. At the upper right, select **Sign in**, and then sign in by using a personal Microsoft account, such as a *live.com* or *hotmail.com* account, or a work or school account.

Optional: Limit sign-in access to certain users

By default, any personal accounts, such as *outlook.com* or *live.com* accounts, or work or school accounts from organizations that are integrated with Azure AD can request tokens and access your web API.

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

Option 1: Limit access to a single organization (single tenant)

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

1. Open the `App_Start\Startup.Auth` file, and then 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 set 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 this parameter, see [TokenValidationParameters class](#).

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [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

In this quickstart, you download an ASP.NET Core web API code sample and review the way it restricts resource access to authorized accounts only. The sample supports authorization of personal Microsoft accounts and accounts in any Azure Active Directory (Azure AD) organization.

Prerequisites

- Azure account with an active subscription. [Create an account for free.](#)
- [Azure Active Directory tenant](#)
- [.NET Core SDK 3.1+](#)
- [Visual Studio 2019](#) or [Visual Studio Code](#)

Step 1: Register the application

First, register the web API in your Azure AD tenant and add a scope by following these steps:

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. For **Name**, enter a name for your application. For example, enter **AspNetCoreWebApi-Quickstart**. Users of your app will see this name, and you can change it later.
6. Select **Register**.
7. Under **Manage**, select **Expose an API > Add a scope**. For **Application ID URI**, accept the default by selecting **Save and continue**, and then enter the following details:
 - **Scope name:** `access_as_user`
 - **Who can consent?:** **Admins and users**
 - **Admin consent display name:** `Access AspNetCoreWebApi-Quickstart`
 - **Admin consent description:**
`Allows the app to access AspNetCoreWebApi-Quickstart as the signed-in user.`
 - **User consent display name:** `Access AspNetCoreWebApi-Quickstart`
 - **User consent description:**
`Allow the application to access AspNetCoreWebApi-Quickstart on your behalf.`
 - **State:** **Enabled**
8. Select **Add scope** to complete the scope addition.

Step 2: Download the ASP.NET Core project

[Download the ASP.NET Core solution](#) from GitHub.

TIP

To avoid errors caused by path length limitations in Windows, we recommend extracting the archive or cloning the repository into a directory near the root of your drive.

Step 3: Configure the ASP.NET Core project

In this step, configure the sample code to work with the app registration that you created earlier.

1. Extract the .zip archive into a folder near the root of your drive. For example, extract into `C:\Azure-Samples`.

We recommend extracting the archive into a directory near the root of your drive to avoid errors caused by path length limitations on Windows.

2. Open the solution in the `webapi` folder in your code editor.
3. Open the `appsettings.json` file and modify the following code:

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

- Replace `Enter_the_Application_Id_here` with the application (client) ID of the application that you registered in the Azure portal. You can find the application (client) ID on the app's **Overview** page.
- Replace `Enter_the_Tenant_Info_Here` with one of the following:
 - If your application supports **Accounts in this organizational directory only**, replace this value with the directory (tenant) ID (a GUID) or tenant name (for example, `contoso.onmicrosoft.com`). You can find the directory (tenant) ID on the app's **Overview** page.
 - If your application supports **Accounts in any organizational directory**, replace this value with `organizations`.
 - If your application supports **All Microsoft account users**, leave this value as `common`.

For this quickstart, don't change any other values in the `appsettings.json` file.

How the sample works

The web API receives a token from a client application, and the code in the web API validates the token. This scenario is explained in more detail in [Scenario: Protected web API](#).

Startup class

The `Microsoft.AspNetCore.Authentication` middleware uses a `Startup` class that's executed when the hosting process starts. In its `ConfigureServices` method, the `AddMicrosoftIdentityWebApi` extension method provided by `Microsoft.Identity.Web` is called.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddMicrosoftIdentityWebApi(Configuration, "AzureAd");
}
```

The `AddAuthentication()` method configures the service to add JwtBearer-based authentication.

The line that contains `.AddMicrosoftIdentityWebApi` adds the Microsoft identity platform authorization to your web API. It's then configured to validate access tokens issued by the Microsoft identity platform based on the information in the `AzureAD` section of the `appsettings.json` configuration file:

APPSETTINGS.JSON KEY	DESCRIPTION
<code>ClientId</code>	Application (client) ID of the application registered in the Azure portal.

APPSETTINGS.JSON KEY	DESCRIPTION
Instance	Security token service (STS) endpoint for the user to authenticate. This value is typically https://login.microsoftonline.com/ , indicating the Azure public cloud.
TenantId	Name of your tenant or its tenant ID (a GUID), or common to sign in users with work or school accounts or Microsoft personal accounts.

The `Configure()` method contains two important methods, `app.UseAuthentication()` and `app.UseAuthorization()`, that enable their named functionality:

```
// The runtime calls this method. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    // more code
    app.UseAuthentication();
    app.UseAuthorization();
    // more code
}
```

Protecting a controller, a controller's method, or a Razor page

You can protect a controller or controller methods by using the `[Authorize]` attribute. This attribute restricts access to the controller or methods by allowing only authenticated users. An authentication challenge can be started to access the controller if the user isn't authenticated.

```
namespace webapi.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class WeatherForecastController : ControllerBase
```

Validation of scope in the controller

The code in the API verifies that the required scopes are in the token by using `HttpContext.VerifyUserHasAnyAcceptedScope(scopeRequiredByApi);`:

```
namespace webapi.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class WeatherForecastController : ControllerBase
    {
        // The web API will only accept tokens 1) for users, and 2) having the "access_as_user" scope for
        this API
        static readonly string[] scopeRequiredByApi = new string[] { "access_as_user" };

        [HttpGet]
        public IEnumerable<WeatherForecast> Get()
        {
            HttpContext.VerifyUserHasAnyAcceptedScope(scopeRequiredByApi);

            // some code here
        }
    }
}
```

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

The GitHub repository that contains this ASP.NET Core web API code sample includes instructions and more code samples that show you how to:

- Add authentication to a new ASP.NET Core web API.
- Call the web API from a desktop application.
- Call downstream APIs like Microsoft Graph and other Microsoft APIs.

[ASP.NET Core web API tutorials on GitHub](#)

Microsoft identity platform code samples

4/12/2022 • 9 minutes to read • [Edit Online](#)

These code samples are built and maintained by Microsoft to demonstrate usage of our authentication libraries with the Microsoft identity platform. Common authentication and authorization scenarios are implemented in several [application types](#), development languages, and frameworks.

- Sign in users to web applications and provide authorized access to protected web APIs.
- Protect a web API by requiring an access token to perform API operations.

Each code sample includes a *README.md* file describing how to build the project (if applicable) and run the sample application. Comments in the code help you understand how these libraries are used in the application to perform authentication and authorization by using the identity platform.

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Angular	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Call .NET Core web API• Call .NET Core web API (B2C)• Call Microsoft Graph via OBO• Call .NET Core web API using PoP• Use App Roles for access control• Use Security Groups for access control• Deploy to Azure Storage and App Service	MSAL Angular	<ul style="list-style-type: none">• Authorization code with PKCE• On-behalf-of (OBO)• Proof of Possession (PoP)
Blazor WebAssembly	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Deploy to Azure App Service	MSAL.js	Implicit Flow

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
JavaScript	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API via OBO and CA • Deploy to Azure Storage and App Service 	MSAL.js	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA)
React	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API using PoP • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service • Deploy to Azure Static Web Apps 	MSAL React	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA) • Proof of Possession (PoP)

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Customize token cache • Call Graph (multi-tenant) • Call Azure REST APIs • Protect web API • Protect web API (B2C) • Protect multi-tenant web API • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service 	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	<ul style="list-style-type: none"> • OpenID connect • Authorization code • On-Behalf-Of

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Blazor	Blazor Server Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call web API • Call web API (B2C) 	MSAL.NET	Authorization code Grant Flow
ASP.NET Core	Advanced Token Cache Scenarios	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	On-Behalf-Of (OBO)
ASP.NET Core	Use the Conditional Access auth context to perform step-up authentication	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	Authorization code
ASP.NET Core	Active Directory FS to Azure AD migration	MSAL.NET	<ul style="list-style-type: none"> • SAML • OpenID connect
ASP.NET	<ul style="list-style-type: none"> • Microsoft Graph Training Sample • Sign in users and call Microsoft Graph • Sign in users and call Microsoft Graph with admin restricted scope • Quickstart: Sign in users 	MSAL.NET	<ul style="list-style-type: none"> • OpenID connect • Authorization code
Java Spring	Azure AD Spring Boot Starter Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Groups for access control • Deploy to Azure App Service 	<ul style="list-style-type: none"> • MSAL Java • Azure AD Boot Starter 	Authorization code
Java Servlets	Spring-less Servlet Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure App Service 	MSAL Java	Authorization code
Java	Sign in users and call Microsoft Graph	MSAL Java	Authorization code
Java Spring	Sign in users and call Microsoft Graph via OBO • Web API	MSAL Java	<ul style="list-style-type: none"> • Authorization code • On-Behalf-Of (OBO)

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js Express	Express web app series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service • Use App Roles for access control • Use Security Groups for access control • Web app that sign in users 	MSAL Node	Authorization code
Python Flask	Flask Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Sign in users and call Microsoft Graph • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Python Django	Django Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Ruby	Graph Training <ul style="list-style-type: none"> • Sign in users and call Microsoft Graph 	OmniAuth OAuth2	Authorization code

Web API

The following samples show how to protect a web API with the Microsoft identity platform, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET	Call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
ASP.NET Core	Sign in users and call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
Java	Sign in users	MSAL Java	On-Behalf-Of (OBO)
Node.js	<ul style="list-style-type: none"> • Protect a Node.js web API • Protect a Node.js Web API with Azure AD B2C 	MSAL Node	Authorization bearer

Desktop

The following samples show public client desktop applications that access the Microsoft Graph API, or your own

web API in the name of the user. Apart from the *Desktop (Console) with Web Authentication Manager (WAM)* sample, all these client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> • Call Microsoft Graph • Call Microsoft Graph with token cache • Call Microsoft Graph with custom web UI HTML • Call Microsoft Graph with custom web browser • Sign in users with device code flow 	MSAL.NET	<ul style="list-style-type: none"> • Authorization code with PKCE • Device code
.NET	<ul style="list-style-type: none"> • Call Microsoft Graph with daemon console • Call web API with daemon console 	MSAL.NET	Authorization code with PKCE
.NET	Invoke protected API with integrated Windows authentication	MSAL.NET	Integrated Windows authentication
Java	Call Microsoft Graph	MSAL Java	Integrated Windows authentication
Node.js	Sign in users	MSAL Node	Authorization code with PKCE
PowerShell	Call Microsoft Graph by signing in users using username/password	MSAL.NET	Resource owner password credentials
Python	Sign in users	MSAL Python	Resource owner password credentials
Universal Window Platform (UWP)	Call Microsoft Graph	MSAL.NET	Web account manager
Windows Presentation Foundation (WPF)	Sign in users and call Microsoft Graph	MSAL.NET	Authorization code with PKCE
XAML	<ul style="list-style-type: none"> • Sign in users and call ASP.NET core web API • Sign in users and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Mobile

The following samples show public client mobile applications that access the Microsoft Graph API, or your own web API in the name of the user. These client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
iOS	<ul style="list-style-type: none"> Call Microsoft Graph native Call Microsoft Graph with Azure AD nxauth 	MSAL iOS	Authorization code with PKCE
Java	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Kotlin	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Xamarin	<ul style="list-style-type: none"> Sign in users and call Microsoft Graph Sign in users with broker and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Service / daemon

The following samples show an application that accesses the Microsoft Graph API with its own identity (with no user).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> Call Microsoft Graph Call web API Call own web API Using managed identity and Azure key vault 	MSAL.NET	Client credentials grant
ASP.NET	Multi-tenant with Microsoft identity platform endpoint	MSAL.NET	Client credentials grant
Java	Call Microsoft Graph	MSAL Java	Client credentials grant
Node.js	Sign in users and call web API	MSAL Node	Client credentials grant
Python	<ul style="list-style-type: none"> Call Microsoft Graph with secret Call Microsoft Graph with certificate 	MSAL Python	Client credentials grant

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, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET	.NET Azure function web API secured by Azure AD	MSAL.NET	Authorization code

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Node.js Azure function web API secured by Azure AD	MSAL Node	Authorization bearer
Node.js	Call Microsoft Graph API on behalf of a user	MSAL Node	On-Behalf-Of (OBO)
Python	Python Azure function web API secured by Azure AD	MSAL Python	Authorization code

Headless

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 the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET core	Invoke protected API from text-only device	MSAL.NET	Device code
Java	Sign in users and invoke protected API	MSAL Java	Device code
Python	Call Microsoft Graph	MSAL Python	Device code

Microsoft Teams applications

The following sample illustrates Microsoft Teams Tab application that signs in users. Additionally it demonstrates how to call Microsoft Graph API with the user's identity using the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Teams Tab app: single sign-on (SSO) and call Microsoft Graph	MSAL Node	On-Behalf-Of (OBO)

Multi-tenant SaaS

The following samples show how to configure your application to accept sign-ins from any Azure Active Directory (Azure AD) tenant. Configuring your application to be *multi-tenant* means that you can offer a **Software as a Service** (SaaS) application to many organizations, allowing their users to be able to sign-in to your application after providing consent.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core MVC web application calls Microsoft Graph API	MSAL.NET	OpenID connect
ASP.NET Core	ASP.NET Core MVC web application calls ASP.NET Core Web API	MSAL.NET	Authorization code

Next steps

If you'd like to delve deeper into more sample code, see:

- [Sign in users and call the Microsoft Graph API from an Angular](#)
- [Sign in users in a Nodejs and Express web app](#)
- [Call the Microsoft Graph API from a Universal Windows Platform](#)

Scenario: Protected web API

4/12/2022 • 2 minutes to read • [Edit Online](#)

In this scenario, you learn how to expose a web API. You also learn how to protect the web API so that only authenticated users can access it.

To use your web API, you either enable authenticated users with both work and school accounts or enable Microsoft personal accounts.

Specifics

Here is specific information you need to know to protect web APIs:

- Your app registration must expose at least one *scope* or one *application role*.
 - Scopes are exposed by web APIs that are called on behalf of a user.
 - Application roles are exposed by web APIs called by daemon applications (that calls your web API on their own behalf).
- If you create a new web API app registration, choose the [access token version](#) accepted by your web API to 2. For legacy web APIs, the accepted token version can be `null`, but this value restricts the sign-in audience to organizations only, and personal Microsoft accounts (MSA) won't be supported.
- The code configuration for the web API must validate the token used when the web API is called.
- The code in the controller actions must validate the roles or scopes in the token.

Recommended reading

If you're new to identity and access management (IAM) with OAuth 2.0 and OpenID Connect, or even just new to IAM on the Microsoft identity platform, the following set of articles should be high on your reading list.

Although not required reading before completing your first quickstart or tutorial, they cover topics integral to the platform, and familiarity with them will help you on your path as you build more complex scenarios.

Authentication and authorization

- [Authentication basics](#)
- [ID tokens](#)
- [Access tokens](#)

Microsoft identity platform

- [Audiences](#)
- [Applications and service principals](#)
- [Permissions and consent](#)

Next steps

Move on to the next article in this scenario, [App registration](#).

Protected web API: App registration

4/12/2022 • 5 minutes to read • [Edit Online](#)

This article explains the specifics of app registration for a protected web API.

For the common steps to register an app, see [Quickstart: Register an application with the Microsoft identity platform](#).

Accepted token version

The Microsoft identity platform can issue v1.0 tokens and v2.0 tokens. For more information about these tokens, see [Access tokens](#).

The token version your API may accept depends on your **Supported account types** selection when you create your web API application registration in the Azure portal.

- 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 must be v2.0.
- Otherwise, the accepted token version can 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 **Manifest**.
2. Find the property **accessTokenAcceptedVersion** in the manifest.
3. The value specifies to Azure Active Directory (Azure AD) which token version the web API accepts.
 - If the value is 2, the web API accepts v2.0 tokens.
 - If the value is **null**, the web API accepts v1.0 tokens.
4. If you changed the token version, select **Save**.

The web API specifies which token version it accepts. When a client requests a token for your web API from the Microsoft identity platform, the client gets a token that indicates which token version the web API accepts.

No redirect URI

Web APIs don't need to register a redirect URI because no user is interactively signed in.

Exposed API

Other settings specific to web APIs are the exposed API and the exposed scopes or app roles.

Application ID URI and scopes

Scopes usually have the form `resourceURI/scopedName`. For Microsoft Graph, the scopes have shortcuts. For example, `User.Read` is a shortcut for `https://graph.microsoft.com/user.read`.

During app registration, define these parameters:

- The resource URI
- One or more scopes
- One or more app roles

By default, the application registration portal recommends that you use the resource URI `api://{clientId}`. This

URI is unique but not human readable. If you change the URI, make sure the new value is unique. The application registration portal will ensure that you use a [configured publisher domain](#).

To client applications, scopes show up as *delegated permissions* and app roles show up as *application permissions* for your web API.

Scopes also appear on the consent window that's presented to users of your app. Therefore, provide the corresponding strings that describe the scope:

- As seen by a user.
- As seen by a tenant admin, who can grant admin consent.

App roles cannot be consented to by a user (as they're used by an application that call the web API on behalf of itself). A tenant administrator will need to consent to client applications of your web API exposing app roles. See [Admin consent](#) for details.

Exposing delegated permissions (scopes)

1. Select **Expose an API** 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. Specify these values:
 - Select **Scope name** and enter `access_as_user`.
 - Select **Who can consent** and make sure **Admins and users** is selected.
 - Select **Admin consent display name** and enter `Access TodoListService as a user`.
 - Select **Admin consent description** and enter `Accesses the TodoListService web API as a user`.
 - Select **User consent display name** and enter `Access TodoListService as a user`.
 - Select **User consent description** and enter `Accesses the TodoListService web API as a user`.
 - Keep the **State** value set to **Enabled**.
5. Select **Add scope**.

If your web API is called by a daemon app

In this section, you learn how to register your protected web API so that daemon apps can securely call it.

- You declare and expose only *application permissions* because daemon apps don't interact with users. Delegated permissions wouldn't make sense.
- Tenant admins can require Azure AD to issue web API tokens only to applications that have registered to access one of the API's application permissions.

Exposing application permissions (app roles)

To expose application permissions, edit the manifest.

1. In the application registration for your application, select **Manifest**.
2. To edit the manifest, find the `appRoles` setting and add application roles. The role definitions are provided in the following sample JSON block.
3. Leave `allowedMemberTypes` set to `"Application"` only.
4. Make sure `id` is a unique GUID.
5. Make sure `displayName` and `value` don't contain spaces.
6. Save the manifest.

The following sample shows the contents of `appRoles`, where the value of `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. This role is a software developer's way to expose application permissions. You can also configure Azure AD to issue API tokens only to apps that the tenant admin approves for API access.

To add this increased security:

1. Go to the app **Overview** page for your app registration.
2. Under **Managed application in local directory**, select the link with the name of your app. The label for this selection might be truncated. For example, you might see **Managed application in ...**
When you select this link, you go to the **Enterprise Application Overview** page. This page is associated with the service principal for your application in the tenant where you created it. You can go to the app registration page by using the back button of your browser.
3. Select the **Properties** page in the **Manage** section of the Enterprise application pages.
4. If you want Azure AD to allow access to your web API from only certain clients, set **User assignment required?** to **Yes**.

If you set **User assignment required?** to **Yes**, Azure AD checks the app role assignments of a client when it requests a web API access token. If the client isn't assigned to any app roles, Azure AD will return the error message "invalid_client: AADSTS501051: Application <application name> isn't 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, meaning any client using the client credentials flow, can get an access token for the API just by specifying its audience. Any application can access the API without having to request permissions for it.

But as explained in the previous section, your web API can always 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 role claim and that the value for this claim is correct. In the previous JSON sample, the value is `access_as_application`.

5. Select **Save**.

Next steps

Move on to the next article in this scenario, [App code configuration](#).

Protected web API: Code configuration

4/12/2022 • 6 minutes to read • [Edit Online](#)

To configure the code for your protected web API, understand:

- What defines APIs as protected.
- How to configure a bearer token.
- How to validate the token.

What defines ASP.NET and ASP.NET Core APIs as protected?

Like web apps, the ASP.NET and ASP.NET Core web APIs are protected because their controller actions are prefixed with the `[Authorize]` attribute. The controller actions can be called only if the API is called with an authorized identity.

Consider the following questions:

- Only an app can call a web API. How does the API know the identity of the app that calls it?
- If the app calls the API on behalf of a user, what's the user's identity?

Bearer token

The bearer token that's set in the header when the app is called holds information about the app identity. It also holds information about the user unless the web app accepts service-to-service calls from a daemon app.

Here's a C# code example that shows a client calling the API after it acquires a token with the 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

A client application requests the bearer token to the Microsoft identity platform *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.

In the future, the web API might require 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": "*"
}
```

Case where you used a custom App ID URI for your web API

If you've accepted the App ID URI proposed by the app registration portal, you don't need to specify the audience (see [Application ID URI and scopes](#)). Otherwise, you should add an `Audience` property whose value is the App ID URI for your web API.

```
{
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "ClientId": "[Client_id-of-web-api-eg-2ec40e65-ba09-4853-bcde-bcb60029e596]",
    "TenantId": "common",
    "Audience": "custom App ID URI for your web API"
  },
  // more lines
}
```

Code initialization

When an app is called on a controller action that holds an `[Authorize]` attribute, ASP.NET and ASP.NET Core extract the access token from the Authorization header's bearer token. The access token is then forwarded to the JwtBearer middleware, which calls Microsoft IdentityModel Extensions for .NET.

Microsoft.Identity.Web

Microsoft recommends you use the [Microsoft.Identity.Web](#) NuGet package when developing a web API with ASP.NET Core.

Microsoft.Identity.Web provides the glue between ASP.NET Core, the authentication middleware, and the [Microsoft Authentication Library \(MSAL\)](#) for .NET. It allows for a clearer, more robust developer experience and leverages the power of the Microsoft identity platform and Azure AD B2C.

Using Microsoft.Identity.Web templates

You can create a web API from scratch by using Microsoft.Identity.Web project templates. For details see [Microsoft.Identity.Web - Web API project template](#).

Starting from an existing ASP.NET Core 3.1 application

ASP.NET Core 3.1 uses the Microsoft.AspNetCore.AzureAD.UI library. The middleware is initialized in the Startup.cs file.

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
```

The middleware is added to the web API by this instruction:

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(AzureADDefaults.JwtBearerAuthenticationScheme)
        .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. They don't sign in users with personal accounts. However, you can change the templates to use the Microsoft identity platform by using [Microsoft.Identity.Web](#) replacing the code in *Startup.cs*.

```
using Microsoft.Identity.Web;
```

```
public void ConfigureServices(IServiceCollection services)
{
    // Adds Microsoft Identity platform (AAD v2.0) support to protect this API
    services.AddMicrosoftIdentityWebApiAuthentication(Configuration, "AzureAd");

    services.AddControllers();
}
```

you can also write the following (which is equivalent)

```
public void ConfigureServices(IServiceCollection services)
{
    // Adds Microsoft Identity platform (AAD v2.0) support to protect this API
    services.AddAuthentication(AzureADDefaults.JwtBearerAuthenticationScheme)
        .AddMicrosoftIdentityWebApi(Configuration, "AzureAd");

    services.AddControllers();
}
```

NOTE

If you use Microsoft.Identity.Web and don't set the `Audience` in `appsettings.json`, the following is used:

- `"${ClientId}"` if you have set the [access token accepted version](#) to `2`, or for Azure AD B2C web APIs.
- `"api://${ClientId}"` in all other cases (for v1.0 [access tokens](#)). For details, see Microsoft.Identity.Web [source code](#).

The preceding code snippet is extracted from the [ASP.NET Core web API incremental tutorial](#). The detail of [AddMicrosoftIdentityWebApiAuthentication](#) is available in [Microsoft.Identity.Web](#). This method calls [AddMicrosoftIdentityWebAPI](#), which itself instructs the middleware on how to validate the token.

Token validation

In the preceding snippet, the JwtBearer middleware, like the OpenID Connect middleware in web apps, validates the token based on the value of `TokenValidationParameters`. 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:

- Audience: The token is targeted for the web API.
- Sub: It was issued for an app that's allowed to call the web API.
- Issuer: It was issued by a trusted security token service (STS).
- Expiry: Its lifetime is in range.
- Signature: It wasn't tampered with.

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 provided by the [Microsoft IdentityModel Extensions for .NET](#) open-source library. The validators are defined in the library source file [Microsoft.IdentityModel.Tokens/Validators.cs](#).

This table describes the validators:

VALIDATOR	DESCRIPTION
<code>ValidateAudience</code>	Ensures the token is for the application that validates the token for you.
<code>ValidateIssuer</code>	Ensures the token was issued by a trusted STS, meaning it's from someone you trust.
<code>ValidateIssuerSigningKey</code>	Ensures the application validating the token trusts the key that was used to sign the token. There's a special case where the key is embedded in the token. But this case doesn't usually arise.
<code>ValidateLifetime</code>	Ensures the token is still or already valid. The validator checks if the lifetime of the token is in the range specified by the <code>notbefore</code> and <code>expires</code> claims.
<code>ValidateSignature</code>	Ensures the token hasn't been tampered with.
<code>ValidateTokenReplay</code>	Ensures the token isn't replayed. There's a special case for some onetime-use protocols.

Customizing token validation

The validators are associated with properties of the `TokenValidationParameters` class. The properties are initialized from the ASP.NET and ASP.NET Core configuration.

In most cases, you don't need to change the parameters. Apps that aren't single tenants are exceptions. These web apps accept users from any organization or from personal Microsoft accounts. Issuers in this case must be validated. `Microsoft.Identity.Web` takes care of the issuer validation as well.

In ASP.NET Core, if you want to customize the token validation parameters, use the following snippet in your `Startup.cs`:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddMicrosoftIdentityWebApi(Configuration);
services.Configure<JwtBearerOptions>(JwtBearerDefaults.AuthenticationScheme, options =>
{
    var existingOnTokenValidatedHandler = options.Events.OnTokenValidated;
    options.Events.OnTokenValidated = async context =>
    {
        await existingOnTokenValidatedHandler(context);
        // Your code to add extra configuration that will be executed after the current event implementation.
        options.TokenValidationParameters.ValidIssuers = new[] { /* list of valid issuers */ };
        options.TokenValidationParameters.ValidAudiences = new[] { /* list of valid audiences */ };
    };
});
```

For ASP.NET MVC, the following code sample shows how to do custom token validation:

<https://github.com/azure-samples/active-directory-dotnet-webapi-manual-jwt-validation>

Token validation in Azure Functions

You can also validate incoming access tokens in Azure Functions. You can find examples of such validation in the following code samples on GitHub:

- .NET: [Azure-Samples/ms-identity-dotnet-webapi-azurefunctions](https://github.com/Azure-Samples/ms-identity-dotnet-webapi-azurefunctions)
- Node.js: [Azure-Samples/ms-identity-nodejs-webapi-azurefunctions](https://github.com/Azure-Samples/ms-identity-nodejs-webapi-azurefunctions)
- Python: [Azure-Samples/ms-identity-python-webapi-azurefunctions](https://github.com/Azure-Samples/ms-identity-python-webapi-azurefunctions)

Next steps

Move on to the next article in this scenario, [Verify scopes and app roles in your code](#).

Protected web API: Verify scopes and app roles

4/12/2022 • 7 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 and roles.
- Daemon apps that have the right application roles.

The code snippets in this article are extracted from the following code samples on GitHub:

- [ASP.NET Core web API incremental tutorial](#)
- [ASP.NET web API sample](#)

To protect an ASP.NET or ASP.NET Core web API, you must add the `[Authorize]` attribute to one of the following items:

- The controller itself if you want all controller actions 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 and ASP.NET Core validate the token. Your API needs to verify that the token used to call the API is requested with the expected claims. These claims in particular need verification:

- 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.

Verify scopes in APIs called on behalf of users

If a client app calls your API on behalf of a user, the API needs to request a bearer token that has specific scopes for the API. For more information, see [Code configuration | Bearer token](#).

- [ASP.NET Core](#)
- [ASP.NET Classic](#)

In ASP.NET Core, you can use `Microsoft.Identity.Web` to verify scopes in each controller action. You can also verify them at the level of the controller or for the whole application.

Verify the scopes on each controller action

You can verify the scopes in the controller action by using the `[RequiredScope]` attribute. This attribute has several overrides. One that takes the required scopes directly, and one that takes a key to the configuration.

Verify the scopes on a controller action with hardcoded scopes

The following code snippet shows the usage of the `[RequiredScope]` attribute with hardcoded scopes.

```

using Microsoft.Identity.Web

[Authorize]
public class TodoListController : Controller
{
    /// <summary>
    /// The web API will accept only tokens that have the `access_as_user` scope for
    /// this API.
    /// </summary>
    const string scopeRequiredByApi = "access_as_user";

    // GET: api/values
    [HttpGet]
    [RequiredScope(scopeRequiredByApi)]
    public IEnumerable<TodoItem> Get()
    {
        // Do the work and return the result.
        // ...
    }
    // ...
}

```

Verify the scopes on a controller action with scopes defined in configuration

You can also declare these required scopes in the configuration, and reference the configuration key:

For instance if, in the appsettings.json you have the following configuration:

```
{
    "AzureAd" : {
        // more settings
        "Scopes" : "access_as_user access_as_admin"
    }
}
```

Then, reference it in the `[RequiredScope]` attribute:

```

using Microsoft.Identity.Web

[Authorize]
public class TodoListController : Controller
{
    // GET: api/values
    [HttpGet]
    [RequiredScope(RequiredScopesConfigurationKey = "AzureAd:Scopes")]
    public IEnumerable<TodoItem> Get()
    {
        // Do the work and return the result.
        // ...
    }
    // ...
}

```

Verify scopes conditionally

There are cases where you want to verify scopes conditionally. You can do this using the `VerifyUserHasAnyAcceptedScope` extension method on the `HttpContext`.

```

using Microsoft.Identity.Web

[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>
    static readonly string[] scopeRequiredByApi = new string[] { "access_as_user" };

    // GET: api/values
    [HttpGet]
    public IEnumerable<TodoItem> Get()
    {
        HttpContext.VerifyUserHasAnyAcceptedScope(scopeRequiredByApi);
        // Do the work and return the result.
        // ...
    }
    // ...
}

```

Verify the scopes at the level of the controller

You can also verify the scopes for the whole controller

Verify the scopes on a controller with hardcoded scopes

The following code snippet shows the usage of the `[RequiredScope]` attribute with hardcoded scopes on the controller.

```

using Microsoft.Identity.Web

[Authorize]
[RequiredScope(scopeRequiredByApi)]
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>
    static readonly string[] scopeRequiredByApi = new string[] { "access_as_user" };

    // GET: api/values
    [HttpGet]
    public IEnumerable<TodoItem> Get()
    {
        // Do the work and return the result.
        // ...
    }
    // ...
}

```

Verify the scopes on a controller with scopes defined in configuration

Like on action, you can also declare these required scopes in the configuration, and reference the configuration key:

```

using Microsoft.Identity.Web

[Authorize]
[RequiredScope(RequiredScopesConfigurationKey = "AzureAd:Scopes")]
public class TodoListController : Controller
{
    // GET: api/values
    [HttpGet]
    public IEnumerable<TodoItem> Get()
    {
        // Do the work and return the result.
        // ...
    }
    // ...
}

```

Verify the scopes more globally

Defining granular scopes for your web API and verifying the scopes in each controller action is the recommended approach. However it's also possible to verify the scopes at the level of the application or a controller. For details, see [Claim-based authorization](#) in the ASP.NET core documentation.

What is verified?

The `[RequiredScope]` attribute and `VerifyUserHasAnyAcceptedScope` method, does something like the following steps:

- Verify there's a claim named `http://schemas.microsoft.com/identity/claims/scope` or `scp`.
- Verify the claim has a value that contains the scope expected by the API.

Verify 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. As shown in [Exposing application permissions \(app roles\)](#), your API exposes such permissions. One example is the `access_as_application` app role.

You now need to have your API verify that the token it receives contains the `roles` claim and that this claim has the expected value. The verification code is similar to the code that verifies delegated permissions, except that your controller action tests for roles instead of scopes:

- [ASP.NET Core](#)
- [ASP.NET Classic](#)

The following code snippet shows how to verify the application role

```

using Microsoft.Identity.Web

[Authorize]
public class TodoListController : ApiController
{
    public IEnumerable<TodoItem> Get()
    {
        HttpContext.ValidateAppRole("access_as_application");
        // ...
    }
}

```

Instead, you can use the `[Authorize(Roles = "access_as_application")]` attributes on the controller or an action (or a razor page).

```
[Authorize(Roles = "access_as_application")]
MyController : ApiController
{
    // ...
}
```

[Role-based authorization in ASP.NET Core](#) lists several approaches to implement role based authorization. Developers can choose one among them which suits to their respective scenarios.

For working samples, see the web app incremental tutorial on [authorization by roles and groups](#).

Verify app roles in APIs called on behalf of users

Users can also use roles claims in user assignment patterns, as shown in [How to add app roles in your application and receive them in the token](#). If the roles are assignable to both, checking roles will let apps sign in as users and users sign in as apps. We recommend that you declare different roles for users and apps to prevent this confusion.

If you have defined app roles with user/group, then roles claim can also be verified in the API along with scopes. The verification logic of the app roles in this scenario remains same as if API is called by the daemon apps since there is no differentiation in the role claim for user/group and application.

Accepting app-only tokens if the web API should be called only by daemon apps

If you want only daemon apps to call your web API, add the condition that the token is an app-only token when you validate the app role.

```
string oid = ClaimsPrincipal.Current.FindFirst("oid")?.Value;
string sub = ClaimsPrincipal.Current.FindFirst("sub")?.Value;
bool isAppOnly = oid != null && sub != null && oid == sub;
```

Checking the inverse condition allows only apps that sign in a user to call your API.

Using ACL-based authorization

Alternatively to app-roles based authorization, you can protect your web API with an Access Control List (ACL) based authorization pattern to [control tokens without the roles claim](#).

If you are using Microsoft.Identity.Web on ASP.NET core, you'll need to declare that you are using ACL-based authorization, otherwise Microsoft Identity Web will throw an exception when neither roles nor scopes are in the Claims provided:

```
System.UnauthorizedAccessException: IDW10201: Neither scope or roles claim was found in the bearer token.
```

To avoid this exception, set the `AllowWebApiToBeAuthorizedByACL` configuration property to true, in the `appsettings.json` or programmatically.

```
{
  "AzureAD": {
    // other properties
    "AllowWebApiToBeAuthorizedByACL" : true,
    // other properties
  }
}
```

If you set `AllowWebApiToBeAuthorizedByACL` to true, this is **your responsibility** to ensure the ACL mechanism.

Next steps

Move on to the next article in this scenario, [Move to production](#).

Protected web API - move to production

4/12/2022 • 2 minutes to read • [Edit Online](#)

Now that you know how to protect your web API, here are some things to consider when moving your application to production.

Enable logging

To help in debugging and authentication failure troubleshooting scenarios, the Microsoft Authentication Library provides built-in logging support. Logging is each library is covered in the following articles:

- [Logging in MSAL.NET](#)
- [Logging in MSAL for Android](#)
- [Logging in MSAL.js](#)
- [Logging in MSAL for iOS/macOS](#)
- [Logging in MSAL for Java](#)
- [Logging in MSAL for Python](#)

Here are some suggestions for data collection:

- Users might ask for help when they have problems. A best practice is to capture and temporarily store logs. Provide a location where users can upload the logs. MSAL provides logging extensions to capture detailed information about authentication.
- If telemetry is available, enable it through MSAL to gather data about how users sign in to your app.

Validate your integration

Test your integration by following the [Microsoft identity platform integration checklist](#).

Build for resilience

Learn how to increase resiliency in your app. For details, see [Increase resilience of authentication and authorization applications you develop](#)

Next steps

Learn how to call a downstream API in [Scenario: A web API that calls web APIs](#).

Learn more with tutorials and samples on GitHub:

- [Calling a protected API using a daemon](#)
- [ASP.NET Core web API tutorial](#)
- [ASP.NET web API sample](#)

Scenario: A web API that calls web APIs

4/12/2022 • 2 minutes to read • [Edit Online](#)

Learn what you need to know to build a web API that calls web APIs.

Prerequisites

This scenario, in which a protected web API calls other web APIs, builds on [Scenario: Protected web API](#).

Overview

- A web, desktop, mobile, or single-page application client (not represented in the accompanying diagram) calls a protected web API and provides a JSON Web Token (JWT) bearer token in its "Authorization" HTTP header.
- The protected web API validates the token and uses the Microsoft Authentication Library (MSAL) `AcquireTokenOnBehalfOf` method to request another token from Azure Active Directory (Azure AD) so that the protected web API can call a second web API, or downstream web API, on behalf of the user.



`AcquireTokenOnBehalfOf` refreshes the token when needed.

Specifics

The app registration part that's related to API permissions is classical. The app configuration involves using the OAuth 2.0 On-Behalf-Of flow to use the JWT bearer token for obtaining a second token for a downstream API. The second token in this case is added to the token cache, where it's available in the web API's controllers. This second token can be used to acquire an access token silently to call downstream APIs whenever required.

Next steps

Move on to the next article in this scenario, [App registration](#).

A web API that calls web APIs: App registration

4/12/2022 • 2 minutes to read • [Edit Online](#)

A web API that calls downstream web APIs has the same registration as a protected web API. Follow the instructions in [Protected web API: App registration](#).

Because the web app now calls web APIs, it becomes a confidential client application. That's why extra registration information is required: the app needs to share secrets (client credentials) with the Microsoft identity platform.

Add a client secret or certificate

As with any confidential client application, you need to add a secret or certificate to act as that application's *credentials* so it can authenticate as itself, without user interaction.

You can add credentials to your client app's registration by using the [Azure portal](#) or by using a command-line tool like [PowerShell](#).

Add client credentials by using the Azure portal

To add credentials to your confidential client application's app registration, follow the steps in [Quickstart: Register an application with the Microsoft identity platform](#) for the type of credential you want to add:

- [Add a client secret](#)
- [Add a certificate](#)

Add client credentials by using PowerShell

Alternatively, you can add credentials when you register your application with the Microsoft identity platform by using PowerShell.

The [active-directory-dotnetcore-daemon-v2](#) code sample on GitHub shows how to add an application secret or certificate when registering an application:

- For details on how to add a **client secret** with PowerShell, see [AppCreationScripts/Configure.ps1](#).
- For details on how to add a **certificate** with PowerShell, see [AppCreationScripts-withCert/Configure.ps1](#).

API permissions

Web apps call APIs on behalf of users for whom the bearer token was received. The web apps need to request delegated permissions. For more information, see [Add permissions to access your web API](#).

Next steps

Move on to the next article in this scenario, [App Code configuration](#).

A web API that calls web APIs: Code configuration

4/12/2022 • 5 minutes to read • [Edit Online](#)

After you've registered your web API, you can configure the code for the application.

The code that you use to configure your web API so that it calls downstream web APIs builds on top of the code that's used to protect a web API. For more information, see [Protected web API: App configuration](#).

- [ASP.NET Core](#)
- [Java](#)
- [Python](#)

Microsoft.Identity.Web

Microsoft recommends that you use the [Microsoft.Identity.Web](#) NuGet package when developing an ASP.NET Core protected API calling downstream web APIs. See [Protected web API: Code configuration | Microsoft.Identity.Web](#) for a quick presentation of that library in the context of a web API.

Client secrets or client certificates

Given that your web API now calls a downstream web API, provide a client secret or client certificate in the `appsettings.json` file. You can also add a section that specifies:

- The URL of the downstream web API
- The scopes required for calling the API

In the following example, the `GraphBeta` section specifies these settings.

```
{  
  "AzureAd": {  
    "Instance": "https://login.microsoftonline.com/",  
    "ClientId": "[Client_id-of-web-api-eg-2ec40e65-ba09-4853-bcde-bcb60029e596]",  
    "TenantId": "common"  
  
    // To call an API  
    "ClientSecret": "[Copy the client secret added to the app from the Azure portal]",  
    "ClientCertificates": [  
    ]  
  },  
  "GraphBeta": {  
    "BaseUrl": "https://graph.microsoft.com/beta",  
    "Scopes": "user.read"  
  }  
}
```

Instead of a client secret, you can provide a client certificate. The following code snippet shows using a certificate stored in Azure Key Vault.

```
{
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "ClientId": "[Client_id-of-web-api-eg-2ec40e65-ba09-4853-bcde-bcb60029e596]",
    "TenantId": "common"

    // To call an API
    "ClientCertificates": [
      {
        "SourceType": "KeyVault",
        "KeyVaultUrl": "https://msidentitywebsamples.vault.azure.net",
        "KeyVaultCertificateName": "MicrosoftIdentitySamplesCert"
      }
    ]
  },
  "GraphBeta": {
    "BaseUrl": "https://graph.microsoft.com/beta",
    "Scopes": "user.read"
  }
}
```

`Microsoft.Identity.Web` provides several ways to describe certificates, both by configuration or by code. For details, see [Microsoft.Identity.Web wiki - Using certificates](#) on GitHub.

Startup.cs

Your web API will need to acquire a token for the downstream API. You specify it by adding the

`.EnableTokenAcquisitionToCallDownstreamApi()` line after `.AddMicrosoftIdentityWebApi(Configuration)`. This line exposes the `ITokenAcquisition` service, that you can use in your controller/pages actions. However, as you'll see in the next two bullet points, you can do even simpler. You'll also need to choose a token cache implementation, for example `.AddInMemoryTokenCaches()`, in *Startup.cs*:

```
using Microsoft.Identity.Web;

public class Startup
{
  // ...
  public void ConfigureServices(IServiceCollection services)
  {
    // ...
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
      .AddMicrosoftIdentityWebApi(Configuration, Configuration.GetSection("AzureAd"))
      .EnableTokenAcquisitionToCallDownstreamApi()
      .AddInMemoryTokenCaches();
    // ...
  }
  // ...
}
```

If you don't want to acquire the token yourself, `Microsoft.Identity.Web` provides two mechanisms for calling a downstream web API from another API. The option you choose depends on whether you want to call Microsoft Graph or another API.

Option 1: Call Microsoft Graph

If you want to call Microsoft Graph, `Microsoft.Identity.Web` enables you to directly use the `GraphServiceClient` (exposed by the Microsoft Graph SDK) in your API actions. To expose Microsoft Graph:

1. Add the `Microsoft.Identity.Web.MicrosoftGraph` NuGet package to your project.
2. Add `.AddMicrosoftGraph()` after `.EnableTokenAcquisitionToCallDownstreamApi()` in the *Startup.cs* file.

`.AddMicrosoftGraph()` has several overrides. Using the override that takes a configuration section as a parameter, the code becomes:

```
using Microsoft.Identity.Web;

public class Startup
{
    // ...
    public void ConfigureServices(IServiceCollection services)
    {
        // ...
        services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
            .AddMicrosoftIdentityWebApi(Configuration, Configuration.GetSection("AzureAd"))
            .EnableTokenAcquisitionToCallDownstreamApi()
                .AddMicrosoftGraph(Configuration.GetSection("GraphBeta"))
            .AddInMemoryTokenCaches();
        // ...
    }
    // ...
}
```

Option 2: Call a downstream web API other than Microsoft Graph

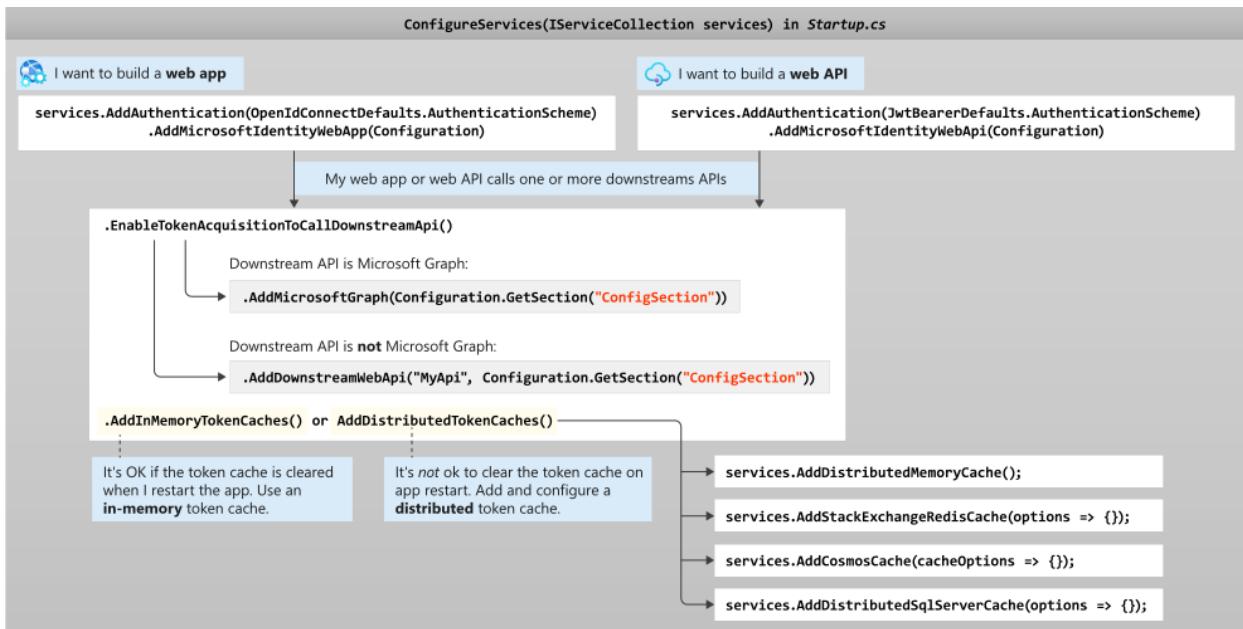
To call a downstream API other than Microsoft Graph, *Microsoft.Identity.Web* provides `.AddDownstreamWebApi()`, which requests tokens and calls the downstream web API.

```
using Microsoft.Identity.Web;

public class Startup
{
    // ...
    public void ConfigureServices(IServiceCollection services)
    {
        // ...
        services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
            .AddMicrosoftIdentityWebApi(Configuration, "AzureAd")
            .EnableTokenAcquisitionToCallDownstreamApi()
                .AddDownstreamWebApi("MyApi", Configuration.GetSection("GraphBeta"))
            .AddInMemoryTokenCaches();
        // ...
    }
    // ...
}
```

As with web apps, you can choose various token cache implementations. For details, see [Microsoft identity web - Token cache serialization](#) on GitHub.

The following image shows the various possibilities of *Microsoft.Identity.Web* and their impact on the *Startup.cs* file:



NOTE

To fully understand the code examples here, be familiar with [ASP.NET Core fundamentals](#), and in particular with [dependency injection](#) and [options](#).

You can also see an example of OBO flow implementation in [Node.js](#) and [Azure Functions](#).

Protocol

For more information about the OBO protocol, see the [Microsoft identity platform and OAuth 2.0 On-Behalf-Of flow](#).

Next steps

Move on to the next article in this scenario, [Acquire a token for the app](#).

A web API that calls web APIs: Acquire a token for the app

4/12/2022 • 3 minutes to read • [Edit Online](#)

After 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

- [ASP.NET Core](#)
- [Java](#)
- [Python](#)

Microsoft.Identity.Web adds extension methods that provide convenience services for calling Microsoft Graph or a downstream web API. These methods are explained in detail in [A web API that calls web APIs: Call an API](#). With these helper methods, you don't need to manually acquire a token.

If, however, you do want to manually acquire a token, the following code shows an example of using *Microsoft.Identity.Web* to do so in an API controller. It calls a downstream API named *todoList*. To get a token to call the downstream API, you inject the `ITokenAcquisition` service by dependency injection in your controller's constructor (or your page constructor if you use Blazor), and you use it in your controller actions, getting a token for the user (`GetAccessTokenForUserAsync`) or for the application itself (`GetAccessTokenForAppAsync`) in the case of a daemon scenario.

```
[Authorize]
public class MyApiController : Controller
{
    /// <summary>
    /// The web API will accept only tokens 1) for users, 2) that have the `access_as_user` scope for
    /// this API.
    /// </summary>
    static readonly string[] scopeRequiredByApi = new string[] { "access_as_user" };

    static readonly string[] scopesToAccessDownstreamApi = new string[] {
        "api://MyTodoListService/access_as_user" };

    private readonly ITokenAcquisition _tokenAcquisition;

    public MyApiController(ITokenAcquisition tokenAcquisition)
    {
        _tokenAcquisition = tokenAcquisition;
    }

    public IActionResult Index()
    {
        HttpContext.VerifyUserHasAnyAcceptedScope(scopeRequiredByApi);

        string accessToken = _tokenAcquisition.GetAccessTokenForUserAsync(scopesToAccessDownstreamApi);
        return await callTodoListService(accessToken);
    }
}
```

For details about the `callTodoListService` method, see [A web API that calls web APIs: Call an API](#).

Next steps

Move on to the next article in this scenario, [Call an API](#).

A web API that calls web APIs: Call an API

4/12/2022 • 3 minutes to read • [Edit Online](#)

After you have a token, you can call a protected web API. You usually call the downstream APIs from the controller or pages of your web API.

Controller code

- [ASP.NET Core](#)
- [Java](#)
- [Python](#)

When you use *Microsoft.Identity.Web*, you have three usage scenarios:

- [Option 1: Call Microsoft Graph with the SDK](#)
- [Option 2: Call a downstream web API with the helper class](#)
- [Option 3: Call a downstream web API without the helper class](#)

Option 1: Call Microsoft Graph with the SDK

In this scenario, you've added `.AddMicrosoftGraph()` in *Startup.cs* as specified in [Code configuration](#), and you can directly inject the `GraphServiceClient` in your controller or page constructor for use in the actions. The following example Razor page displays the photo of the signed-in user.

```
[Authorize]
[AuthorizeForScopes(SCopes = new[] { "user.read" })]
public class IndexModel : PageModel
{
    private readonly GraphServiceClient _graphServiceClient;

    public IndexModel(GraphServiceClient graphServiceClient)
    {
        _graphServiceClient = graphServiceClient;
    }

    public async Task OnGet()
    {
        var user = await _graphServiceClient.Me.Request().GetAsync();
        try
        {
            using (var photoStream = await _graphServiceClient.Me.Photo.Content.Request().GetAsync())
            {
                byte[] photoByte = ((MemoryStream)photoStream).ToArray();
                ViewData["photo"] = Convert.ToBase64String(photoByte);
            }
            ViewData["name"] = user.DisplayName;
        }
        catch (Exception)
        {
            ViewData["photo"] = null;
        }
    }
}
```

Option 2: Call a downstream web API with the helper class

In this scenario, you've added `.AddDownstreamWebApi()` in *Startup.cs* as specified in [Code configuration](#), and you

can directly inject an `IDownstreamWebApi` service in your controller or page constructor and use it in the actions:

```
[Authorize]
[AuthorizeForScopes(ScopeKeySection = "TodoList:Scopes")]
public class TodoListController : Controller
{
    private IDownstreamWebApi _downstreamWebApi;
    private const string ServiceName = "TodoList";

    public TodoListController(IDownstreamWebApi downstreamWebApi)
    {
        _downstreamWebApi = downstreamWebApi;
    }

    public async Task<ActionResult> Details(int id)
    {
        var value = await _downstreamWebApi.CallWebApiForUserAsync(
            ServiceName,
            options =>
            {
                options.RelativePath = $"me";
            });
        return View(value);
    }
}
```

The `CallWebApiForUserAsync` method also has strongly typed generic overrides that enable you to directly receive an object. For example, the following method received a `Todo` instance, which is a strongly typed representation of the JSON returned by the web API.

```
// GET: TodoList/Details/5
public async Task<ActionResult> Details(int id)
{
    var value = await _downstreamWebApi.CallWebApiForUserAsync<object, Todo>(
        ServiceName,
        null,
        options =>
        {
            options.HttpMethod = HttpMethod.Get;
            options.RelativePath = $"api/todolist/{id}";
        });
    return View(value);
}
```

Option 3: Call a downstream web API without the helper class

If you've decided to acquire a token manually by using the `ITokenAcquisition` service, you now need to use the token. In that case, the following code continues the example code shown in [A web API that calls web APIs: Acquire a token for the app](#). The code is called in the actions of the API controllers. It calls a downstream API named *todolist*.

After you've acquired the token, use it as a bearer token to call the downstream API.

```
private async Task CallTodoListService(string accessToken)
{
    // After the token has been returned by Microsoft.Identity.Web, add it to the HTTP authorization header
    // before making the call to access the todolist service.
    _httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", accessToken);

    // Call the todolist service.
    HttpResponseMessage response = await _httpClient.GetAsync(TodoListBaseAddress + "/api/todolist");
    // ...
}
```

Next steps

Move on to the next article in this scenario, [Move to production](#).

A web API that calls web APIs: Move to production

4/12/2022 • 2 minutes to read • [Edit Online](#)

After you've acquired a token to call web APIs, here are some things to consider when moving your application to production.

Enable logging

To help in debugging and authentication failure troubleshooting scenarios, the Microsoft Authentication Library provides built-in logging support. Logging for each library is covered in the following articles:

- [Logging in MSAL.NET](#)
- [Logging in MSAL for Android](#)
- [Logging in MSAL.js](#)
- [Logging in MSAL for iOS/macOS](#)
- [Logging in MSAL for Java](#)
- [Logging in MSAL for Python](#)

Here are some suggestions for data collection:

- Users might ask for help when they have problems. A best practice is to capture and temporarily store logs. Provide a location where users can upload the logs. MSAL provides logging extensions to capture detailed information about authentication.
- If telemetry is available, enable it through MSAL to gather data about how users sign in to your app.

Validate your integration

Test your integration by following the [Microsoft identity platform integration checklist](#).

Build for resilience

Learn how to increase resiliency in your app. For details, see [Increase resilience of authentication and authorization applications you develop](#)

Next steps

Now that you know the basics of how to call web APIs from your own web API, you might be interested in the following tutorial, which describes the code that's used to build a protected web API that calls web APIs.

SAMPLE	PLATFORM	DESCRIPTION
active-directory-aspnetcore-webapi-tutorial-v2 chapter 1	ASP.NET Core web API, Desktop (WPF)	ASP.NET Core web API calls Microsoft Graph, which you call from a WPF application by using the Microsoft identity platform.

Scopes for a web API accepting v1.0 tokens

4/12/2022 • 2 minutes to read • [Edit Online](#)

OAuth2 permissions are permission scopes that a Azure Active Directory (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

To acquire tokens for specific scopes of a v1.0 application (for example the Microsoft Graph API, which is <https://graph.microsoft.com>), 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"];
```

To read and write with MSAL.NET Azure AD using the Microsoft Graph API (<https://graph.microsoft.com/>), create a list of scopes as shown in the following examples:

```
string ResourceId = "https://graph.microsoft.com/";
var scopes = new [] { ResourceId + "Directory.Read", ResourceID + "Directory.Write"}
```

```
var ResourceId = "https://graph.microsoft.com/";
var scopes = [ ResourceId + "Directory.Read", ResourceID + "Directory.Write"];
```

To write the scope corresponding to the Azure Resource Manager API (<https://management.core.windows.net/>), 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

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 (Azure AD v1.0) endpoint with a v1.0 access token (the only possible), aud=resource
- For MSAL (Microsoft identity platform) 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 that accepts 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 a client credential flow/daemon app

For the standard client credentials flow, use `/.default`. For example, `https://graph.microsoft.com/.default`.

Azure AD will automatically include all the app-level permissions the admin has consented to in the access token for the client credentials flow.

Quickstart: Acquire a token and call Microsoft Graph API from a desktop application

4/12/2022 • 15 minutes to read • [Edit Online](#)

In this quickstart, you download and run a code sample that demonstrates how a Universal Windows Platform (UWP) application can sign in users and get an access token to call the Microsoft Graph API.

See [How the sample works](#) for an illustration.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Visual Studio](#)

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 [Azure portal - App registrations](#) quickstart experience.
2. Enter a name for your application and select **Register**.
3. 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, follow these steps:

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application, for example `UWP-App-calling-MsGraph`. Users of your app might see this name, and you can change it later.
6. In the **Supported account types** section, select **Accounts in any organizational directory and personal Microsoft accounts (for example, Skype, Xbox, Outlook.com)**.
7. Select **Register** to create the application, and then record the **Application (client) ID** for use in a later step.
8. Under **Manage**, select **Authentication**.
9. Select **Add a platform > Mobile and desktop applications**.
10. Under **Redirect URIs**, select `https://login.microsoftonline.com/common/oauth2/nativeclient`.
11. Select **Configure**.

Step 2: Download the project

[Download the UWP sample application](#)

TIP

To avoid errors caused by path length limitations in Windows, we recommend extracting the archive or cloning the repository into a directory near the root of your drive.

Step 3: Configure the project

1. Extract the .zip archive to a local folder close to the root of your drive. For example, into C:\Azure-Samples.
2. Open the project in Visual Studio. Install the **Universal Windows Platform development** workload and any individual SDK components if prompted.
3. In *MainPage.Xaml.cs*, change the value of the `ClientId` variable to the **Application (Client) ID** of the application you registered earlier.

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

You can find the **Application (client) ID** on the app's **Overview** pane in the Azure portal ([Azure Active Directory > App registrations > {Your app registration}](#)).

4. Create and then select a new self-signed test certificate for the package:
 - a. In the **Solution Explorer**, double-click the *Package.appxmanifest* file.
 - b. Select **Packaging > Choose Certificate... > Create....**
 - c. Enter a password and then select **OK**. A certificate called *Native_UWP_V2_TemporaryKey.pfx* is created.
 - d. Select **OK** to dismiss the **Choose a certificate** dialog, and then verify that you see *Native_UWP_V2_TemporaryKey.pfx* in Solution Explorer.
 - e. In the **Solution Explorer**, right-click the **Native_UWP_V2** project and select **Properties**.
 - f. Select **Signing**, and then select the .pfx you created in the **Choose a strong name key file** dropdown.

Step 4: Run the application

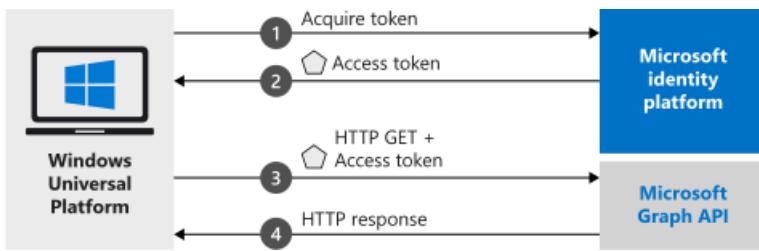
To run the sample application on your local machine:

1. In the Visual Studio toolbar, choose the right platform (probably **x64** or **x86**, not ARM). The target device should change from *Device* to *Local Machine*.
2. Select **Debug > Start Without Debugging**.

If you're prompted to do so, you might first need to enable **Developer Mode**, and then **Start Without Debugging** again to launch the app.

When the app's window appears, you can select the **Call Microsoft Graph API** button, enter your credentials, and consent to the permissions requested by the application. If successful, the application displays some token information and data obtained from the call to the Microsoft Graph API.

How the sample works



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 the Microsoft Identity platform. You can install MSAL by running the following command in Visual Studio's *Package Manager Console*.

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

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 = PublicClientApplicationBuilder.Create(ClientId)
    .WithRedirectUri("https://login.microsoftonline.com/common/oauth2/nativeclient")
    .Build();
```

The value of `ClientId` is the **Application (client) ID** of the app you 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 through a pop-up 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();
```

The `scopes` parameter contains the scopes being requested, such as `{ "user.read" }` for Microsoft Graph or `{ "api://<Application ID>/access_as_user" }` 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();
```

- `scopes` contains the scopes being requested, such as `{ "user.read" }` for Microsoft Graph or `{ "api://<Application ID>/access_as_user" }` for custom web APIs.
- `firstAccount` 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 about your support options, see [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](#)

In this quickstart, you download and run a code sample that demonstrates how a Windows Presentation Foundation (WPF) application can sign in users and get an access token to call the Microsoft Graph API.

See [How the sample works](#) for an illustration.

Prerequisites

- Visual Studio with the [Universal Windows Platform development](#) workload installed

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 [Azure portal - App registrations](#) quickstart experience.
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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application, for example `Win-App-calling-MsGraph`. Users of your app might see this

name, and you can change it later.

6. In the **Supported account types** section, select **Accounts in any organizational directory and personal Microsoft accounts (for example, Skype, Xbox, Outlook.com)**.
7. Select **Register** to create the application.
8. Under **Manage**, select **Authentication**.
9. Select **Add a platform > Mobile and desktop applications**.
10. In the **Redirect URIs** section, select `https://login.microsoftonline.com/common/oauth2/nativeclient` and in **Custom redirect URIs** add `ms-appx-web://microsoft.aad.brokerplugin/{client_id}` where `{client_id}` is the application (client) ID of your application (the same GUID that appears in the `msal{client_id}://auth` checkbox).
11. Select **Configure**.

Step 2: Download the project

[Download the WPF sample application](#)

TIP

To avoid errors caused by path length limitations in Windows, we recommend extracting the archive or cloning the repository into a directory near the root of your drive.

Step 3: Configure the 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.

To find the value of **Application (client) ID**, go to the app's **Overview** page in the Azure portal.

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

Step 4: Run the application

To build and run the sample application in Visual Studio, select the **Debug menu > Start Debugging**, or press the F5 key. Your application's `MainWindow` is displayed.

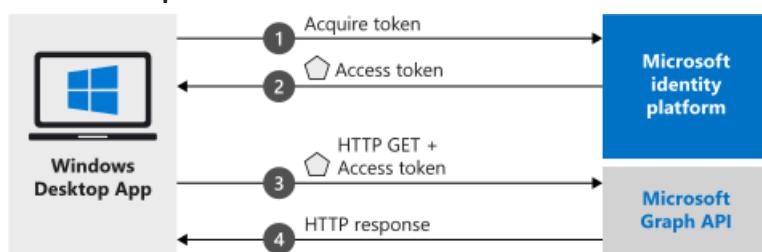
When the app's main window appears, select the **Call Microsoft Graph API** button. You'll be prompted to sign in using your Azure Active Directory account (work or school account) or Microsoft account (live.com,

outlook.com) credentials.

If you're running the application for the first time, you'll be prompted to provide consent to allow the application to access your user profile and sign you in. After consenting to the requested permissions, the application displays that you've successfully logged in. You should see some basic token information and user data obtained from the call to the Microsoft Graph API.

More information

How the sample works



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:

```
IPublicClientApplication publicClientApp = PublicClientApplicationBuilder.Create(ClientId)
    .WithRedirectUri("https://login.microsoftonline.com/common/oauth2/nativeclient")
    .WithAuthority(AzureCloudInstance.AzurePublic, Tenant)
    .Build();
```

WHERE:	DESCRIPTION
<code>ClientId</code>	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 interact with the Microsoft identity platform through a pop-up 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:	DESCRIPTION
_scopes	Contains the scopes being requested, such as { "user.read" } for Microsoft Graph or { "api://<Application ID>/access_as_user" } 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:	DESCRIPTION
scopes	Contains the scopes being requested, such as { "user.read" } for Microsoft Graph or { "api://<Application ID>/access_as_user" } for custom web APIs.
firstAccount	Specifies the first user in the cache (MSAL support multiple users in a single app).

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [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](#)

In this quickstart, you download and run a code sample that demonstrates how an Electron desktop application can sign in users and acquire access tokens to call the Microsoft Graph API.

This quickstart uses the [Microsoft Authentication Library for Node.js \(MSAL Node\)](#) with the [authorization code flow with PKCE](#).

Prerequisites

- [Node.js](#)

- [Visual Studio Code](#) or another code editor

Register and download the sample application

Follow the steps below to get started.

Step 1: Register the 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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application, for example `msal-node-desktop`. Users of your app might see this name, and you can change it later.
6. Select **Register** to create the application.
7. Under **Manage**, select **Authentication**.
8. Select **Add a platform > Mobile and desktop applications**.
9. In the **Redirect URIs** section, enter `msal://redirect`.
10. Select **Configure**.

Step 2: Download the Electron sample project

[Download the code sample](#)

Step 3: Configure the Electron sample project

1. Extract the zip file to a local folder close to the root of the disk, for example, *C:/Azure-Samples*.

2. Edit `.env` and replace the values of the fields `TENANT_ID` and `CLIENT_ID` with the following snippet:

```
"TENANT_ID": "Enter_the_Tenant_Id_Here",
"CLIENT_ID": "Enter_the_Application_Id_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)

TIP

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

Step 4: Run the application

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

```
npm install
```

Then, run the application via command prompt or console:

```
npm start
```

You should see application's UI with a **Sign in** button.

About the code

Below, some of the important aspects of the sample application are discussed.

MSAL Node

MSAL Node is the library used to sign in users and request tokens used to access an API protected by Microsoft identity platform. For more information on how to use MSAL Node with desktop apps, see [this article](#).

You can install MSAL Node by running the following npm command.

```
npm install @azure/msal-node --save
```

MSAL initialization

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

```
const { PublicClientApplication } = require('@azure/msal-node');
```

Then, initialize MSAL using the following code:

```
const MSAL_CONFIG = {
    auth: {
        clientId: "Enter_the_Application_Id_Here",
        authority: "https://login.microsoftonline.com/Enter_the_Tenant_Id_Here",
    },
};

const pca = new PublicClientApplication(MSAL_CONFIG);
```

WHERE:	DESCRIPTION
<code>clientId</code>	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.
<code>authority</code>	The STS endpoint for user to authenticate. Usually <code>https://login.microsoftonline.com/{tenant}</code> for public cloud, where {tenant} is the name of your tenant or your tenant Id.

Requesting tokens

In the first leg of authorization code flow with PKCE, prepare and send an authorization code request with the appropriate parameters. Then, in the second leg of the flow, listen for the authorization code response. Once the code is obtained, exchange it to obtain a token.

```
// The redirect URI you setup during app registration with a custom file protocol "msal"
const redirectUri = "msal://redirect";

const cryptoProvider = new CryptoProvider();
```

```

const pkceCodes = {
    challengeMethod: "S256", // Use SHA256 Algorithm
    verifier: "", // Generate a code verifier for the Auth Code Request first
    challenge: "" // Generate a code challenge from the previously generated code verifier
};

/**
 * Starts an interactive token request
 * @param {object} authWindow: Electron window object
 * @param {object} tokenRequest: token request object with scopes
 */
async function getTokenInteractive(authWindow, tokenRequest) {

    /**
     * Proof Key for Code Exchange (PKCE) Setup
     *
     * MSAL enables PKCE in the Authorization Code Grant Flow by including the codeChallenge and
     * codeChallengeMethod
     * parameters in the request passed into getAuthCodeUrl() API, as well as the codeVerifier parameter in
     * the
     * second leg (acquireTokenByCode() API).
     */

    const {verifier, challenge} = await cryptoProvider.generatePkceCodes();

    pkceCodes.verifier = verifier;
    pkceCodes.challenge = challenge;

    const authCodeUrlParams = {
        redirectUri: redirectUri
        scopes: tokenRequest.scopes,
        codeChallenge: pkceCodes.challenge, // PKCE Code Challenge
        codeChallengeMethod: pkceCodes.codeChallengeMethod // PKCE Code Challenge Method
    };

    const authCodeUrl = await pca.getAuthCodeUrl(authCodeUrlParams);

    // register the custom file protocol in redirect URI
    protocol.registerFileProtocol(redirectUri.split(":")[0], (req, callback) => {
        const requestUrl = url.parse(req.url, true);
        callback(path.normalize(`$__dirname__/${requestUrl.path}`));
    });

    const authCode = await listenForAuthCode(authCodeUrl, authWindow); // see below

    const authResponse = await pca.acquireTokenByCode({
        redirectUri: redirectUri,
        scopes: tokenRequest.scopes,
        code: authCode,
        codeVerifier: pkceCodes.verifier // PKCE Code Verifier
    });

    return authResponse;
}

/**
 * Listens for auth code response from Azure AD
 * @param {string} navigateUrl: URL where auth code response is parsed
 * @param {object} authWindow: Electron window object
 */
async function listenForAuthCode(navigateUrl, authWindow) {

    authWindow.loadURL(navigateUrl);

    return new Promise((resolve, reject) => {
        authWindow.webContents.on('will-redirect', (event, responseUrl) => {
            try {
                const parsedUrl = new URL(responseUrl);
                const authCode = parsedUrl.searchParams.get('code');
            }
        })
    })
}

```

```
        resolve(authCode);
    } catch (err) {
        reject(err);
    }
});
});
}
}
```

WHERE:	DESCRIPTION
<code>authWindow</code>	Current Electron window in process.
<code>tokenRequest</code>	Contains the scopes being requested, such as <code>"User.Read"</code> for Microsoft Graph or <code>"api://<Application ID>/access_as_user"</code> for custom web APIs.

Next steps

To learn more about Electron desktop app development with MSAL Node, see the tutorial:

[Tutorial: Sign in users and call the Microsoft Graph API in an Electron desktop app](#)

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

4/12/2022 • 13 minutes to read • [Edit Online](#)

In this tutorial, you build a native Universal Windows Platform (UWP) app that signs in users and gets an access token to call the Microsoft Graph API.

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).

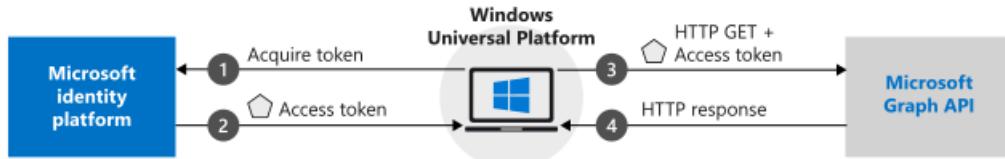
In this tutorial:

- Create a *Universal Windows Platform (UWP)* project in Visual Studio
- Register the application in the Azure portal
- Add code to support user sign-in and sign-out
- Add code to call Microsoft Graph API
- Test the app

Prerequisites

- [Visual Studio 2019](#) with the [Universal Windows Platform development](#) workload installed

How this guide works



This guide creates a sample UWP application that queries the Microsoft Graph API. For this scenario, a token is added to HTTP requests by using the Authorization header. The Microsoft Authentication Library handles token acquisitions and renewals.

NuGet packages

This guide uses the following NuGet package:

LIBRARY	DESCRIPTION
Microsoft.Identity.Client	Microsoft Authentication Library
Microsoft.Graph	Microsoft Graph Client 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 the Microsoft Graph API.

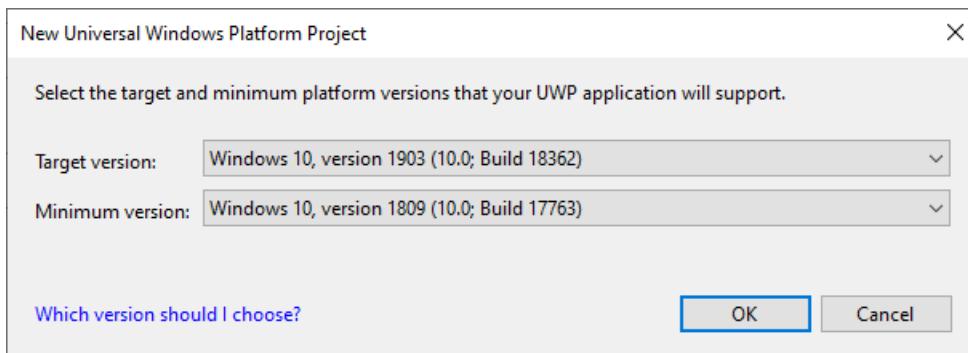
This guide creates an application that displays a button that queries the Microsoft Graph API and a button to sign out. It also displays text boxes that contain the results of the calls.

TIP

To see a completed version of the project you build in this tutorial, you can [download it from GitHub](#).

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 the Microsoft Authentication Library to your project

1. In Visual Studio, select **Tools > NuGet Package Manager > Package Manager Console**.
2. Copy and paste the following commands in the **Package Manager Console** window:

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

NOTE

The first command installs the [Microsoft Authentication Library \(MSAL.NET\)](#). MSAL.NET acquires, caches, and refreshes user tokens that access APIs that are protected by the Microsoft identity platform. The second command installs [Microsoft Graph .NET Client Library](#) to authenticate requests to Microsoft Graph and make calls to the service.

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 the Microsoft Authentication Library to get a token for the Microsoft Graph API

This section shows how to use the Microsoft Authentication Library to get a token for the 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 Microsoft.Graph;
using System.Diagnostics;
using System.Threading.Tasks;
using System.Net.Http.Headers;

```

2. Replace your `MainPage` class with the following code:

```

public sealed partial class MainPage : Page
{
    //Set the scope for API call to user.read
    private 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
    private const string ClientId = "[Application Id pasted from the application registration portal]";

    private const string Tenant = "common"; // Alternatively "[Enter your tenant, as obtained from the Azure portal, e.g. kko365.onmicrosoft.com]"
    private const string Authority = "https://login.microsoftonline.com/" + Tenant;

    // The MSAL Public client app
    private static IPublicClientApplication PublicClientApp;

    private static string MSGraphURL = "https://graph.microsoft.com/v1.0/";
    private static AuthenticationResult authResult;

    public MainPage()
    {
        this.InitializeComponent();
    }

    /// <summary>
    /// Call AcquireTokenAsync - to acquire a token requiring user to sign in
    /// </summary>
    private async void CallGraphButton_Click(object sender, RoutedEventArgs e)
    {
        try

```

```

{
    // Sign in user using MSAL and obtain an access token for Microsoft Graph
    GraphServiceClient graphClient = await SignInAndInitializeGraphServiceClient(scopes);

    // Call the /me endpoint of Graph
    User graphUser = await graphClient.Me.Request().GetAsync();

    // Go back to the UI thread to make changes to the UI
    await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
    {
        ResultText.Text = "Display Name: " + graphUser.DisplayName + "\nBusiness Phone: " +
graphUser.BusinessPhones.FirstOrDefault()
            + "\nGiven Name: " + graphUser.GivenName + "\nid: " + graphUser.Id
            + "\nUser Principal Name: " + graphUser.UserPrincipalName;
        DisplayBasicTokenInfo(authResult);
        this.SignOutButton.Visibility = Visibility.Visible;
    });
}

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;
}
}

/// <summary>
/// Signs in the user and obtains an access token for Microsoft Graph
/// </summary>
/// <param name="scopes"></param>
/// <returns> Access Token</returns>
private static async Task<string> SignInUserAndGetTokenUsingMSAL(string[] scopes)
{
    // Initialize the MSAL library by building a public client application
    PublicClientApp = PublicClientApplicationBuilder.Create(ClientId)
        .WithAuthority(Authority)
        .WithUseCorporateNetwork(false)
        .WithRedirectUri("https://login.microsoftonline.com/common/oauth2/nativeclient")
        .WithLogging((level, message, containsPii) =>
    {
        Debug.WriteLine($"MSAL: {level} {message} ");
    }, LogLevel.Warning, enablePiiLogging: false, enableDefaultPlatformLogging: true)
        .Build();

    // 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 AcquireTokenSilentAsync. This indicates you need
to call AcquireTokenAsync to acquire a token
        Debug.WriteLine($"MsalUiRequiredException: {ex.Message}");

        authResult = await PublicClientApp.AcquireTokenInteractive(scopes)
            .ExecuteAsync()
            .ConfigureAwait(false);
    }
}

```

```
        return authResult.AccessToken;
    }
}
```

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. The Microsoft Authentication Library 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 the Microsoft Authentication Library 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.

Instantiate the Microsoft Graph Service Client by obtaining the token from the `SignInUserAndGetTokenUsingMSAL` method

Add the following new method to `MainPage.xaml.cs`:

```
/// <summary>
/// Sign in user using MSAL and obtain a token for Microsoft Graph
/// </summary>
/// <returns>GraphServiceClient</returns>
private async static Task<GraphServiceClient> SignInAndInitializeGraphServiceClient(string[] scopes)
{
    GraphServiceClient graphClient = new GraphServiceClient(MSGraphURL,
        new DelegateAuthenticationProvider(async (requestMessage) =>
    {
        requestMessage.Headers.Authorization = new AuthenticationHeaderValue("bearer", await
SignInUserAndGetTokenUsingMSAL(scopes));
    }));
    return await Task.FromResult(graphClient);
}
```

[More information on making a REST call against a protected API](#)

In this sample application, the `GetGraphServiceClient` method instantiates `GraphServiceClient` by using an access token. Then, `GraphServiceClient` is used to get the user's profile information from the `me` endpoint.

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}";
    }
}
```

MSAL.NET uses asynchronous methods to acquire tokens or manipulate accounts. As such, 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 Microsoft Authentication Library user cache. This method effectively tells the Microsoft Authentication Library 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. The Microsoft Authentication Library 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 the Microsoft Authentication Library 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, register your application:

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application, for example `UWP-App-calling-MSGraph`. Users of your app might see this name, and you can change it later.
6. Under **Supported account types**, select **Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**.
7. Select **Register**.
8. 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 > Add a platform**, and then select **Mobile and desktop applications**.
2. In the **Redirect URIs** section, enter `https://login.microsoftonline.com/common/oauth2/nativeclient`.
3. Select **Configure**.

Configure API permissions for your application:

1. Under **Manage**, select **API permissions > Add a permission**.
2. Select **Microsoft Graph**.
3. Select **Delegated permissions**, search for *User.Read*, and verify that **User.Read** is selected.
4. If you made any changes, select **Add permissions** to save them.

Enable integrated authentication on federated domains (optional)

To enable integrated Windows 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 isn't 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 integrated Windows authentication with a federated Azure AD domain.

Alternate approach to using WithDefaultRedirectURI()

In the current sample, the `WithRedirectUri("https://login.microsoftonline.com/common/oauth2/nativeclient")` method is used. To use `WithDefaultRedirectURI()`, complete these steps:

1. In *MainPage.Xaml.cs*, replace `WithRedirectUri` with `WithDefaultRedirectUri`:

Current code

```
PublicClientApp = PublicClientApplicationBuilder.Create(ClientId)
    .WithAuthority(Authority)
    .WithUseCorporateNetwork(false)
    .WithRedirectUri("https://login.microsoftonline.com/common/oauth2/nativeclient")
    .WithLogging((level, message, containsPii) =>
    {
        Debug.WriteLine($"MSAL: {level} {message}");
    }, LogLevel.Warning, enablePiiLogging: false, enableDefaultPlatformLogging: true)
    .Build();
```

Updated code

```
PublicClientApp = PublicClientApplicationBuilder.Create(ClientId)
    .WithAuthority("https://login.microsoftonline.com/common")
    .WithUseCorporateNetwork(false)
    .WithDefaultRedirectUri()
    .WithLogging((level, message, containsPii) =>
    {
        Debug.WriteLine($"MSAL: {level} {message}");
    }, LogLevel.Warning, enablePiiLogging: false, enableDefaultPlatformLogging: true)
    .Build();
```

2. Find the callback URI for your app by adding the `redirectURI` field in *MainPage.xaml.cs* and setting a breakpoint on it:

```
public sealed partial class MainPage : Page
{
    ...
    string redirectURI = Windows.Security.Authentication.Web.WebAuthenticationBroker
        .GetCurrentApplicationCallbackUri().ToString();
    public MainPage()
    {
        ...
    }
    ...
}
```

Run the app, and then copy the value of `redirectUri` when the breakpoint is hit. The value should look something similar to the following value:

```
ms-app://s-1-15-2-1352796503-54529114-405753024-3540103335-3203256200-511895534-1429095407/
```

You can then remove the line of code because it's required only once, to fetch the value.

3. In the app registration portal, add the returned value in **RedirectUri** in the **Authentication** pane.

Test your code

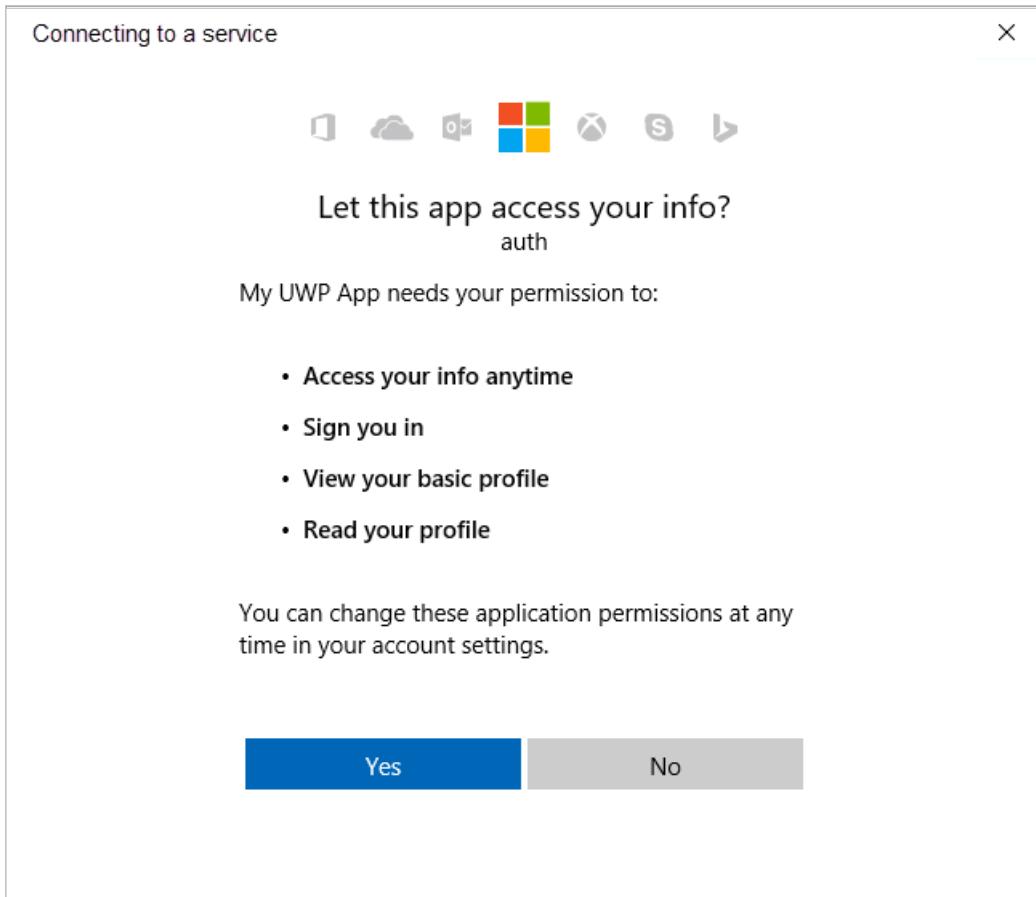
To test your application, select the F5 key to run your project in Visual Studio. Your main window appears:



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 test, the application displays a window asking the user to sign in.

Consent

The first time you sign in to your application, a consent screen appears similar to the following image. 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 a UWP application window titled "active-directory-dotnet-native-uwp-v2". The top right has standard window controls. Below them are two buttons: "Call Microsoft Graph API" and "Sign-Out".

API Call Results

Display Name: Shama-K
Business Phone:
Given Name: Shama
id: eb255477-ce5e-4c8f-95cf-e89ce6c4c6f0
User Principal Name: sAdmin@ms0604.onmicrosoft.com

Token Info

User Name: sAdmin@ms0604.onmicrosoft.com
Token Expires: 5/21/2020 6:12:03 PM -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. The Microsoft Authentication Library 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 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, 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.

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 multifactor authentication configured. The list of certificates appears. If you choose to use your PIN, the PIN window never appears.

Cause: This issue is a known limitation of the web authentication broker in UWP applications that run on Windows 10 desktops. 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 about your support options, see [Help and support for developers](#).

Next steps

Learn more about using the Microsoft Authentication Library (MSAL) for authorization and authentication in .NET applications:

[Overview of the Microsoft Authentication Library \(MSAL\)](#)

Tutorial: Sign in users and call Microsoft Graph in Windows Presentation Foundation (WPF) desktop app

4/12/2022 • 12 minutes to read • [Edit Online](#)

In this tutorial, you build a native Windows Desktop .NET (XAML) app that signs in users and gets an access token to call the Microsoft Graph API.

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.

In this tutorial:

- Create a *Windows Presentation Foundation (WPF)* project in Visual Studio
- Install the Microsoft Authentication Library (MSAL) for .NET
- Register the application in the Azure portal
- Add code to support user sign-in and sign-out
- Add code to call Microsoft Graph API
- Test the app

Prerequisites

- [Visual Studio 2019](#)

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. The 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 the Microsoft identity platform.

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
Microsoft.Identity.Client	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 the 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 - App registrations](#) quickstart experience.
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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application, for example `Win-App-calling-MsGraph`. Users of your app might see this name, and you can change it later.
6. In the **Supported account types** section, select **Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**.
7. Select **Register**.
8. Under **Manage**, select **Authentication > Add a platform**.
9. Select **Mobile and desktop applications**.
10. In the **Redirect URIs** section, select
<https://login.microsoftonline.com/common/oauth2/nativeclient>.
11. Select **Configure**.
12. 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)
            .WithDefaultRedirectUri()
            .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;
    }
}
```

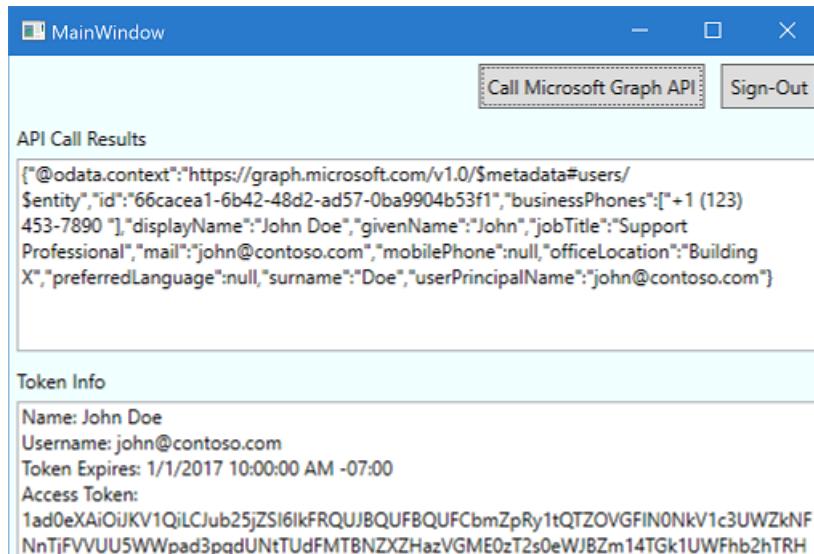
More information

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](#)

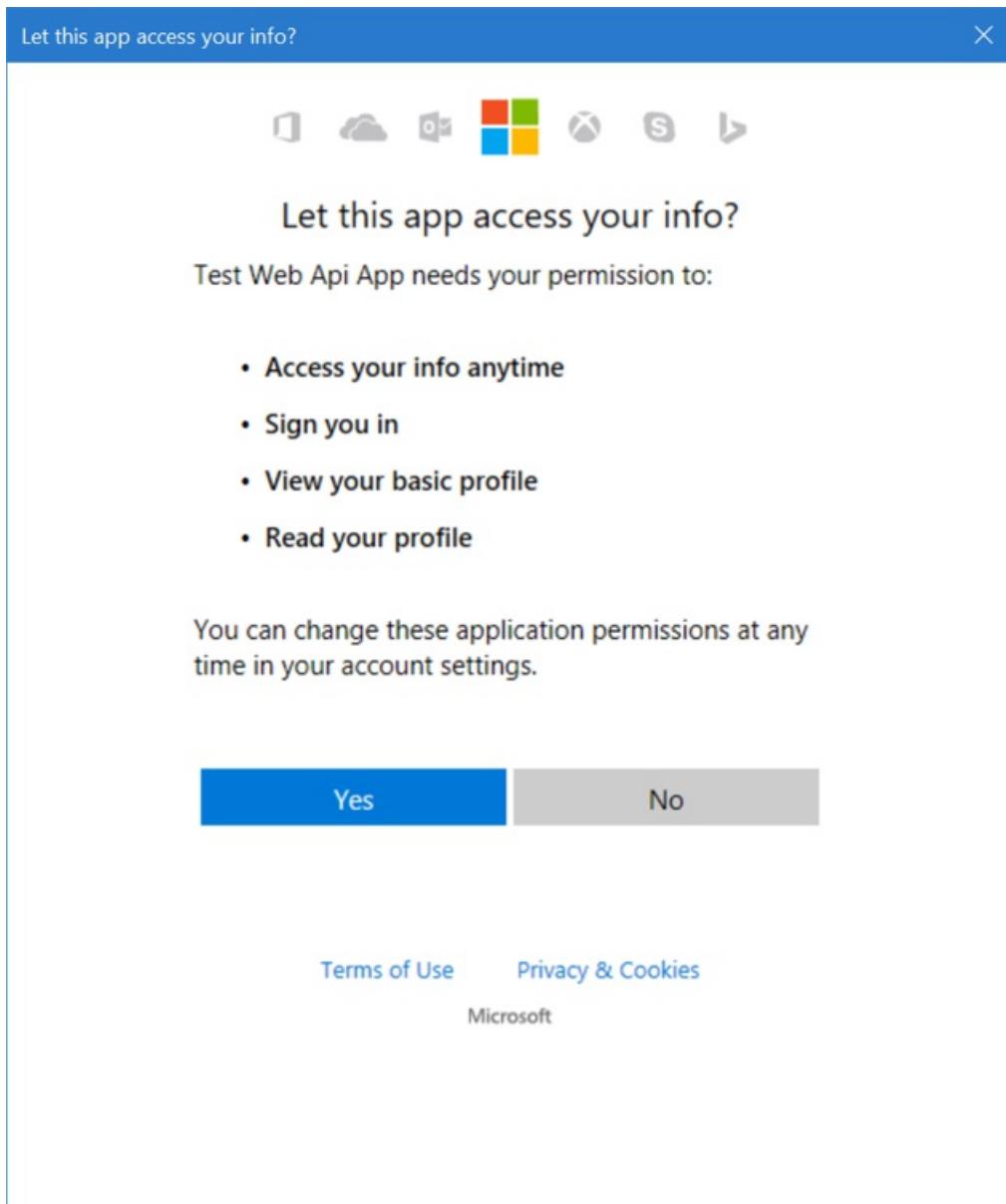
© 2017 Microsoft

[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 about your support options, see [Help and support for developers](#).

Next steps

Learn more about building desktop apps that call protected web APIs in our multi-part scenario series:

[Scenario: Desktop app that calls web APIs](#)

Tutorial: Sign in users and call the Microsoft Graph API in an Electron desktop app

4/12/2022 • 13 minutes to read • [Edit Online](#)

In this tutorial, you build an Electron desktop application that signs in users and calls Microsoft Graph by using the authorization code flow with PKCE. The desktop app you build uses the [Microsoft Authentication Library \(MSAL\) for Node.js](#).

Follow the steps in this tutorial to:

- Register the application in the Azure portal
- Create an Electron desktop app project
- Add authentication logic to your app
- Add a method to call a web API
- Add app registration details
- Test the app

Prerequisites

- [Node.js](#)
- [Electron](#)
- [Visual Studio Code](#) or another code editor

Register the application

First, complete the steps in [Register an application with the Microsoft identity platform](#) to register your app.

Use the following settings for your app registration:

- Name: `ElectronDesktopApp` (suggested)
- Supported account types: **Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)**
- Platform type: **Mobile and desktop applications**
- Redirect URI: `msal://redirect`

Create the project

Create a folder to host your application, for example `ElectronDesktopApp`.

1. First, change to your project directory in your terminal and then run the following `npm` commands:

```
npm init -y
npm install --save @azure/msal-node axios bootstrap dotenv jquery popper.js
npm install --save-dev babel electron@10.1.6 webpack
```

2. Then, create a folder named `App`. Inside this folder, create a file named `index.htm`/ that will serve as UI.

Add the following code there:

```
<!DOCTYPE html>
<html lang="en">
```

```

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0, shrink-to-fit=no">
    <meta http-equiv="Content-Security-Policy" content="script-src 'self' 'unsafe-inline';" />
    <title>MSAL Node Electron Sample App</title>

    <!-- adding Bootstrap 4 for UI components -->
    <link rel="stylesheet" href="../node_modules/bootstrap/dist/css/bootstrap.min.css">

    <link rel="SHORTCUT ICON" href="https://c.s-microsoft.com/favicon.ico?v2" type="image/x-icon">
</head>

<body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-primary">
        <a class="navbar-brand">Microsoft identity platform</a>
        <div class="btn-group ml-auto dropleft">
            <button type="button" id="signIn" class="btn btn-secondary" aria-expanded="false">
                Sign in
            </button>
            <button type="button" id="signOut" class="btn btn-success" hidden aria-expanded="false">
                Sign out
            </button>
        </div>
    </nav>
    <br>
    <h5 class="card-header text-center">Electron sample app calling MS Graph API using MSAL Node</h5>
    <br>
    <div class="row" style="margin:auto">
        <div id="cardDiv" class="col-md-3" style="display:none">
            <div class="card text-center">
                <div class="card-body">
                    <h5 class="card-title" id="WelcomeMessage">Please sign-in to see your profile and
                    read your mails
                    </h5>
                    <div id="profileDiv"></div>
                    <br>
                    <br>
                    <button class="btn btn-primary" id="seeProfile">See Profile</button>
                    <br>
                    <br>
                    <button class="btn btn-primary" id="readMail">Read Mails</button>
                </div>
            </div>
        </div>
        <br>
        <br>
        <div class="col-md-4">
            <div class="list-group" id="list-tab" role="tablist">
                </div>
            </div>
            <div class="col-md-5">
                <div class="tab-content" id="nav-tabContent">
                    </div>
                </div>
            </div>
        </div>
        <br>
        <br>
    </div>
    <script>
        window.jQuery = window.$ = require('jquery');
        require("./renderer.js");
    </script>

    <!-- importing bootstrap.js and supporting js libraries -->
    <script src="../node_modules/jquery/dist/jquery.js"></script>
    <script src="../node_modules/popper.js/dist/umd/popper.js"></script>
    <script src="../node_modules/bootstrap/dist/js/bootstrap.js"></script>
</body>

```

```
</body>
```

```
</html>
```

3. Next, create file named *main.js* and add the following code:

```
require('dotenv').config()

const path = require('path');
const { app, ipcMain, BrowserWindow } = require('electron');
const { IPC_MESSAGES } = require('./constants');

const { callEndpointWithToken } = require('./fetch');
const AuthProvider = require('./AuthProvider');

const authProvider = new AuthProvider();
let mainWindow;

function createWindow () {
    mainWindow = new BrowserWindow({
        width: 800,
        height: 600,
        webPreferences: {
            nodeIntegration: true,
            contextIsolation: false
        }
    });

    mainWindow.loadFile(path.join(__dirname, './index.html'));
}

app.on('ready', () => {
    createWindow();
});

app.on('window-all-closed', () => {
    app.quit();
});

// Event handlers
ipcMain.on(IPC_MESSAGES.LOGIN, async() => {
    const account = await authProvider.login(mainWindow);

    await mainWindow.loadFile(path.join(__dirname, './index.html'));

    mainWindow.webContents.send(IPC_MESSAGES.SHOW_WELCOME_MESSAGE, account);
});

ipcMain.on(IPC_MESSAGES.LOGOUT, async() => {
    await authProvider.logout();
    await mainWindow.loadFile(path.join(__dirname, './index.html'));
});

ipcMain.on(IPC_MESSAGES.GET_PROFILE, async() => {

    const tokenRequest = {
        scopes: ['User.Read'],
    };

    const token = await authProvider.getToken(mainWindow, tokenRequest);
    const account = authProvider.account

    await mainWindow.loadFile(path.join(__dirname, './index.html'));

    const graphResponse = await
callEndpointWithToken(` ${process.env.GRAPH_ENDPOINT_HOST}${process.env.GRAPH_ME_ENDPOINT}` , token);
```

```

        mainWindow.webContents.send(IPC_MESSAGES.SHOW_WELCOME_MESSAGE, account);
        mainWindow.webContents.send(IPC_MESSAGES.SET_PROFILE, graphResponse);
    });

    ipcMain.on(IPC_MESSAGES.GET_MAIL, async() => {

        const tokenRequest = {
            scopes: ['Mail.Read'],
        };

        const token = await authProvider.getToken(mainWindow, tokenRequest);
        const account = authProvider.account;

        await mainWindow.loadFile(path.join(__dirname, './index.html'));

        const graphResponse = await
callEndpointWithToken(` ${process.env.GRAPH_ENDPOINT_HOST}${process.env.GRAPH_MAIL_ENDPOINT}` , token);

        mainWindow.webContents.send(IPC_MESSAGES.SHOW_WELCOME_MESSAGE, account);
        mainWindow.webContents.send(IPC_MESSAGES.SET_MAIL, graphResponse);
    });
}

```

In the code snippet above, we initialize an Electron main window object and create some event handlers for interactions with the Electron window. We also import configuration parameters, instantiate *authProvider* class for handling sign-in, sign-out and token acquisition, and call the Microsoft Graph API.

4. In the same folder (*App*), create another file named *renderer.js* and add the following code:

```

const { ipcRenderer } = require('electron');
const { IPC_MESSAGES } = require('./constants');

// UI event handlers
document.querySelector('#signIn').addEventListener('click', () => {
    ipcRenderer.send(IPC_MESSAGES.LOGIN);
});

document.querySelector('#signOut').addEventListener('click', () => {
    ipcRenderer.send(IPC_MESSAGES.LOGOUT);
});

document.querySelector('#seeProfile').addEventListener('click', () => {
    ipcRenderer.send(IPC_MESSAGES.GET_PROFILE);
});

document.querySelector('#readMail').addEventListener('click', () => {
    ipcRenderer.send(IPC_MESSAGES.GET_MAIL);
});

// Main process message subscribers
ipcRenderer.on(IPC_MESSAGES.SHOW_WELCOME_MESSAGE, (event, account) => {
    showWelcomeMessage(account);
});

ipcRenderer.on(IPC_MESSAGES.SET_PROFILE, (event, graphResponse) => {
    updateUI(graphResponse, ` ${process.env.GRAPH_ENDPOINT_HOST}${process.env.GRAPH_ME_ENDPOINT}`);
});

ipcRenderer.on(IPC_MESSAGES.SET_MAIL, (event, graphResponse) => {
    updateUI(graphResponse, ` ${process.env.GRAPH_ENDPOINT_HOST}${process.env.GRAPH_MAIL_ENDPOINT}`);
});

// DOM elements to work with
const welcomeDiv = document.getElementById("WelcomeMessage");
const signInButton = document.getElementById("signIn");
const signOutButton = document.getElementById("signOut");
const cardDiv = document.getElementById("cardDiv");

```

```

const cardDiv = document.getElementById("cardDiv");
const profileDiv = document.getElementById("profileDiv");
const tabList = document.getElementById("list-tab");
const tabContent = document.getElementById("nav-tabContent");

function showWelcomeMessage(account) {
    cardDiv.style.display = "initial";
    welcomeDiv.innerHTML = `Welcome ${account.name}`;
    signInButton.hidden = true;
    signOutButton.hidden = false;
}

function clearTabs() {
    tabList.innerHTML = "";
    tabContent.innerHTML = "";
}

function updateUI(data, endpoint) {

    console.log(`Graph API responded at: ${new Date().toString()}`);

    if (endpoint === `${process.env.GRAPH_ENDPOINT_HOST}${process.env.GRAPH_ME_ENDPOINT}`) {
        setProfile(data);
    } else if (endpoint === `${process.env.GRAPH_ENDPOINT_HOST}${process.env.GRAPH_MAIL_ENDPOINT}`) {
        setMail(data);
    }
}

function setProfile(data) {
    profileDiv.innerHTML = ''

    const title = document.createElement('p');
    const email = document.createElement('p');
    const phone = document.createElement('p');
    const address = document.createElement('p');

    title.innerHTML = "<strong>Title: </strong>" + data.jobTitle;
    email.innerHTML = "<strong>Mail: </strong>" + data.mail;
    phone.innerHTML = "<strong>Phone: </strong>" + data.businessPhones[0];
    address.innerHTML = "<strong>Location: </strong>" + data.officeLocation;

    profileDiv.appendChild(title);
    profileDiv.appendChild(email);
    profileDiv.appendChild(phone);
    profileDiv.appendChild(address);
}

function setMail(data) {
    const mailInfo = data;
    if (mailInfo.value.length < 1) {
        alert("Your mailbox is empty!")
    } else {
        clearTabs();
        mailInfo.value.slice(0, 10).forEach((d, i) => {
            createAndAppendListItem(d, i);
            createAndAppendContentItem(d, i);
        });
    }
}

function createAndAppendListItem(d, i) {
    const listItem = document.createElement("a");
    listItem.setAttribute("class", "list-group-item list-group-item-action");
    listItem.setAttribute("id", "list" + i + "list")
    listItem.setAttribute("data-toggle", "list")
    listItem.setAttribute("href", "#list" + i)
    listItem.setAttribute("role", "tab")
    listItem.setAttribute("aria-controls", i)
    listItem.innerHTML = d.subject;
    tabList.appendChild(listItem);
}

```

```

        tabContent.appendChild(contentItem);
    }

    function createAndAppendContentItem(d, i) {
        const contentItem = document.createElement("div");
        contentItem.setAttribute("class", "tab-pane fade")
        contentItem.setAttribute("id", "list" + i)
        contentItem.setAttribute("role", "tabpanel")
        contentItem.setAttribute("aria-labelledby", "list" + i + "list")

        if (d.from) {
            contentItem.innerHTML = "<strong> from: " + d.from.emailAddress.address + "</strong><br><br>" +
            + d.bodyPreview + "...";
            tabContent.appendChild(contentItem);
        }
    }
}

```

- Finally, create a file named *constants.js* that will store the strings constants for describing the application events:

```

const IPC_MESSAGES = {
    SHOW_WELCOME_MESSAGE: 'SHOW_WELCOME_MESSAGE',
    LOGIN: 'LOGIN',
    LOGOUT: 'LOGOUT',
    GET_PROFILE: 'GET_PROFILE',
    SET_PROFILE: 'SET_PROFILE',
    GET_MAIL: 'GET_MAIL',
    SET_MAIL: 'SET_MAIL'
}

module.exports = {
    IPC_MESSAGES: IPC_MESSAGES,
}

```

You now have a simple GUI and interactions for your Electron app. After completing the rest of the tutorial, the file and folder structure of your project should look similar to the following:

```

ElectronDesktopApp/
├── App
│   ├── authProvider.js
│   ├── constants.js
│   ├── fetch.js
│   ├── main.js
│   ├── renderer.js
│   └── index.html
└── package.json
└── .env

```

Add authentication logic to your app

In *App* folder, create a file named *AuthProvider.js*. This will contain an authentication provider class that will handle login, logout, token acquisition, account selection and related authentication tasks using MSAL Node. Add the following code there:

```

const { PublicClientApplication, LogLevel, CryptoProvider } = require('@azure/msal-node');
const { protocol } = require('electron');
const path = require('path');
const url = require('url');

/**
 * To demonstrate best security practices, this Electron sample application makes use of
 * - https://github.com/AzureAD/microsoft-authentication-library-for-js#best-practices
 */

```

```

    * a custom file protocol instead of a regular web (https://) redirect URL in order to
    * handle the redirection step of the authorization flow, as suggested in the OAuth2.0 specification for
Native Apps.
  */
const CUSTOM_FILE_PROTOCOL_NAME = process.env.REDIRECT_URI.split(':')[0]; // e.g. msal://redirect

/**
 * Configuration object to be passed to MSAL instance on creation.
 * For a full list of MSAL Node configuration parameters, visit:
 * https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-
node/docs/configuration.md
 */
const MSAL_CONFIG = {
  auth: {
    clientId: process.env.CLIENT_ID,
    authority: `${process.env.AAD_ENDPOINT_HOST}${process.env.TENANT_ID}`,
    redirectUri: process.env.REDIRECT_URI,
  },
  system: {
    loggerOptions: {
      loggerCallback(loglevel, message, containsPii) {
        console.log(message);
      },
      piiLoggingEnabled: false,
      logLevel: LogLevel.Verbose,
    }
  }
};

class AuthProvider {

  clientApplication;
  cryptoProvider;
  authCodeUrlParams;
  authCodeRequest;
  pkceCodes;
  account;

  constructor() {
    /**
     * Initialize a public client application. For more information, visit:
     * https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-
node/docs/initialize-public-client-application.md
    */
    this.clientApplication = new PublicClientApplication(MSAL_CONFIG);
    this.account = null;

    // Initialize CryptoProvider instance
    this.cryptoProvider = new CryptoProvider();

    this.setRequestObjects();
  }

  /**
   * Initialize request objects used by this AuthModule.
   */
  setRequestObjects() {
    const requestScopes = ['openid', 'profile', 'User.Read'];
    const redirectUri = process.env.REDIRECT_URI;

    this.authCodeUrlParams = {
      scopes: requestScopes,
      redirectUri: redirectUri
    };

    this.authCodeRequest = {
      scopes: requestScopes,
      redirectUri: redirectUri,
      code: null
    };
  }
}

```

```

    }

    this.pkceCodes = {
        challengeMethod: "S256", // Use SHA256 Algorithm
        verifier: "", // Generate a code verifier for the Auth Code Request first
        challenge: "" // Generate a code challenge from the previously generated code verifier
    };
}

async login(authWindow) {
    const authResult = await this.getTokenInteractive(authWindow, this.authCodeUrlParams);
    return this.handleResponse(authResult);
}

async logout() {
    if (this.account) {
        await this.clientApplication.getTokenCache().removeAccount(this.account);
        this.account = null;
    }
}

async getToken(authWindow, tokenRequest) {
    let authResponse;

    authResponse = await this.getTokenInteractive(authWindow, tokenRequest);

    return authResponse.accessToken || null;
}

// This method contains an implementation of access token acquisition in authorization code flow
async getTokenInteractive(authWindow, tokenRequest) {

    /**
     * Proof Key for Code Exchange (PKCE) Setup
     *
     * MSAL enables PKCE in the Authorization Code Grant Flow by including the codeChallenge and
     * codeChallengeMethod parameters
     * in the request passed into getAuthCodeUrl() API, as well as the codeVerifier parameter in the
     * second leg (acquireTokenByCode() API).
     *
     * MSAL Node provides PKCE Generation tools through the CryptoProvider class, which exposes
     * the generatePkceCodes() asynchronous API. As illustrated in the example below, the verifier
     * and challenge values should be generated previous to the authorization flow initiation.
     *
     * For details on PKCE code generation logic, consult the
     * PKCE specification https://tools.ietf.org/html/rfc7636#section-4
     */
}

const {verifier, challenge} = await this.cryptoProvider.generatePkceCodes();

this.pkceCodes.verifier = verifier;
this.pkceCodes.challenge = challenge;

const authCodeUrlParams = {
    ...this.authCodeUrlParams,
    scopes: tokenRequest.scopes,
    codeChallenge: this.pkceCodes.challenge, // PKCE Code Challenge
    codeChallengeMethod: this.pkceCodes.codeChallengeMethod // PKCE Code Challenge Method
};

const authCodeUrl = await this.clientApplication.getAuthCodeUrl(authCodeUrlParams);

protocol.registerFileProtocol(CUSTOM_FILE_PROTOCOL_NAME, (req, callback) => {
    const requestUrl = url.parse(req.url, true);
    callback(path.normalize(`$__dirname__/${requestUrl.path}`));
});

const authCode = await this.listenForAuthCode(authCodeUrl, authWindow);
}

```

```

        const authResponse = await this.clientApplication.acquireTokenByCode({
            ...this.authCodeRequest,
            scopes: tokenRequest.scopes,
            code: authCode,
            codeVerifier: this.pkceCodes.verifier // PKCE Code Verifier
        });

        return authResponse;
    }

    // Listen for authorization code response from Azure AD
    async listenForAuthCode(navigateUrl, authWindow) {

        authWindow.loadURL(navigateUrl);

        return new Promise((resolve, reject) => {
            authWindow.webContents.on('will-redirect', (event, responseUrl) => {
                try {
                    const parsedUrl = new URL(responseUrl);
                    const authCode = parsedUrl.searchParams.get('code');
                    resolve(authCode);
                } catch (err) {
                    reject(err);
                }
            });
        });
    }

    /**
     * Handles the response from a popup or redirect. If response is null, will check if we have any
     * accounts and attempt to sign in.
     * @param response
     */
    async handleResponse(response) {
        if (response !== null) {
            this.account = response.account;
        } else {
            this.account = await this.getAccount();
        }

        return this.account;
    }

    /**
     * Calls getAllAccounts and determines the correct account to sign into, currently defaults to first
     * account found in cache.
     * https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-
     * common/docs/Accounts.md
     */
    async getAccount() {
        const cache = this.clientApplication.getTokenCache();
        const currentAccounts = await cache.getAllAccounts();

        if (currentAccounts === null) {
            console.log('No accounts detected');
            return null;
        }

        if (currentAccounts.length > 1) {
            // Add choose account code here
            console.log('Multiple accounts detected, need to add choose account code.');
            return currentAccounts[0];
        } else if (currentAccounts.length === 1) {
            return currentAccounts[0];
        } else {
            return null;
        }
    }
}

```

```
module.exports = AuthProvider;
```

In the code snippet above, we first initialized MSAL Node `PublicClientApplication` by passing a configuration object (`msalConfig`). We then exposed `login`, `logout` and `getToken` methods to be called by main module (`main.js`). In `login` and `getToken`, we acquire ID and access tokens, respectively, by first requesting an authorization code and then exchanging this with a token using MSAL Node `acquireTokenByCode` public API.

Add a method to call a web API

Create another file named `fetch.js`. This file will contain an Axios HTTP client for making REST calls to the Microsoft Graph API.

```
const axios = require('axios');

/**
 * Makes an Authorization 'Bearer' request with the given accessToken to the given endpoint.
 * @param endpoint
 * @param accessToken
 */
async function callEndpointWithToken(endpoint, accessToken) {
    const options = {
        headers: {
            Authorization: `Bearer ${accessToken}`
        }
    };

    console.log('Request made at: ' + new Date().toString());

    const response = await axios.default.get(endpoint, options);

    return response.data;
}

module.exports = {
    callEndpointWithToken: callEndpointWithToken,
};
```

Add app registration details

Finally, create an environment file to store the app registration details that will be used when acquiring tokens. To do so, create a file named `.env` inside the root folder of the sample (*ElectronDesktopApp*), and add the following code:

```

# Credentials
CLIENT_ID=Enter_the_Application_Id_Here
TENANT_ID=Enter_the_Tenant_Id_Here

# Configuration
REDIRECT_URI=msal://redirect

# Endpoints
AAD_ENDPOINT_HOST=Enter_the_Cloud_Instance_Id_Here
GRAPH_ENDPOINT_HOST=Enter_the_Graph_Endpoint_Here

# RESOURCES
GRAPH_ME_ENDPOINT=v1.0/me
GRAPH_MAIL_ENDPOINT=v1.0/me/messages

# SCOPES
GRAPH_SCOPES=User.Read Mail.Read

```

Fill in these details with the values you obtain from Azure app registration portal:

- `Enter_the_Tenant_Id_Here` should be one of the following:
 - 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`.
- `Enter_the_Application_Id_Here`: The **Application (client) ID** of the application you registered.
- `Enter_the_Cloud_Instance_Id_Here`: The Azure cloud instance in which your application is registered.
 - For the main (or *global*) Azure cloud, enter `https://login.microsoftonline.com/`.
 - For **national** clouds (for example, China), you can find appropriate values in [National clouds](#).
- `Enter_the_Graph_Endpoint_Here` is the instance of the Microsoft Graph API the application should communicate with.
 - For the **global** Microsoft Graph API endpoint, replace both instances of this string with `https://graph.microsoft.com/`.
 - For endpoints in **national** cloud deployments, see [National cloud deployments](#) in the Microsoft Graph documentation.

Test the app

You've completed creation of the application and are now ready to launch the Electron desktop app and test the app's functionality.

1. Start the app by running the following command from within the root of your project folder:

```
electron App/main.js
```

2. In application main window, you should see the contents of your `index.htm`/file and the **Sign In** button.

Test sign in and sign out

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



Sign in

Email, phone, or Skype

[Can't access your account?](#)

[Sign-in options](#)

Back

Next

If you consent to the requested permissions, the web application displays your user name, signifying a successful login:

The screenshot shows a Microsoft identity platform sign-in page at the top, followed by a sample application interface.

Microsoft identity platform (blue header bar)

Sign out (button on the right of the header)

Electron sample app calling MS Graph API using MSAL Node (title bar)

Welcome Diego Siciliani (User profile)

See Profile (button)

Read Mails (button)

Test web API call

After you sign in, select **See Profile** to view the user profile information returned in the response from the call to the Microsoft Graph API:

Electron sample app calling MS Graph API using MSAL Node

Welcome Diego Siciliani

Title: HR Manager

Mail:
DiegoS@msaltestingjs.onmicrosoft.com

Phone: +1 205 555 0108

Location: 14/1108

[See Profile](#)

[Read Mails](#)

Select **Read Mails** to view the messages in user's account. You'll be presented with a consent screen:



diegos@msaltestingjs.onmicrosoft.com

Permissions requested

[msal-node-desktop](#)

[App info](#)

This application is not published by Microsoft.

This app would like to:

- ✓ Read your mail
- ✓ View your basic profile
- ✓ Maintain access to data you have given it access to

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](#)

Does this app look suspicious? [Report it here](#)

[Cancel](#) [Accept](#)

After consent, you will view the messages returned in the response from the call to the Microsoft Graph API:

Electron sample app calling MS Graph API using MSAL Node

Welcome Diego Siciliani

See Profile

Read Mails

Welcome to MyAnalytics

Company All Hands

Lunch?

Meeting

Person in charge of Project Falcon?

Meeting update

from: no-reply@microsoft.com

MyAnalytics Discover your habits. Work smarter. For your eyes only Learn more > Welcome, Diego Siciliani! Your Office 365 account now includes MyAnalytics, a way to discover how you work MyAnalytics helps improve your... Foc...

How the application works

When a user selects the **Sign In** button for the first time, get `getTokenInteractive` method of *AuthProvider.js* is called. This method redirects the user to sign-in with the *Microsoft identity platform endpoint* and validate the user's credentials, and then obtains an **authorization code**. This code is then exchanged for an access token using `acquireTokenByCode` public API of MSAL Node.

At this point, a PKCE-protected authorization code is sent to the CORS-protected token endpoint and is exchanged for tokens. An ID token, access token, and refresh token are received by your application and processed by MSAL Node, and the information contained in the tokens is cached.

The ID token contains basic information about the user, like their display name. The access token has a limited lifetime and expires after 24 hours. If you plan to use these tokens for accessing protected resource, your back-end server *must* validate it to guarantee the token was issued to a valid user for your application.

The desktop app you've created in this tutorial makes a REST call to the Microsoft Graph API using an access token as bearer token in request header ([RFC 6750](#)).

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 in the Azure 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 `Mail.Read` scope in order to list the user's email.

As you add scopes, your users might be prompted to provide additional consent for the added scopes.

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

If you'd like to dive deeper into Node.js and Electron desktop application development on the Microsoft identity platform, see our multi-part scenario series:

[Scenario: Desktop app that calls web APIs](#)

Microsoft identity platform code samples

4/12/2022 • 9 minutes to read • [Edit Online](#)

These code samples are built and maintained by Microsoft to demonstrate usage of our authentication libraries with the Microsoft identity platform. Common authentication and authorization scenarios are implemented in several [application types](#), development languages, and frameworks.

- Sign in users to web applications and provide authorized access to protected web APIs.
- Protect a web API by requiring an access token to perform API operations.

Each code sample includes a *README.md* file describing how to build the project (if applicable) and run the sample application. Comments in the code help you understand how these libraries are used in the application to perform authentication and authorization by using the identity platform.

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Angular	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Call .NET Core web API• Call .NET Core web API (B2C)• Call Microsoft Graph via OBO• Call .NET Core web API using PoP• Use App Roles for access control• Use Security Groups for access control• Deploy to Azure Storage and App Service	MSAL Angular	<ul style="list-style-type: none">• Authorization code with PKCE• On-behalf-of (OBO)• Proof of Possession (PoP)
Blazor WebAssembly	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Deploy to Azure App Service	MSAL.js	Implicit Flow

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
JavaScript	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API via OBO and CA • Deploy to Azure Storage and App Service 	MSAL.js	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA)
React	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API using PoP • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service • Deploy to Azure Static Web Apps 	MSAL React	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA) • Proof of Possession (PoP)

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Customize token cache • Call Graph (multi-tenant) • Call Azure REST APIs • Protect web API • Protect web API (B2C) • Protect multi-tenant web API • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service 	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	<ul style="list-style-type: none"> • OpenID connect • Authorization code • On-Behalf-Of

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Blazor	Blazor Server Series <ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Call web API• Call web API (B2C)	MSAL.NET	Authorization code Grant Flow
ASP.NET Core	Advanced Token Cache Scenarios	<ul style="list-style-type: none">• MSAL.NET• Microsoft.Identity.Web	On-Behalf-Of (OBO)
ASP.NET Core	Use the Conditional Access auth context to perform step-up authentication	<ul style="list-style-type: none">• MSAL.NET• Microsoft.Identity.Web	Authorization code
ASP.NET Core	Active Directory FS to Azure AD migration	MSAL.NET	<ul style="list-style-type: none">• SAML• OpenID connect
ASP.NET	<ul style="list-style-type: none">• Microsoft Graph Training Sample• Sign in users and call Microsoft Graph• Sign in users and call Microsoft Graph with admin restricted scope• Quickstart: Sign in users	MSAL.NET	<ul style="list-style-type: none">• OpenID connect• Authorization code
Java Spring	Azure AD Spring Boot Starter Series <ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Use App Roles for access control• Use Groups for access control• Deploy to Azure App Service	<ul style="list-style-type: none">• MSAL Java• Azure AD Boot Starter	Authorization code
Java Servlets	Spring-less Servlet Series <ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Use App Roles for access control• Use Security Groups for access control• Deploy to Azure App Service	MSAL Java	Authorization code
Java	Sign in users and call Microsoft Graph	MSAL Java	Authorization code
Java Spring	Sign in users and call Microsoft Graph via OBO • Web API	MSAL Java	<ul style="list-style-type: none">• Authorization code• On-Behalf-Of (OBO)

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js Express	Express web app series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service • Use App Roles for access control • Use Security Groups for access control • Web app that sign in users 	MSAL Node	Authorization code
Python Flask	Flask Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Sign in users and call Microsoft Graph • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Python Django	Django Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Ruby	Graph Training <ul style="list-style-type: none"> • Sign in users and call Microsoft Graph 	OmniAuth OAuth2	Authorization code

Web API

The following samples show how to protect a web API with the Microsoft identity platform, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET	Call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
ASP.NET Core	Sign in users and call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
Java	Sign in users	MSAL Java	On-Behalf-Of (OBO)
Node.js	<ul style="list-style-type: none"> • Protect a Node.js web API • Protect a Node.js Web API with Azure AD B2C 	MSAL Node	Authorization bearer

Desktop

The following samples show public client desktop applications that access the Microsoft Graph API, or your own

web API in the name of the user. Apart from the *Desktop (Console) with Web Authentication Manager (WAM)* sample, all these client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> • Call Microsoft Graph • Call Microsoft Graph with token cache • Call Microsoft Graph with custom web UI HTML • Call Microsoft Graph with custom web browser • Sign in users with device code flow 	MSAL.NET	<ul style="list-style-type: none"> • Authorization code with PKCE • Device code
.NET	<ul style="list-style-type: none"> • Call Microsoft Graph with daemon console • Call web API with daemon console 	MSAL.NET	Authorization code with PKCE
.NET	Invoke protected API with integrated Windows authentication	MSAL.NET	Integrated Windows authentication
Java	Call Microsoft Graph	MSAL Java	Integrated Windows authentication
Node.js	Sign in users	MSAL Node	Authorization code with PKCE
PowerShell	Call Microsoft Graph by signing in users using username/password	MSAL.NET	Resource owner password credentials
Python	Sign in users	MSAL Python	Resource owner password credentials
Universal Window Platform (UWP)	Call Microsoft Graph	MSAL.NET	Web account manager
Windows Presentation Foundation (WPF)	Sign in users and call Microsoft Graph	MSAL.NET	Authorization code with PKCE
XAML	<ul style="list-style-type: none"> • Sign in users and call ASP.NET core web API • Sign in users and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Mobile

The following samples show public client mobile applications that access the Microsoft Graph API, or your own web API in the name of the user. These client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
iOS	<ul style="list-style-type: none"> Call Microsoft Graph native Call Microsoft Graph with Azure AD nxauth 	MSAL iOS	Authorization code with PKCE
Java	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Kotlin	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Xamarin	<ul style="list-style-type: none"> Sign in users and call Microsoft Graph Sign in users with broker and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Service / daemon

The following samples show an application that accesses the Microsoft Graph API with its own identity (with no user).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> Call Microsoft Graph Call web API Call own web API Using managed identity and Azure key vault 	MSAL.NET	Client credentials grant
ASP.NET	Multi-tenant with Microsoft identity platform endpoint	MSAL.NET	Client credentials grant
Java	Call Microsoft Graph	MSAL Java	Client credentials grant
Node.js	Sign in users and call web API	MSAL Node	Client credentials grant
Python	<ul style="list-style-type: none"> Call Microsoft Graph with secret Call Microsoft Graph with certificate 	MSAL Python	Client credentials grant

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, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET	.NET Azure function web API secured by Azure AD	MSAL.NET	Authorization code

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Node.js Azure function web API secured by Azure AD	MSAL Node	Authorization bearer
Node.js	Call Microsoft Graph API on behalf of a user	MSAL Node	On-Behalf-Of (OBO)
Python	Python Azure function web API secured by Azure AD	MSAL Python	Authorization code

Headless

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 the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET core	Invoke protected API from text-only device	MSAL.NET	Device code
Java	Sign in users and invoke protected API	MSAL Java	Device code
Python	Call Microsoft Graph	MSAL Python	Device code

Microsoft Teams applications

The following sample illustrates Microsoft Teams Tab application that signs in users. Additionally it demonstrates how to call Microsoft Graph API with the user's identity using the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Teams Tab app: single sign-on (SSO) and call Microsoft Graph	MSAL Node	On-Behalf-Of (OBO)

Multi-tenant SaaS

The following samples show how to configure your application to accept sign-ins from any Azure Active Directory (Azure AD) tenant. Configuring your application to be *multi-tenant* means that you can offer a **Software as a Service** (SaaS) application to many organizations, allowing their users to be able to sign-in to your application after providing consent.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core MVC web application calls Microsoft Graph API	MSAL.NET	OpenID connect
ASP.NET Core	ASP.NET Core MVC web application calls ASP.NET Core Web API	MSAL.NET	Authorization code

Next steps

If you'd like to delve deeper into more sample code, see:

- [Sign in users and call the Microsoft Graph API from an Angular](#)
- [Sign in users in a Nodejs and Express web app](#)
- [Call the Microsoft Graph API from a Universal Windows Platform](#)

Scenario: Desktop app that calls web APIs

4/12/2022 • 2 minutes to read • [Edit Online](#)

Learn all you need to build a desktop app that calls web APIs.

Get started

If you haven't already, create your first application by completing a 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](#)
- [Quickstart: Acquire a token and call Microsoft Graph API from a Nodejs & Electron app](#)

Overview

You write a desktop application, and you want to sign in users to your application and call web APIs such as Microsoft Graph, other Microsoft APIs, or your own web API. You've several options:

- You can use the interactive token acquisition:
 - If your desktop application supports graphical controls, for instance, if it's a Windows Form application, a Windows Presentation Foundation (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 Active Directory (Azure AD) happen in the system browser.
 - Or, if it's a Node.js Electron application, which runs on a Chromium instance.
- For Windows hosted applications, it's also possible for applications running on computers joined to a Windows domain or Azure AD joined to acquire a token silently by using integrated Windows authentication.
- Finally, and although it's not recommended, you can use a username and a password in public client applications. It's still needed in some scenarios like DevOps. Using it imposes constraints on your application. For instance, it can't sign in a user who needs to do [multifactor 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.



- If you write a portable command-line tool, probably a .NET Core application that runs on Linux or Mac, and if you accept that authentication will be delegated to the system browser, you can use interactive authentication. .NET Core doesn't provide a [web browser](#), so 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 Internet of Things (IoT) applications.



Specifics

Desktop applications have few specificities. They depend mainly on whether your application uses interactive authentication or not.

Recommended reading

If you're new to identity and access management (IAM) with OAuth 2.0 and OpenID Connect, or even just new to IAM on the Microsoft identity platform, the following set of articles should be high on your reading list.

Although not required reading before completing your first quickstart or tutorial, they cover topics integral to the platform, and familiarity with them will help you on your path as you build more complex scenarios.

Authentication and authorization

- [Authentication basics](#)
- [ID tokens](#)
- [Access tokens](#)

Microsoft identity platform

- [Audiences](#)
- [Applications and service principals](#)
- [Permissions and consent](#)

Next steps

Move on to the next article in this scenario, [App registration](#).

Desktop app that calls web APIs: App registration

4/12/2022 • 3 minutes to read • [Edit Online](#)

This article covers the app registration specifics for a desktop application.

Supported account types

The account types supported in a 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 a username and a password, your application needs to sign in users in your own tenant, for example, if you're a line-of-business (LOB) developer. Or, in Azure Active Directory organizations, your application needs to sign in users in your own tenant if it's an ISV scenario. These authentication flows aren't supported for Microsoft personal accounts.
- If you sign in users with social identities that pass a business-to-commerce (B2C) authority and policy, you can only use the interactive and username-password authentication.

Redirect URIs

The redirect URIs to use in a desktop application depend on the flow you want to use.

Specify the redirect URI for your app by [configuring the platform settings](#) for the app in [App registrations](#) in the Azure portal.

- For apps that use [Web Authentication Manager \(WAM\)](#), redirect URIs need not be configured in MSAL, but they must be configured in the [app registration](#).
- For apps that use interactive authentication:
 - Apps that use embedded browsers: `https://login.microsoftonline.com/common/oauth2/nativeclient` (Note: If your app would pop up a window which typically contains no address bar, it is using the "embedded browser".)
 - Apps that use system browsers: `http://localhost` (Note: If your app would bring your system's default browser (such as Edge, Chrome, Firefox, etc.) to visit Microsoft login portal, it is using the "system browser".)

IMPORTANT

As a security best practice, we recommend explicitly setting

`https://login.microsoftonline.com/common/oauth2/nativeclient` or `http://localhost` as the redirect URI.

Some authentication libraries like MSAL.NET use a default value of `urn:ietf:wg:oauth:2.0:oob` when no other redirect URI is specified, which is not recommended. This default will be updated as a breaking change in the next major release.

- If you build a native Objective-C or Swift app for macOS, register the redirect URI 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 you build a Node.js Electron app, use a custom file protocol instead of a regular web (<https://>) redirect URI in order to handle the redirection step of the authorization flow, for instance `msal://redirect`. The custom file protocol name shouldn't be obvious to guess and should follow the suggestions in the [OAuth2.0 specification for Native Apps](#).
- If your app uses only integrated Windows authentication or a username and a 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. Your application won't be called back on any specific URI.
- To distinguish [device code flow](#), [integrated Windows authentication](#), and a [username and a password](#) from a confidential client application using a client credential flow used in [daemon applications](#), none of which requires a redirect URI, configure it as a public client application. To achieve this configuration:
 1. In the [Azure portal](#), select your app in [App registrations](#), and then select [Authentication](#).
 2. In [Advanced settings > Allow public client flows > Enable the following mobile and desktop flows:](#), select **Yes**.

Advanced settings

Allow public client flows ⓘ

Enable the following mobile and desktop flows:

Yes

No

- App collects plaintext password (Resource Owner Password Credential Flow) [Learn more ↗](#)
- No keyboard (Device Code Flow) [Learn more ↗](#)
- SSO for domain-joined Windows (Windows Integrated Auth Flow) [Learn more ↗](#)

API permissions

Desktop applications call APIs for the signed-in user. They need to request delegated permissions. They can't request application permissions, which are handled only in [daemon applications](#).

Next steps

Move on to the next article in this scenario, [App Code configuration](#).

Desktop app that calls web APIs: Code configuration

4/12/2022 • 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.

Microsoft libraries supporting desktop apps

The following Microsoft libraries support desktop apps:

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
Electron	MSAL Node.js	<code>msal-node</code>	—	✓	✓	Public preview
Java	MSAL4J	<code>msal4j</code>	—	✓	✓	GA
macOS (Swift/Obj-C)	MSAL for iOS and macOS	<code>MSAL</code>	Tutorial	✓	✓	GA
UWP	MSAL.NET	<code>Microsoft.Identity.Client</code>	Tutorial	✓	✓	GA
WPF	MSAL.NET	<code>Microsoft.Identity.Client</code>	Tutorial	✓	✓	GA

¹ [Supplemental terms of use for Microsoft Azure Previews](#) apply to libraries in *Public preview*.

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](#)
- [MacOS](#)
- [Node.js](#)
- [Python](#)

You'll need to build and manipulate MSAL.NET `IPublicClientApplication`.

The screenshot shows the Microsoft API Explorer interface for the `IPublicClientApplication` interface. The interface inherits from `IClientApplicationBase`. It has two sections: **Properties** and **Methods**. The **Properties** section contains one item: `IsSystemWebViewAvailable { get; } : bool`. The **Methods** section contains four items: `AcquireTokenByIntegratedWindowsAuth(IEnumerable<string> scopes) : AcquireTokenByIntegratedWindowsA...`, `AcquireTokenByUsernamePassword(IEnumerable<string> scopes, string username, SecureString password) : A...`, `AcquireTokenInteractive(IEnumerable<string> scopes, object parent) : AcquireTokenInteractiveParameterBuil...`, and `AcquireTokenWithDeviceCode(IEnumerable<string> scopes, Func<DeviceCodeResult, Task> deviceCodeResul...`.

Exclusively by code

The following code instantiates a public client application and signs in users in the Microsoft Azure public cloud with a work or 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 previously, use the `.WithRedirectUri` modifier.

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

Use 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 multitenant application in a national cloud, such as US Government shown here, you could write:

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
    .WithDefaultRedirectUri()
    .WithAadAuthority(AzureCloudInstance.AzureUsGovernment,
                      AadAuthorityAudience.AzureAdMultipleOrgs)
    .Build();
```

MSAL.NET also contains a modifier for Active Directory Federation Services 2019:

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
    .WithAdfsAuthority("https://consoso.com/adfs")
    .Build();
```

Finally, if you want to acquire tokens for an Azure Active Directory (Azure AD) B2C tenant, 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 about how to configure an MSAL.NET desktop application:

- For a list of all modifiers available on `PublicClientApplicationBuilder`, see the reference documentation [PublicClientApplicationBuilder](#).
- For a 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 in this file by 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 runs
    /// in Microsoft Azure public clouds or national or 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 configuration
        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, write the following code:

```

SampleConfiguration config = SampleConfiguration.ReadFromJsonFile("appsettings.json");
var app = PublicClientApplicationBuilder.CreateWithApplicationOptions(config.PublicClientApplicationOptions)
    .WithDefaultRedirectUri()
    .Build();

```

Before the call to the `.Build()` method, you can override your configuration with calls to `.Withxxx` methods, as seen previously.

Next steps

Move on to the next article in this scenario, [Acquire a token for the desktop app](#).

Desktop app that calls web APIs: Acquire a token

4/12/2022 • 3 minutes to read • [Edit Online](#)

After you've 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 to use is:

- Systematically attempt to get a token from the token cache by calling `AcquireTokenSilent`.
- If this call fails, use the `AcquireToken` flow that you want to use, which is represented here by `AcquireTokenXX`
 - [.NET](#)
 - [Java](#)
 - [macOS](#)
 - [Node.js](#)
 - [Python](#)

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();
}
```

There are various ways you can acquire tokens in a desktop application.

- [Interactively](#)
- [Integrated Windows authentication](#)
- [WAM](#)
- [Username Password](#)
- [Device code flow](#)

Next steps

Move on to the next article in this scenario, [Call a web API from the desktop app](#).

Desktop app that calls web APIs: Acquire a token interactively

4/12/2022 • 10 minutes to read • [Edit Online](#)

The following example shows minimal code to get a token interactively for reading the user's profile with Microsoft Graph.

- [.NET](#)
- [Java](#)
- [macOS](#)
- [Node.js](#)
- [Python](#)

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 Microsoft Graph, the required scopes can be found in the 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" must be used. For more information, see [Microsoft Graph permissions reference](#).

On Android, you also need to specify the parent activity by using `.WithParentActivityOrWindow`, as shown, so that the token gets back to that parent activity after the interaction. If you don't specify it, an exception is thrown when calling `.ExecuteAsync()`.

Specific optional parameters in MSAL.NET

`WithParentActivityOrWindow`

The UI is important because it's interactive. `AcquireTokenInteractive` has one specific optional parameter that can specify, for platforms that support it, the parent UI. When used in a desktop application,

`.WithParentActivityOrWindow` has a different type, which depends on the platform. Alternatively you can omit the optional parent window parameter to create a window, if you do not want to control where the sign-in dialog appears on the screen. This would be applicable for applications which are command line based, used to pass calls to any other backend service and do not need any windows for user interaction.

```

// 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 `Activity` on Android, `UIViewController` on iOS, `NSWindow` on Mac, and `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 might cause messages to not pump properly and deadlock scenarios with the UI. One way of calling Microsoft Authentication Libraries (MSALs) 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 the `WindowInteropHelper.Handle` class. Then the call is 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 exact behavior can be controlled by using the [Microsoft.Identity.Client.Prompt](#) structure.

The struct defines the following constants:

- `SelectAccount` forces the STS to present the account selection dialog box that contains accounts for which the user has a session. This option is useful when application developers want to let users choose among different identities. This option drives MSAL to send `prompt=select_account` to the identity provider. This option is the default. It does a good job of providing the best possible experience based on the available information, such as account and presence of a session for the user. Don't change it unless you have good reason to do it.
- `Consent` enables the application developer to force the user to 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 with the consent dialog box 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 might not be needed. This option can be useful to let the user sign in again if acquiring a token fails. In this case, MSAL sends `prompt=login` to the identity provider. Sometimes it's used in security-focused applications where the organization governance demands that the user re-signs in each time they access specific parts of an application.
- `Create` triggers a sign-up experience, which is used for External Identities, by sending `prompt=create` to the identity provider. This prompt should not be sent for Azure AD B2C apps. For more information, see [Add a self-service sign-up user flow to an app](#).
- `Never` (for .NET 4.5 and WinRT only) won't prompt the user, but instead tries to use the cookie stored in the hidden embedded web view. For more information, see [web views in MSAL.NET](#). Using this option might fail.

In that case, `AcquireTokenInteractive` throws an exception to notify that a UI interaction is needed. You'll need to use another `Prompt` parameter.

- `NoPrompt` won't send any prompt to the identity provider which therefore will decide to present the best sign-in experience to the user (single-sign-on, or select account). This option is also mandatory for Azure Active Directory (Azure AD) B2C edit profile policies. For more information, see [Azure AD B2C specifics](#).

WithUseEmbeddedWebView

This method enables you to specify if you want to force the usage of an embedded WebView or the system WebView (when available). For more information, see [Usage of web browsers](#).

```
var result = await app.AcquireTokenInteractive(scopes)
    .WithUseEmbeddedWebView(true)
    .ExecuteAsync();
```

WithExtraScopeToConsent

This modifier is used in an advanced scenario where you want the user to pre-consent to several resources upfront, and you don't want to use incremental consent, which is normally used with MSAL.NET/the Microsoft identity platform. For more information, see [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 cases where you might want to host the browser yourself:

- Platforms that aren't explicitly covered by MSAL, for example, Blazor, Unity, and Mono on desktops.
- You want to UI test your application and use an automated browser that can be used with Selenium.
- The browser and the app that run MSAL are in separate processes.

At a glance

To achieve this, you give to MSAL `start Uri`, which needs to be displayed in a browser of choice so that the end user can enter items such as their username. After authentication finishes, your app needs to pass back to MSAL `end Uri`, which contains a code provided by Azure AD. The host of `end Uri` is always `redirectUri`. To intercept `end Uri`, do one of the following things:

- Monitor browser redirects until `redirect Uri` is hit.
- Have the browser redirect to a URL, which you monitor.

WithCustomWebUi is an extensibility point

`WithCustomWebUi` is an extensibility point that you can use to provide your own UI in public client applications. You can also 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. For example, it's used in Visual Studio to have electron applications (for instance, Visual Studio 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 Proof Key for Code Exchange (PKCE) standard to ensure that security is respected. Only MSAL.NET can redeem the code. For more information, see [RFC 7636 - Proof Key for Code Exchange by OAuth Public Clients](#).

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

Use WithCustomWebUi

To use `.WithCustomWebUI`, follow these steps.

1. Implement the `ICustomWebUi` interface. For more information, see [this website](#). Implement one `AcquireAuthorizationCodeAsync` method and accept the authorization code URL computed by MSAL.NET. Then let the user go through the interaction with the identity provider and return back the URL by which the identity provider would have called your implementation back along with the authorization code. If you have issues, your implementation should throw a `MsalExtensionException` exception to nicely cooperate with MSAL.
2. In your `AcquireTokenInteractive` call, use the `.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: [SeleniumWebUI](#)

The MSAL.NET team has rewritten the UI tests to use this extensibility mechanism. If you're interested, look at the [SeleniumWebUI](#) class in the MSAL.NET source code.

Provide a great experience with `SystemWebViewOptions`

From MSAL.NET 4.1 `SystemWebViewOptions`, you can specify:

- The URI to go to (`BrowserRedirectError`) or the HTML fragment to display (`HtmlMessageError`) in case of sign-in or consent errors in the system web browser.
- The URI to go to (`BrowserRedirectSuccess`) or the HTML fragment to display (`HtmlMessageSuccess`) in case of successful sign-in or consent.
- The action to run to start the system browser. You can provide your own implementation by setting the `OpenBrowserAsync` delegate. The class also provides a default implementation for two browsers: `OpenWithEdgeBrowserAsync` and `OpenWithChromeEdgeBrowserAsync` for Microsoft Edge and [Microsoft Edge on Chromium](#), respectively.

To use this structure, write something like the following example:

```
IPublicClientApplication app;
...

options = new SystemWebViewOptions
{
    HtmlMessageError = "<b>Sign-in failed. You can close this tab ...</b>",
    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

To learn more about all the other optional parameters for `AcquireTokenInteractive`, see [AcquireTokenInteractiveParameterBuilder](#).

Next steps

Move on to the next article in this scenario, [Call a web API from the desktop app](#).

Desktop app that calls web APIs: Acquire a token using WAM

4/12/2022 • 5 minutes to read • [Edit Online](#)

MSAL is able to call Web Account Manager, a Windows 10 component that ships with the OS. This component acts as an authentication broker and users of your app benefit from integration with accounts known from Windows, such as the account you signed-in with in your Windows session.

Availability

MSAL 4.25+ supports WAM on UWP, .NET Classic, .NET Core 3.1, and .NET 5.

For .NET Classic and .NET Core 3.1, WAM functionality is fully supported but you have to add a reference to [Microsoft.Identity.Client.Desktop](#) package, alongside MSAL, and instead of `WithBroker()`, call `.WithWindowsBroker()`.

For .NET 5, target `net5.0-windows10.0.17763.0` (or higher) and not just `net5.0`. Your app will still run on older versions of Windows if you add `<SupportedOSPlatformVersion>7</SupportedOSPlatformVersion>` in the csproj. MSAL will use a browser when WAM is not available.

WAM value proposition

Using an authentication broker such as WAM has numerous benefits.

- Enhanced security (your app does not have to manage the powerful refresh token)
- Better support for Windows Hello, Conditional Access and FIDO keys
- Integration with Windows' "Email and Accounts" view
- Better Single Sign-On (users don't have to reenter passwords)
- Most bug fixes and enhancements will be shipped with Windows

WAM limitations

- B2C and ADFS authorities are not supported. MSAL will fallback to a browser.
- Available on Win10+ and Win Server 2019+. On Mac, Linux and earlier Windows MSAL will fallback to a browser.
- Not available on Xbox.

WAM calling pattern

You can use the following pattern to use WAM.

```

// 1. Configuration - read below about redirect URI
var pca = PublicClientApplicationBuilder.Create("client_id")
    .WithBroker()
    .Build();

// Add a token cache, see https://docs.microsoft.com/en-us/azure/active-directory/develop/msal-net-token-cache-serialization?tabs=desktop

// 2. GetAccounts
var accounts = await pca.GetAccountsAsync();
var accountToLogin = // choose an account, or null, or use PublicClientApplication.OperatingSystemAccount
for the default OS account

try
{
    // 3. AcquireTokenSilent
    var authResult = await pca.AcquireTokenSilent(new[] { "User.Read" }, accountToLogin)
        .ExecuteAsync();
}

catch (MsalUiRequiredException) // no change in the pattern
{
    // 4. Specific: Switch to the UI thread for next call . Not required for console apps.
    await SwitchToUiThreadAsync(); // not actual code, this is different on each platform / tech

    // 5. AcquireTokenInteractive
    var authResult = await pca.AcquireTokenInteractive(new[] { "User.Read" })
        .WithAccount(accountToLogin) // this already exists in MSAL, but it is more
important for WAM
        .WithParentActivityOrWindow(myWindowHandle) // to be able to parent WAM's
windows to your app (optional, but highly recommended; not needed on UWP)
        .ExecuteAsync();
}

```

Call `.WithBroker(true)`. If a broker is not present (e.g. Win8.1, Mac, or Linux), then MSAL will fallback to a browser! Redirect URI rules apply to the browser.

Redirect URI

WAM redirect URIs do not need to be configured in MSAL, but they must be configured in the app registration.

Win32 (.NET framework / .NET 5)

```
ms-appx-web://microsoft.aad.brokerplugin/{client_id}
```

UWP

```

// returns smth like S-1-15-2-2601115387-131721061-1180486061-1362788748-631273777-3164314714-
2766189824
string sid = WebAuthenticationBroker.GetCurrentApplicationCallbackUri().Host.ToUpper();

// the redirect uri you need to register
string redirectUri = $"ms-appx-web://microsoft.aad.brokerplugin/{sid}";

```

Token cache persistence

It's important to persist MSAL's token cache because MSAL needs to save internal WAM account IDs there. Without it, restarting the app means that `GetAccounts` API will miss some of the accounts. Note that on UWP, MSAL knows where to save the token cache.

GetAccounts

`GetAccounts` returns accounts of users who have previously logged in interactively into the app.

In addition to this, WAM can list the OS-wide Work and School accounts configured in Windows (for Win32 apps but not for UWP apps). To opt-into this feature, set `ListWindowsWorkAndSchoolAccounts` in `WindowsBrokerOptions` to `true`. You can enable it as below.

```
.WithWindowsBrokerOptions(new WindowsBrokerOptions()
{
    // GetAccounts will return Work and School accounts from Windows
    ListWindowsWorkAndSchoolAccounts = true,

    // Legacy support for 1st party apps only
    MsaPassthrough = true
})
```

NOTE

Microsoft (i.e. outlook.com etc.) accounts will not be listed in Win32 nor UWP for privacy reasons.

Applications cannot remove accounts from Windows!

RemoveAsync

- Removes all account information from MSAL's token cache (this includes MSA - i.e. personal accounts - account info and other account information copied by MSAL into its cache).
- Removes app-only (not OS-wide) accounts.

NOTE

Apps cannot remove OS accounts. Only users can do that. If an OS account is passed into `RemoveAsync`, and then `GetAccounts` is called with `ListWindowsWorkAndSchoolAccounts` enabled - the same OS account will still be returned.

Other considerations

- WAM's interactive operations require being on the UI thread. MSAL throws a meaningful exception when not on UI thread. This does NOT apply to console apps.
- `WithAccount` provides an accelerated authentication experience if the MSAL account was originally obtained via WAM, or, WAM can find a work and school account in Windows.
- WAM is not able to pre-populate the username field with a login hint, unless a Work and School account with the same username is found in Windows.
- If WAM is unable to offer an accelerated authentication experience, it will show an account picker. Users can add new accounts.

Sign in

X

Let's get you signed in

Use one of these accounts



garth.fort@contoso.onmicrosoft.com

Work or school account



gfort@contoso.com

Work or school account



garth.fort@outlook.com

Microsoft account

Use a different account



Microsoft account



Email, phone, or Skype

Continue

- New accounts are automatically remembered by Windows. Work and School have the option of joining the organization's directory or opting out completely, in which case the account will not appear under "Email & Accounts". Microsoft accounts are automatically added to Windows. Apps cannot list these accounts programmatically (but only through the Account Picker).

Troubleshooting

"Either the user cancelled the authentication or the WAM Account Picker crashed because the app is running in an elevated process" error message

When an app that uses MSAL is run as an elevated process, some of these calls within WAM may fail due to different process security levels. Internally MSAL.NET uses native Windows methods ([COM](#)) to integrate with WAM. Starting with version 4.32.0, MSAL will display a descriptive error message when it detects that the app process is elevated and WAM returned no accounts.

One solution is to not run the app as elevated, if possible. Another solution is for the app developer to call `WindowsNativeUtils.InitializeProcessSecurity` method when the app starts up. This will set the security of the processes used by WAM to the same levels. See [this sample app](#) for an example. However, note, that this

solution is not guaranteed to succeed due to external factors like the underlying CLR behavior. In that case, an `MsalClientException` will be thrown. See issue #2560 for additional information.

"WAM Account Picker did not return an account" error message

This message indicates that either the application user closed the dialog that displays accounts, or the dialog itself crashed. A crash might occur if AccountsControl, a Windows control, is registered incorrectly in Windows. To resolve this issue:

1. In the taskbar, right-click **Start**, and then select **Windows PowerShell (Admin)**.
2. If you're prompted by a User Account Control (UAC) dialog, select **Yes** to start PowerShell.
3. Copy and then run the following script:

```
if (-not (Get-AppxPackage Microsoft.AccountsControl)) { Add-AppxPackage -Register  
"$env:windir\SystemApps\Microsoft.AccountsControl_cw5n1h2txyewy\AppxManifest.xml" -  
DisableDevelopmentMode -ForceApplicationShutdown } Get-AppxPackage Microsoft.AccountsControl
```

Connection issues

The application user sees an error message similar to "Please check your connection and try again". If this issue occurs regularly, see the [troubleshooting guide for Office](#), which also uses WAM.

Sample

[WPF sample that uses WAM](#)

[UWP sample that uses WAM, along Xamarin](#)

Next steps

Move on to the next article in this scenario, [Call a web API from the desktop app](#).

Desktop app that calls web APIs: Acquire a token using integrated Windows authentication

4/12/2022 • 5 minutes to read • [Edit Online](#)

To sign in a domain user on a domain or Azure AD joined machine, use integrated Windows authentication (IWA).

Constraints

- Integrated Windows authentication is available for *federated+* users only, that is, users created in Active Directory and backed by Azure AD. Users created directly in Azure AD without Active Directory backing, known as *managed* users, can't use this authentication flow. This limitation doesn't affect the username and password flow.
- IWA doesn't bypass [multi-factor authentication \(MFA\)](#). If MFA is configured, IWA might fail if an MFA challenge is required, because MFA requires user interaction.

IWA is non-interactive, but MFA requires user interactivity. You don't control when the identity provider requests MFA to be performed, the tenant admin does. From our observations, MFA is required when you sign in from a different country/region, 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 AD uses AI to continuously learn if MFA is required. Fall back to a user prompt like interactive authentication or device code flow if IWA fails.

- The authority passed in `PublicClientApplicationBuilder` needs to be:
 - Tenanted of the form `https://login.microsoftonline.com/{tenant}/`, where `{tenant}` is either the GUID that represents 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 supported. You can't 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 selected the **Grant** button in the Azure portal for yourself.
 - Or, a tenant admin selected the **Grant/revoke admin consent for {tenant domain}** button on the **API permissions** tab of the registration for the application. For more information, see [Add permissions to access your web API](#).
 - Or, you've provided a way for users to consent to the application. For more information, see [Requesting individual user consent](#).
 - Or, you've provided a way for the tenant admin to consent to the application. For more information, see [Admin consent](#).
- This flow is enabled for .NET desktop, .NET Core, and UWP apps.

For more information on consent, see the [Microsoft identity platform permissions and consent](#).

[Learn how to use it](#)

- .NET
- Java
- macOS
- Node.js
- Python

In MSAL.NET, use:

```
AcquireTokenByIntegratedWindowsAuth(IEnumerable<string> scopes)
```

You normally need only one parameter (`scopes`). Depending on the way your Windows administrator set up the policies, applications on your Windows machine might not be allowed to look up the signed-in user. In that case, use a second method, `.WithUsername()`, and pass in the username of the signed-in user as a UPN format, for example, `joe@contoso.com`.

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](#).

Next steps

Move on to the next article in this scenario, [Call a web API from the desktop app](#).

Desktop app that calls web APIs: Acquire a token using Username and Password

4/12/2022 • 9 minutes to read • [Edit Online](#)

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

The username and password flow is *not recommended* because having your application ask a user for their password isn't secure. 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](#). You can also use [device code flow](#).

Using a username and password is useful in some cases, such as DevOps scenarios. But if you want to use a username and password in interactive scenarios where you provide your own UI, think about how to move away from it. By using a username and password, you're giving up a number of things:

- Core tenets of modern identity. A password can get phished and replayed because a shared secret can be intercepted. It's incompatible with passwordless.
- Users who need to do MFA can't sign in because there's no interaction.
- Users can't do single sign-on (SSO).

Constraints

The following constraints also apply:

- The username and 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

For more information, see [Resource Owner Password Credentials \(ROPC\) with B2C](#).

Use it

- [.NET](#)
- [Java](#)
- [macOS](#)
- [Node.js](#)
- [Python](#)

`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 = "AADSTS50002: 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)
        // -----
    }
}

```

```
// 1D3242: 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 more information on all the modifiers that can be applied to `AcquireTokenByUsernamePassword`, see [AcquireTokenByUsernamePasswordParameterBuilder](#).

Next steps

Move on to the next article in this scenario, [Call a web API from the desktop app](#).

Desktop app that calls web APIs: Acquire a token using Device Code flow

4/12/2022 • 6 minutes to read • [Edit Online](#)

If you're writing a command-line tool that doesn't have web controls, and you can't or don't want to use the previous flows, use the device code flow.

Device code flow

Interactive authentication with Azure AD requires a web browser. For more information, see [Usage of web browsers](#). To authenticate users on devices or operating systems that don't provide a web browser, device code flow lets the user use another device such as a computer or a mobile phone to sign in interactively. By using the device code flow, the application obtains tokens through a two-step process that's designed for these devices or operating systems. Examples of such applications are applications that run on IoT or command-line tools (CLI). The idea is that:

1. Whenever user authentication is required, the app provides a code for the user. The user is asked to use another device, such as an internet-connected smartphone, to go to a URL, for instance, <https://microsoft.com/devicelogin>. Then the user is prompted to enter the code. That done, the web page leads the user through a normal authentication experience, which includes 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.

Use it

- [.NET](#)
- [Java](#)
- [macOS](#)
- [Node.js](#)
- [Python](#)

`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 receives the `DeviceCodeResult`.

The following sample code presents the synopsis of most current cases, with explanations of the kind of exceptions you can get and their mitigation. For a fully functional code sample, see [active-directory-dotnetcore-devicecodeflow-v2](#) on GitHub.

```
private const string ClientId = "<client_guid>";
private const string Authority = "https://login.microsoftonline.com/contoso.com";
private readonly string[] scopes = new string[] { "User.read" };
```

```

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 static 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 authoriy 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  
}  
}
```

Next steps

Move on to the next article in this scenario, [Call a web API from the desktop app](#).

Desktop app that calls web APIs: Call a web API

4/12/2022 • 4 minutes to read • [Edit Online](#)

Now that you have a token, you can call a protected web API.

Call a web API

- [.NET](#)
- [Java](#)
- [MacOS](#)
- [Node.js](#)
- [Python](#)

AuthenticationResult properties in MSAL.NET

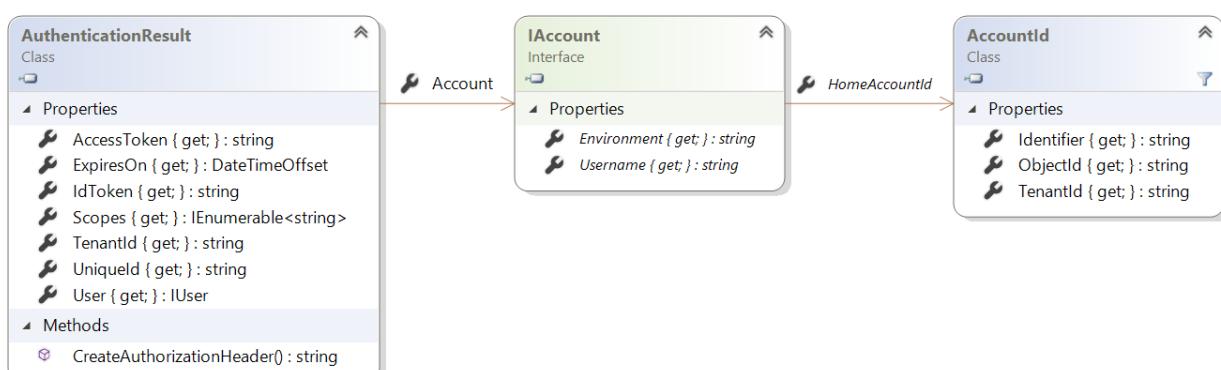
The methods to acquire tokens return `AuthenticationResult`. For async methods, `Task<AuthenticationResult>` returns.

In MSAL.NET, `AuthenticationResult` exposes:

- `AccessToken` for the web API to access resources. This parameter is a string, usually a Base-64-encoded JWT. 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. Writing code that depends on access token content on the client is one of the biggest sources of errors and client logic breaks. For more information, see [Access tokens](#).
- `IdToken` for the user. This parameter is an encoded JWT. For more information, see [ID tokens](#).
- `ExpiresOn` tells the date and time when the token expires.
- `TenantId` contains the tenant in which the user was found. For guest users in Azure Active Directory (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 an account through the `IAccount` interface. This breaking change provides the right semantics. The same user can have several accounts, in different Azure AD directories. Also, MSAL.NET provides better information in the case of guest scenarios because 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 with the properties shown in the following table.

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. They're not Base 64 encoded.

The `IAccount` interface represents information about a single account. The same user can be present in different tenants, which means that a user can have multiple accounts. Its members are shown in the following table.

PROPERTY	DESCRIPTION
<code>Username</code>	A string that contains the displayable value in UserPrincipalName (UPN) format, for example, john.doe@contoso.com. This string can be null, unlike <code>HomeAccountId</code> and <code>HomeAccountId.Identifier</code> , which 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 that contains 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. Here, the value is only the host.
<code>HomeAccountId</code>	The account ID of the home account for the user. This property uniquely identifies the user across Azure AD tenants.

Use the token to call a protected API

After `AuthenticationResult` is returned by MSAL in `result`, add it to the HTTP authorization header before you make the call to access the protected web API.

```
httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer",
result.AccessToken);

// Call the web API.
HttpResponseMessage response = await _httpClient.GetAsync(apiUri);
...
}
```

Next steps

Move on to the next article in this scenario, [Move to production](#).

Desktop app that calls web APIs: Move to production

4/12/2022 • 2 minutes to read • [Edit Online](#)

In this article, you learn how to move your desktop app that calls web APIs to production.

Handle errors in desktop applications

In the different flows, you've learned how to handle the errors for the silent flows, as shown in the code snippets. You've also seen that there are cases where interaction is needed, as in incremental consent and conditional access.

Have the user consent upfront for several resources

NOTE

Getting consent for several resources works for the Microsoft identity platform but not for Azure Active Directory (Azure AD) B2C. Azure AD B2C supports only admin consent, not user consent.

You can't get a token for several resources at once with the Microsoft identity platform. The `scopes` parameter can contain scopes for only a single resource. You can ensure that the user pre-consents to several resources by using the `extraScopesToConsent` parameter.

For instance, you might have two resources that have two scopes each:

- `https://mytenant.onmicrosoft.com/customerapi` with the scopes `customer.read` and `customer.write`
- `https://mytenant.onmicrosoft.com/vendorapi` with the scopes `vendor.read` and `vendor.write`

In this example, use the `.WithExtraScopesToConsent` 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())
    .WithExtraScopesToConsent(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 gets you an access token for the first web API.

When calling the second web API, call the `AcquireTokenSilent` API.

```
AcquireTokenSilent(scopesForVendorApi, accounts.FirstOrDefault()).ExecuteAsync();
```

Microsoft personal account requires reconsent each time the app runs

For Microsoft personal account users, reprompting for consent on each native client (desktop or mobile app) call to `authorize` is the intended behavior. Native client identity is inherently insecure, which is contrary to confidential client application identity. Confidential client applications 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.

Enable logging

To help in debugging and authentication failure troubleshooting scenarios, the Microsoft Authentication Library provides built-in logging support. Logging is covered in the following articles:

- [Logging in MSAL.NET](#)
- [Logging in MSAL for Android](#)
- [Logging in MSAL.js](#)
- [Logging in MSAL for iOS/macOS](#)
- [Logging in MSAL for Java](#)
- [Logging in MSAL for Python](#)

Here are some suggestions for data collection:

- Users might ask for help when they have problems. A best practice is to capture and temporarily store

logs. Provide a location where users can upload the logs. MSAL provides logging extensions to capture detailed information about authentication.

- If telemetry is available, enable it through MSAL to gather data about how users sign in to your app.

Validate your integration

Test your integration by following the [Microsoft identity platform integration checklist](#).

Build for resilience

Learn how to increase resiliency in your app. For details, see [Increase resilience of authentication and authorization applications you develop](#)

Next steps

To try out additional samples, see [Desktop public client applications](#).

Quickstart: Sign in users and call the Microsoft Graph API from a mobile application

4/12/2022 • 17 minutes to read • [Edit Online](#)

In this quickstart, you download and run a code sample that demonstrates how an Android application can sign in users and get an access token to call the Microsoft Graph API.

See [How the sample works](#) for an illustration.

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

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- Android Studio
- Android 16+

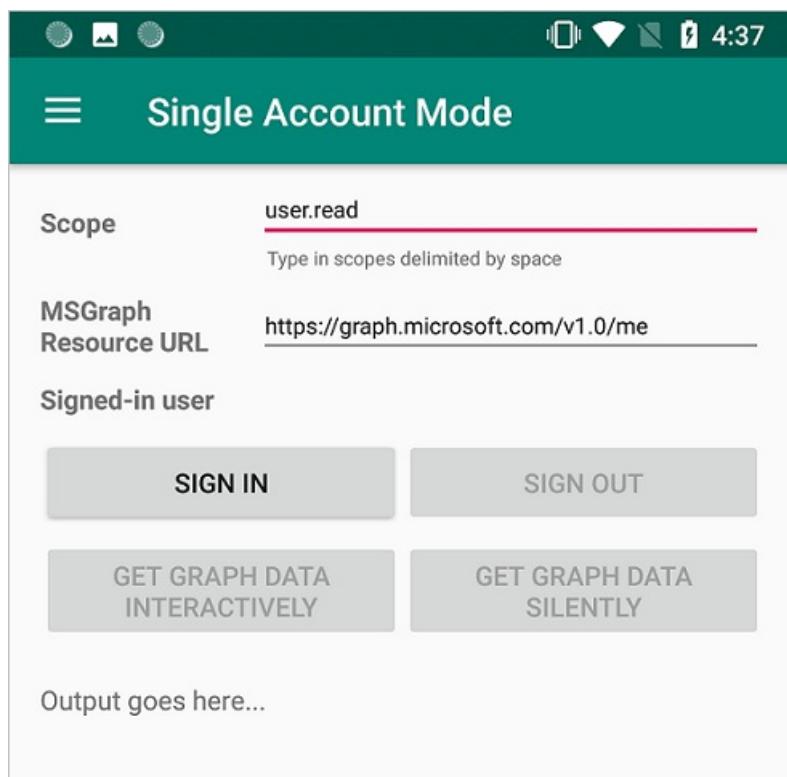
Step 1: Get the sample app

[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.



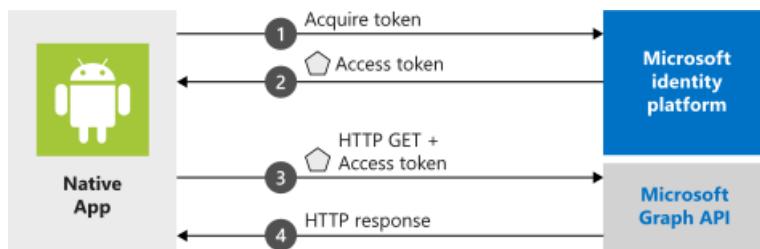
Use the app menu to change between single and multiple account modes.

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
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**:

```
dependencies {
    ...
    implementation 'com.microsoft.identity.client:msal:2.+'
    ...
}
```

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

You must also add references to maven to the **allprojects > repositories** portion of the **build.gradle (Module: app)** like so:

```
allprojects {
    repositories {
        mavenCentral()
        google()
        mavenLocal()
        maven {
            url 'https://pkgs.dev.azure.com/MicrosoftDeviceSDK/DuoSDK-Public/_packaging/Duo-SDK-Feed/maven/v1'
        }
        maven {
            name "vsts-maven-adal-android"
            url "https://identitydivision.pkgs.visualstudio.com/_packaging/AndroidADAL/maven/v1"
            credentials {
                username System.getenv("ENV_VSTS_MVN_ANDROIDADAL_USERNAME") != null ?
System.getenv("ENV_VSTS_MVN_ANDROIDADAL_USERNAME") : project.findProperty("vstsUsername")
                password System.getenv("ENV_VSTS_MVN_ANDROIDADAL_ACESSTOKEN") != null ?
System.getenv("ENV_VSTS_MVN_ANDROIDADAL_ACESSTOKEN") : project.findProperty("vstsMavenAccessToken")
            }
        }
        jcenter()
    }
}
```

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(getContext(),
    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.

Sigining 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.azuresamples.msalandroidapp/1wIqXsqBj7w%2Bh11ZifsnqwgyKrY%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%2Bh11ZifsnqwyKrY%3D",
  "account_mode" : "MULTIPLE",
  "broker_redirect_uri_registered": true,
  "authorities" : [
    {
      "type": "AAD",
      "audience": {
        "type": "AzureADandPersonalMicrosoftAccount",
        "tenant_id": "common"
      }
    }
  ]
}
```

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

Move on to the Android tutorial in which you build an Android app that gets an access token from the Microsoft identity platform and uses it to call the Microsoft Graph API.

[Tutorial: Sign in users and call the Microsoft Graph from an Android application](#)

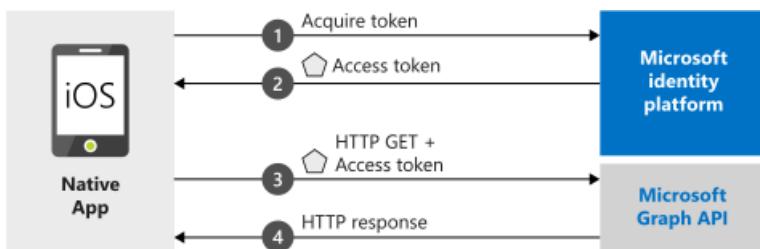
In this quickstart, you download and run a code sample that demonstrates how a native iOS or macOS application can sign in users and get an access token to call the Microsoft Graph API.

The quickstart applies to both iOS and macOS apps. Some steps are needed only for iOS apps and will be indicated as such.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- XCode 10+
- iOS 10+
- macOS 10.12+

How the sample works



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 [Azure portal - App registrations](#) quickstart experience.
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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application. Users of your app might see this name, and you can change it later.
6. Select **Register**.
7. Under **Manage**, select **Authentication > Add Platform > iOS**.
8. Enter the **Bundle Identifier** for your application. The bundle identifier is 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.
9. Select **Configure** and save the **MSAL Configuration** details for later in this quickstart.
10. Select **Done**.

Step 2: Download the sample project

- [Download the code sample for iOS](#)
- [Download the code sample for macOS](#)

Step 3: Install dependencies

1. Extract the zip file.
2. 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. 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. If you're building an app for [Azure AD national clouds](#), replace the line starting with 'let kGraphEndpoint' and 'let kAuthority' with correct endpoints. For global access, use default values:

```
let kGraphEndpoint = "https://graph.microsoft.com/"  
let kAuthority = "https://login.microsoftonline.com/common"
```

4. Other endpoints are documented [here](#). For example, to run the quickstart with Azure AD Germany, use following:

```
let kGraphEndpoint = "https://graph.microsoft.de/"  
let kAuthority = "https://login.microsoftonline.de/common"
```

5. Open the project settings. In the **Identity** section, enter the **Bundle Identifier** that you entered into the portal.
6. Right-click **Info.plist** and select **Open As > Source Code**.
7. Under the dict root node, replace `Enter_the_bundle_Id_Here` with the **Bundle Id** that you used in the portal. Notice the `msauth.` prefix in the string.

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

8. Build and 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:	DESCRIPTION
clientId	The Application ID from the application registered in portal.azure.com
authority	The Microsoft identity platform. In most of cases this will be https://login.microsoftonline.com/common
redirectUri	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>microsoftauthv2</string>
    <string>microsoftauthv3</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:	DESCRIPTION
<code>scopes</code>	Contains the scopes being requested (that is, <code>["user.read"]</code> for Microsoft Graph or <code>["<Application ID URL>/scope"]</code> for custom web APIs (<code>api://<Application ID>/access_as_user</code>)

`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.

```

self.applicationContext!.getCurrentAccount(with: nil) { (currentAccount, previousAccount, error) in

    guard let account = currentAccount else {
        return
    }

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

```

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

WHERE:	DESCRIPTION
<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>accountsFromDeviceForParameters:completionBlock:</code> and passing correct <code>accountIdentifier</code>

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

Move on to the step-by-step tutorial in which you build an iOS or macOS app that gets an access token from the Microsoft identity platform and uses it to call the Microsoft Graph API.

[Tutorial: Sign in users and call Microsoft Graph from an iOS or macOS app](#)

Tutorial: Sign in users and call the Microsoft Graph API from an Android application

4/12/2022 • 10 minutes to read • [Edit Online](#)

In this tutorial, you build an Android app that integrates with the Microsoft identity platform to sign in users and get an access token to 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.

In this tutorial:

- Create an Android app project in *Android Studio*
- Register the app in the Azure portal
- Add code to support user sign-in and sign-out
- Add code to call the Microsoft Graph API
- Test the app

Prerequisites

- Android Studio 3.5+

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).

This tutorial demonstrates simplified examples of working with MSAL for Android. For simplicity, it uses Single Account Mode only. To explore more complex scenarios, see a completed [working code sample](#) on GitHub.

Create a project

If you do not already have an Android application, follow these steps to set up a new project.

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`.

Integrate with the Microsoft Authentication Library

Register your application

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application. Users of your app might see this name, and you can change it later.
6. Select **Register**.
7. Under **Manage**, select **Authentication > Add a platform > Android**.
8. Enter your project's **Package Name**. If you downloaded the code, this value is
`com.azure.samples.msalandroidapp`.
9. In the **Signature hash** section of the **Configure your Android app** page, select **Generating a development Signature Hash**. and copy the KeyTool command to use for your platform.

KeyTool.exe is installed as part of the Java Development Kit (JDK). You must also install the OpenSSL tool to execute the KeyTool command. Refer to the [Android documentation on generating a key](#) for more information.

10. Enter the **Signature hash** generated by KeyTool.
11. Select **Configure** and save the **MSAL Configuration** that appears in the **Android configuration** page so you can enter it when you configure your app later.
12. Select **Done**.

Configure your application

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_single_account.json` and paste the MSAL Configuration that you saved earlier.

Below the redirect URI, paste:

```
"account_mode" : "SINGLE",
```

Your config file should resemble this example:

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

This tutorial only demonstrates how to configure an app in Single Account mode. View the documentation for more information on [single vs. multiple account mode](#) and [configuring your app](#)

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. Ensure that there is a leading `/` at the beginning of your Signature Hash.

The "Package Name" you will replace the `android:host` value with should look similar to:

`com.azure.samples.msalandroidapp`. The "Signature Hash" you will replace your `android:path` value with should look similar to: `/1wIqXSqBj7w+h11ZifsqnwggyKrY=`.

You will also be able to find these values in the Authentication blade of your app registration. Note that your redirect URI will look similar to:

`msauth://com.azure.samples.msalandroidapp/1wIqXSqBj7w%2Bh11ZifsqnwggyKrY%3D`. While the Signature Hash is URL-encoded at the end of this value, the Signature Hash should **not** be URL-encoded in your `android:path` value.

Use MSAL

[Add MSAL to your project](#)

1. In the Android Studio project window, navigate to **app > build.gradle** and add the following:

```
apply plugin: 'com.android.application'

allprojects {
    repositories {
        mavenCentral()
        google()
        mavenLocal()
        maven {
            url 'https://pkgs.dev.azure.com/MicrosoftDeviceSDK/DuoSDK-Public/_packaging/Duo-SDK-Feed/maven/v1'
        }
        maven {
            name "vsts-maven-adal-android"
            url "https://identitydivision.pkgs.visualstudio.com/_packaging/AndroidADAL/maven/v1"
            credentials {
                username System.getenv("ENV_VSTS_MVN_ANDROIDADAL_USERNAME") != null ?
                    System.getenv("ENV_VSTS_MVN_ANDROIDADAL_USERNAME") : project.findProperty("vstsUsername")
                password System.getenv("ENV_VSTS_MVN_ANDROIDADAL_ACESSTOKEN") != null ?
                    System.getenv("ENV_VSTS_MVN_ANDROIDADAL_ACESSTOKEN") : project.findProperty("vstsMavenAccessToken")
            }
        }
        jcenter()
    }
}
dependencies{
    implementation 'com.microsoft.identity.client:msal:2.+'
    implementation 'com.microsoft.graph:microsoft-graph:1.5.+'
}
packagingOptions{
    exclude("META-INF/jersey-module-version")
}
```

More on the Microsoft Graph SDK

Required Imports

Add the following to the top of **app > src > main > java > com.example(yourapp) > MainActivity.java**

```
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import androidx.appcompat.app.AppCompatActivity;
import com.google.gson.JsonObject;
import com.microsoft.graph.authentication.IAuthenticationProvider; //Imports the Graph sdk Auth interface
import com.microsoft.graph.concurrency.ICallback;
import com.microsoft.graph.core.ClientException;
import com.microsoft.graph.http.IHttpRequest;
import com.microsoft.graph.models.extensions.*;
import com.microsoft.graph.requests.extensions.GraphServiceClient;
import com.microsoft.identity.client.AuthenticationCallback; // Imports MSAL auth methods
import com.microsoft.identity.client.*;
import com.microsoft.identity.client.exception.*;
```

Instantiate PublicClientApplication

Initialize Variables

```
private final static String[] SCOPES = {"Files.Read"};
/* Azure AD v2 Configs */
final static String AUTHORITY = "https://login.microsoftonline.com/common";
private ISingleAccountPublicClientApplication mSingleAccountApp;

private static final String TAG = MainActivity.class.getSimpleName();

/* UI & Debugging Variables */
Button signInButton;
Button signOutButton;
Button callGraphApiInteractiveButton;
Button callGraphApiSilentButton;
TextView logTextView;
TextView currentUserTextView;
```

onCreate

Inside the `MainActivity` class, refer to the following `onCreate()` method to instantiate MSAL using the `SingleAccountPublicClientApplication`.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    initializeUI();

    PublicClientApplication.createSingleAccountPublicClientApplication(getApplicationContext(),
        R.raw.auth_config_single_account, new
    IPublicClientApplication.ISingleAccountApplicationCreatedListener() {
        @Override
        public void onCreated(ISingleAccountPublicClientApplication application) {
            mSingleAccountApp = application;
            loadAccount();
        }
        @Override
        public void onError(MsalException exception) {
            displayError(exception);
        }
    });
}
```

loadAccount

```
//When app comes to the foreground, load existing account to determine if user is signed in
private void loadAccount() {
    if (mSingleAccountApp == null) {
        return;
    }

    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);
        }
    });
}
```

initializeUI

Listen to buttons and call methods or log errors accordingly.

```

private void initializeUI(){
    signInButton = findViewById(R.id.signIn);
    callGraphApiSilentButton = findViewById(R.id.callGraphSilent);
    callGraphApiInteractiveButton = findViewById(R.id.callGraphInteractive);
    signOutButton = findViewById(R.id.clearCache);
    logTextView = findViewById(R.id.txt_log);
    currentUserTextView = findViewById(R.id.current_user);

    //Sign in user
    signInButton.setOnClickListener(new View.OnClickListener(){
        public void onClick(View v) {
            if (mSingleAccountApp == null) {
                return;
            }
            mSingleAccountApp.signIn(MainActivity.this, null, SCOPES, getAuthInteractiveCallback());
        }
    });

    //Sign out user
    signOutButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if (mSingleAccountApp == null){
                return;
            }
            mSingleAccountApp.signOut(new ISingleAccountPublicClientApplication.SignOutCallback() {
                @Override
                public void onSignOut() {
                    updateUI(null);
                    performOperationOnSignOut();
                }
                @Override
                public void onError(@NonNull MsalException exception){
                    displayError(exception);
                }
            });
        }
    });

    //Interactive
    callGraphApiInteractiveButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if (mSingleAccountApp == null) {
                return;
            }
            mSingleAccountApp.acquireToken(MainActivity.this, SCOPES, getAuthInteractiveCallback());
        }
    });

    //Silent
    callGraphApiSilentButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if (mSingleAccountApp == null){
                return;
            }
            mSingleAccountApp.acquireTokenSilentAsync(SCOPES, AUTHORITY, getAuthSilentCallback());
        }
    });
}

```

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.

getAuthInteractiveCallback

Callback used for interactive requests.

```
private AuthenticationCallback getAuthInteractiveCallback() {
    return new AuthenticationCallback() {
        @Override
        public void onSuccess(IAuthenticationResult authenticationResult) {
            /* Successfully got a token, use it to call a protected resource - MSGraph */
            Log.d(TAG, "Successfully authenticated");
            /* Update UI */
            updateUI(authenticationResult.getAccount());
            /* call graph */
            callGraphAPI(authenticationResult);
        }

        @Override
        public void onError(MsalException exception) {
            /* Failed to acquireToken */
            Log.d(TAG, "Authentication failed: " + exception.toString());
            displayError(exception);
        }
        @Override
        public void onCancel() {
            /* User canceled the authentication */
            Log.d(TAG, "User cancelled login.");
        }
    };
}
```

getAuthSilentCallback

Callback used for silent requests

```
private SilentAuthenticationCallback getAuthSilentCallback() {
    return new SilentAuthenticationCallback() {
        @Override
        public void onSuccess(IAuthenticationResult authenticationResult) {
            Log.d(TAG, "Successfully authenticated");
            callGraphAPI(authenticationResult);
        }

        @Override
        public void onError(MsalException exception) {
            Log.d(TAG, "Authentication failed: " + exception.toString());
            displayError(exception);
        }
    };
}
```

Call Microsoft Graph API

The following code demonstrates how to call the Graph API using the Graph SDK.

callGraphAPI

```

private void callGraphAPI(IAuthenticationResult authenticationResult) {

    final String accessToken = authenticationResult.getAccessToken();

    IGraphServiceClient graphClient =
        GraphServiceClient
            .builder()
            .authenticationProvider(new IAuthenticationProvider() {
                @Override
                public void authenticateRequest(IHttpRequest request) {
                    Log.d(TAG, "Authenticating request," + request.getRequestUrl());
                    request.addHeader("Authorization", "Bearer " + accessToken);
                }
            })
            .buildClient();

    graphClient
        .me()
        .drive()
        .buildRequest()
        .get(new ICallback<Drive>() {
            @Override
            public void success(final Drive drive) {
                Log.d(TAG, "Found Drive " + drive.id);
                displayGraphResult(drive.getRawObject());
            }

            @Override
            public void failure(ClientException ex) {
                displayError(ex);
            }
        });
}

```

Add UI

Activity

If you would like to model your UI off this tutorial, the following methods provide a guide to updating text and listening to buttons.

updateUI

Enable/disable buttons based on sign-in state and set text.

```

private void updateUI(@Nullable final IAccount account) {
    if (account != null) {
        signInButton.setEnabled(false);
        signOutButton.setEnabled(true);
        callGraphApiInteractiveButton.setEnabled(true);
        callGraphApiSilentButton.setEnabled(true);
        currentUserTextView.setText(account.getUsername());
    } else {
        signInButton.setEnabled(true);
        signOutButton.setEnabled(false);
        callGraphApiInteractiveButton.setEnabled(false);
        callGraphApiSilentButton.setEnabled(false);
        currentUserTextView.setText("");
        logTextView.setText("");
    }
}

```

displayError

```
private void displayError(@NonNull final Exception exception) {
    logTextView.setText(exception.toString());
}
```

displayGraphResult

```
private void displayGraphResult(@NonNull final JSONObject graphResponse) {
    logTextView.setText(graphResponse.toString());
}
```

performOperationOnSignOut

Method to update text in UI to reflect sign out.

```
private void performOperationOnSignOut() {
    final String signOutText = "Signed Out.";
    currentUserTextView.setText("");
    Toast.makeText(getApplicationContext(), signOutText, Toast.LENGTH_SHORT)
        .show();
}
```

Layout

Sample `activity_main.xml` file to display buttons and text boxes.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#FFFFFF"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:paddingTop="5dp"
        android:paddingBottom="5dp"
        android:weightSum="10">

        <Button
            android:id="@+id/signIn"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="5"
            android:gravity="center"
            android:text="Sign In"/>

        <Button
            android:id="@+id/clearCache"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="5"
            android:gravity="center"
            android:text="Sign Out"
            android:enabled="false"/>

    </LinearLayout>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```

        android:gravity="center"
        android:orientation="horizontal">

        <Button
            android:id="@+id/callGraphInteractive"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="5"
            android:text="Get Graph Data Interactively"
            android:enabled="false"/>

        <Button
            android:id="@+id/callGraphSilent"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="5"
            android:text="Get Graph Data Silently"
            android:enabled="false"/>
    </LinearLayout>

    <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"/>

    <TextView
        android:id="@+id/current_user"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_marginTop="20dp"
        android:layout_weight="0.8"
        android:text="Account info goes here... " />

    <TextView
        android:id="@+id/txt_log"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_marginTop="20dp"
        android:layout_weight="0.8"
        android:text="Output goes here... " />
</LinearLayout>
```

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. PR 4

Consent

The first time any user signs into your app, they will be prompted by Microsoft identity to consent to the permissions requested. Some Azure AD tenants have disabled user consent which requires admins to consent on behalf of all users. To support this scenario, you will either need to create your own tenant or receive admin consent.

Clean up resources

When no longer needed, delete the app object that you created in the [Register your application](#) step.

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

Learn more about building mobile apps that call protected web APIs in our multi-part scenario series.

[Scenario: Mobile application that calls web APIs](#)

Tutorial: Use shared-device mode in your Android application

4/12/2022 • 7 minutes to read • [Edit Online](#)

In this tutorial, Android developers and Azure Active Directory (Azure AD) tenant administrators learn about the code, Authenticator app, and tenant settings required to enable shared-device mode for an Android app.

In this tutorial:

- Download a code sample
- Enable and detect shared-device mode
- Detect single or multiple account mode
- Detect a user switch, and enable global sign-in and sign-out
- Set up tenant and register the application in the Azure portal
- Set up an Android device in shared-device mode
- Run the sample app

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).

Developer guide

This section of the tutorial provides developer guidance for implementing shared-device mode in an Android application using the Microsoft Authentication Library (MSAL). See the [MSAL Android tutorial](#) to see how to integrate MSAL with your Android app, sign in a user, call Microsoft graph, and sign out a user.

Download the sample

Clone the [sample application](#) from GitHub. The sample has the capability to work in [single or multi account mode](#).

Add the MSAL SDK to your local Maven repository

If you're not using the sample app, add the MSAL library as a dependency in your build.gradle file, like so:

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

Configure your app to use shared-device mode

Refer to the [configuration documentation](#) for more information on setting up your config file.

Set `"shared_device_mode_supported"` to `true` in your MSAL configuration file.

You may not be planning to support multiple-account mode. That could be if you're not using a shared device, and the user can sign into the app with more than one account at the same time. If so, set `"account_mode"` to `"SINGLE"`. This guarantees that your app will always get `ISingleAccountPublicClientApplication`, and significantly simplifies your MSAL integration. The default value of `"account_mode"` is `"MULTIPLE"`, so it is important to change this value in the config file if you're using `"single account"` mode.

Here's an example of the auth_config.json file included in the `app>main>res>raw` directory of the sample app:

```
{
  "client_id": "Client ID after app registration at https://aka.ms/MobileAppReg",
  "authorization_user_agent": "DEFAULT",
  "redirect_uri": "Redirect URI after app registration at https://aka.ms/MobileAppReg",
  "account_mode": "SINGLE",
  "broker_redirect_uri_registered": true,
  "shared_device_mode_supported": true,
  "authorities": [
    {
      "type": "AAD",
      "audience": {
        "type": "AzureADandPersonalMicrosoftAccount",
        "tenant_id": "common"
      }
    }
  ]
}
```

Detect shared-device mode

Shared-device mode allows you to configure Android devices to be shared by multiple employees, while providing Microsoft Identity backed management of the device. Employees can sign in to their devices and access customer information quickly. When they are finished with their shift or task, they will be able to sign-out of all apps on the shared device with a single click and the device will be immediately ready for the next employee to use.

Use `isSharedDevice()` to determine if an app is running on a device that is in shared-device mode. Your app could use this flag to determine if it should modify UX accordingly.

Here's a code snippet that shows how you could use `isSharedDevice()`. It's from the `SingleAccountModeFragment` class in the sample app:

```
deviceModeTextView.setText(mSingleAccountApp.isSharedDevice() ? "Shared" : "Non-Shared");
```

Initialize the PublicClientApplication object

If you set `"account_mode": "SINGLE"` in the MSAL config file, you can safely cast the returned application object as an `ISingleAccountPublicClientApplication`.

```
private ISingleAccountPublicClientApplication mSingleAccountApp;

/*Configure your sample app and save state for this activity*/
PublicClientApplication.create(this.getApplicationContext(),
  R.raw.auth_config,
  new PublicClientApplication.ApplicationCreatedListener(){
    @Override
    public void onCreated(IPublicClientApplication application){
      mSingleAccountApp = (ISingleAccountPublicClientApplication)application;
      loadAccount();
    }
    @Override
    public void onError(MsalException exception{
      /*Fail to initialize PublicClientApplication */
    }
  });
});
```

Detect single vs. multiple account mode

If you're writing an app that will only be used for first-line workers on a shared device, we recommend you write your app to only support single-account mode. This includes most applications that are task focused such as

medical records apps, invoice apps, and most line-of-business apps. This will simplify your development as many features of the SDK won't need to be accommodated.

If your app supports multiple accounts as well as shared device mode, you must perform a type check and cast to the appropriate interface as shown below.

```
private IPublicClientApplication mApplication;

if (mApplication instanceof IMultipleAccountPublicClientApplication) {
    IMultipleAccountPublicClientApplication multipleAccountApplication =
(IMultipleAccountPublicClientApplication) mApplication;
    ...
} else if (mApplication instanceof ISingleAccountPublicClientApplication) {
    ISingleAccountPublicClientApplication singleAccountApplication =
(ISingleAccountPublicClientApplication) mApplication;
    ...
}
```

Get the signed in user and determine if a user has changed on the device

The `loadAccount` method retrieves the account of the signed in user. The `onAccountChanged` method determines if the signed-in user has changed, and if so, clean up:

```
private void loadAccount()
{
    mSingleAccountApp.getCurrentAccountAsync(new
ISingleAccountPublicClientApplication.CurrentAccountCallback()
{
    @Override
    public void onAccountLoaded(@Nullable IAccount activeAccount)
    {
        if (activeAccount != null)
        {
            signedInUser = activeAccount;

mSingleAccountApp.acquireTokenSilentAsync(SCOPES, "http://login.microsoftonline.com/common", getAuthSilentCall
back());
        }
    }
    @Override
    public void onAccountChanged(@Nullable IAccount priorAccount, @Nullable IAccount currentAccount)
    {
        if (currentAccount == null)
        {
            //Perform a cleanup task as the signed-in account changed.
            updateSignedOutUI();
        }
    }
    @Override
    public void onError(@NonNull Exception exception)
    {
    }
}
}
```

Globally sign in a user

The following signs in a user across the device to other apps that use MSAL with the Authenticator App:

```
private void onSignInClicked()
{
    mSingleAccountApp.signIn(getActivity(), SCOPES, null, getAuthInteractiveCallback());
}
```

Globally sign out a user

The following removes the signed-in account and clears cached tokens from not only the app but also from the device that is in shared device mode:

```
private void onSignOutClicked()
{
    mSingleAccountApp.signOut(new ISingleAccountPublicClientApplication.SignOutCallback()
    {
        @Override
        public void onSignOut()
        {
            updateSignedOutUI();
        }
        @Override
        public void onError(@NonNull MsalException exception)
        {
            /*failed to remove account with an exception*/
        }
    });
}
```

Administrator guide

The following steps describe setting up your application in the Azure portal and putting your device into shared-device mode.

Register your application in Azure Active Directory

First, register your application within your organizational tenant. Then provide these values below in auth_config.json in order for your application to run correctly.

For information on how to do this, refer to [Register your application](#).

NOTE

When you register your app, please use the quickstart guide on the left-hand side and then select **Android**. This will lead you to a page where you'll be asked to provide the **Package Name** and **Signature Hash** for your app. These are very important to ensure your app configuration will work. You'll then receive a configuration object that you can use for your app that you'll cut and paste into your auth_config.json file.

Screenshot of the Microsoft Azure portal showing the "Configure your Android app" quickstart for the FirstlineWorkerApp.

The left sidebar shows the navigation path: Home > FirstlineWorkerApp - Quickstart > Quickstart.

The main content area is titled "Configure your Android app". It includes a feedback survey bar at the top.

Quickstart: Sign in users and call APIs from an Android app

This quickstart uses a code sample to demonstrate how an Android app can sign in users through 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.

Diagram: A flowchart showing the process:

- 1. Acquire token (Native App → Microsoft Identity Platform)
- 2. Get access token (Microsoft Identity Platform → Native App)
- 3. Acquire token (Native App → Microsoft Graph API)
- 4. Get response (Microsoft Graph API → Native App)

Note:

Prerequisites:

- Android Studio
- Android 16+

Step 1: Configure your application in the Azure portal

For the code sample for this quickstart to work, you need to add a redirect URI to your application registration in the Azure portal.

[Make these changes for me](#)

Step 2: Download the project

[Download the code sample](#)

Step 3: Configure your project

[Make updates](#) [Cancel](#)

You should select **Make this change for me** and then provide the values the quickstart asks for in the Azure portal. When that's done, we will generate all the configuration files you need.

FirstlineWorkerApp - Quickstart

 Got a second? We would love to hear your feedback on quickstarts. →

Step 3: Configure your project

- Extract and open the Project in Android Studio.
- Inside app > src > main > res > raw, open auth_config_multiple_account.json and replace it with the following code:

```
{
  "client_id": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "authorization_user_agent": "DEFAULT",
  "redirect_uri": "msauth://com.consotos.myapp/xxxxxxxxxxxx%xxxxxxxxx%xxxxxx%xx",
  "account_mode": "MULTIPLE",
  "broker_redirect_uri_registered": true,
  "authorities": [
    {
      "type": "AAD",
      "audience": {
        "type": "AzureADMyOrg",
        "tenant_id": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
      }
    }
  ]
}
```

- Inside app > src > main > res > raw, open auth_config_single_account.json and replace it with the following code:

```
{
  "client_id": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "authorization_user_agent": "DEFAULT",
  "redirect_uri": "msauth://com.consotos.myapp/xxxxxxxxxxxx%xxxxxxxxx%xxxxxx%xx",
  "account_mode": "SINGLE",
  "broker_redirect_uri_registered": true,
  "authorities": [
    {
      "type": "AAD",
      "audience": {
        "type": "AzureADMyOrg",
        "tenant_id": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
      }
    }
  ]
}
```

- Inside app > src > main, open AndroidManifest.xml.
- In the manifest\application node, replace the activity android:name="com.microsoft.identity.client.BrowserTabActivity" node with the following:

```
<!--Intent filter to catch Microsoft's callback 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" />
    <!--
        Add in your scheme/host from registered redirect URI
        note that the leading "/" is required for android:path
    -->
    <data
      android:host="com.consotos.myapp"
      android:path="/2pmj914rSx0yEb/viWBkE/ZQrk="
      android:scheme=" msauth" />
  </intent-filter>
</activity>
```

- Run the app!

Set up a tenant

For testing purposes, set up the following in your tenant: at least two employees, one Cloud Device Administrator, and one Global Administrator. In the Azure portal, set the Cloud Device Administrator by modifying Organizational Roles. In the Azure portal, access your Organizational Roles by selecting **Azure Active Directory > Roles and Administrators > Cloud Device Administrator**. Add the users that can put a device into shared mode.

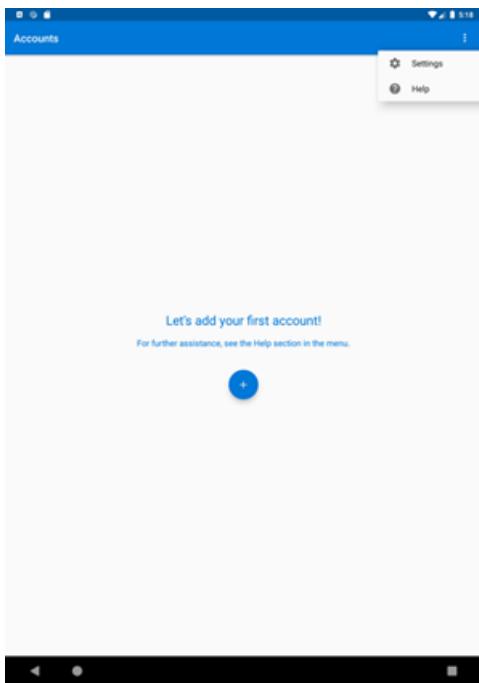
Set up an Android device in shared mode

Download the Authenticator App

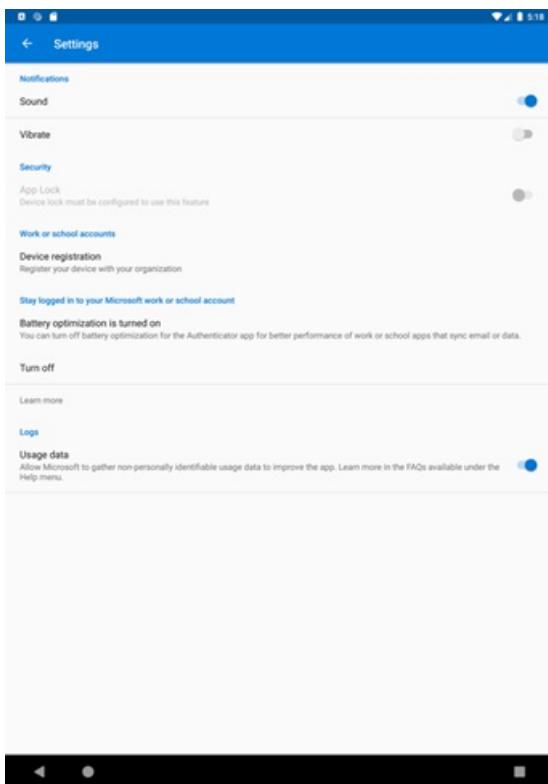
Download the Microsoft Authenticator App from the Google Play store. If you already have the app downloaded, ensure that it is the latest version.

Authenticator app settings & registering the device in the cloud

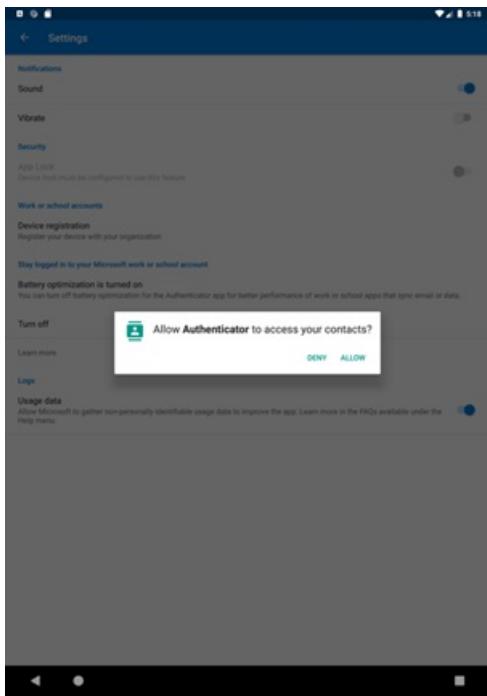
Launch the Authenticator App and navigate to main account page. Once you see the **Add Account** page, you're ready to make the device shared.



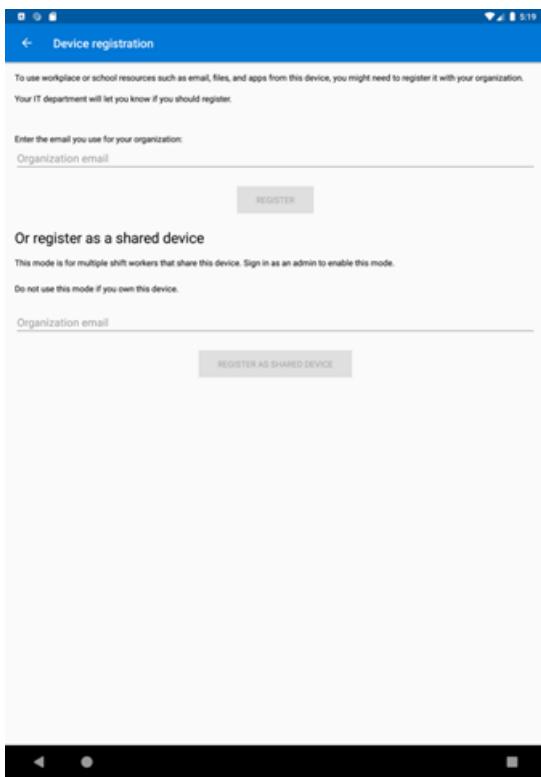
Go to the **Settings** pane using the right-hand menu bar. Select **Device Registration** under **Work & School accounts**.

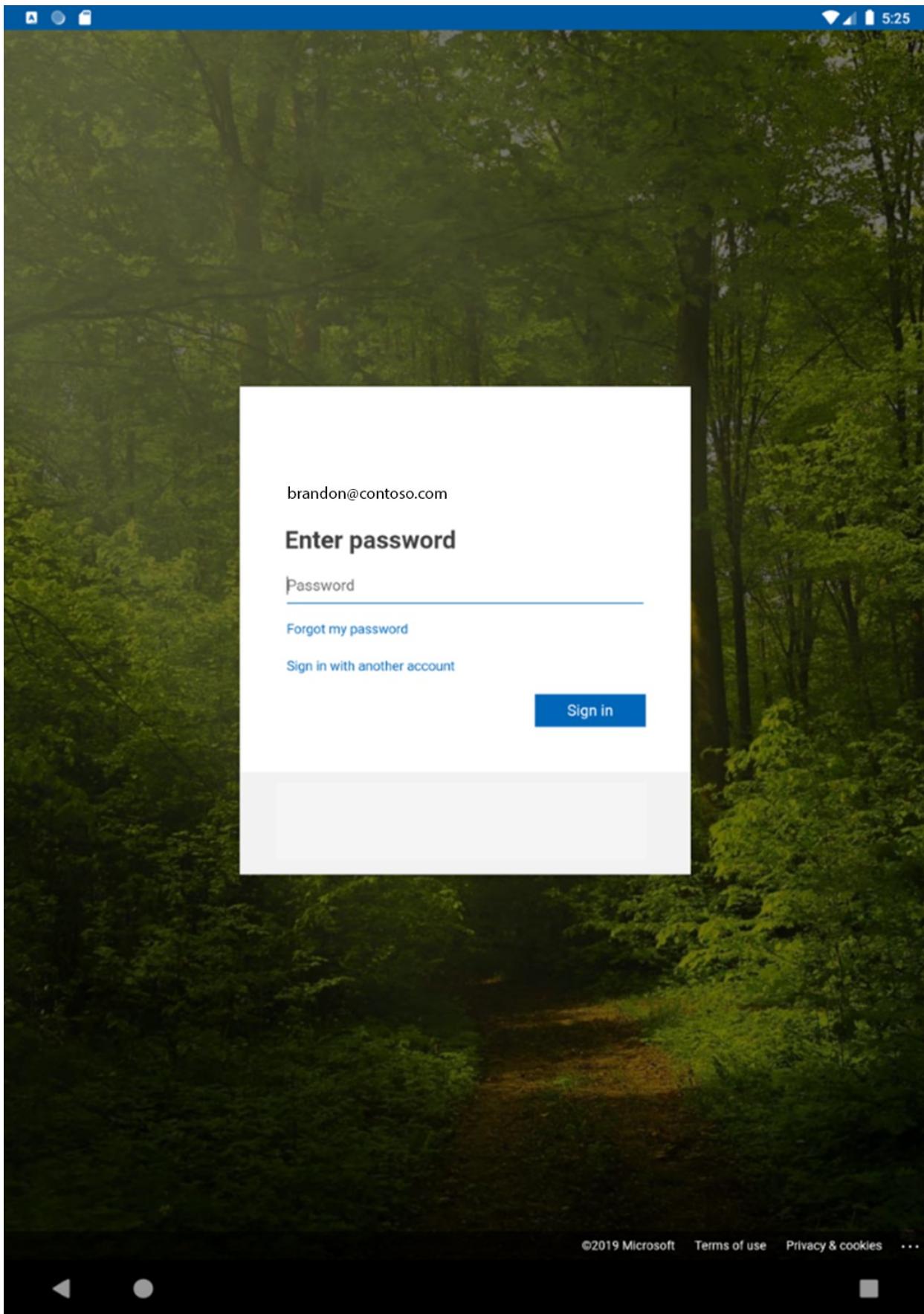


When you click this button, you'll be asked to authorize access to device contacts. This is due to Android's account integration on the device. Choose **allow**.



The Cloud Device Administrator should enter their organizational email under **Or register as a shared device**. Then click the **register as shared device** button, and enter their credentials.





The device is now in shared mode.



Any sign-ins and sign-outs on the device will be global, meaning they apply to all apps that are integrated with MSAL and Microsoft Authenticator on the device. You can now deploy applications to the device that use shared-device mode features.

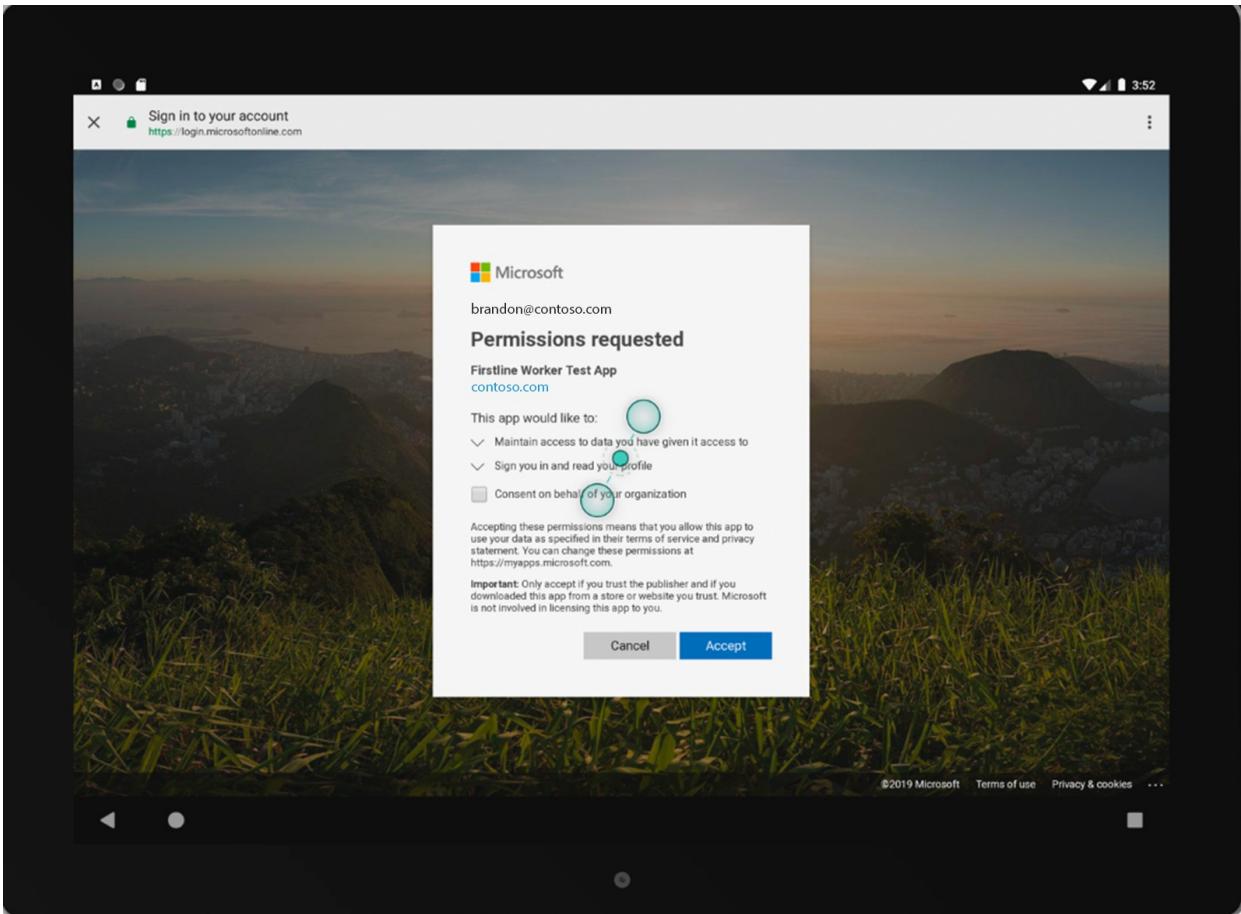
View the shared device in the Azure portal

Once you've put a device in shared-mode, it becomes known to your organization and is tracked in your organizational tenant. You can view your shared devices by looking at the **Join Type** in the Azure Active Directory blade of your Azure portal.

NAME	ENABLED	OS	VERSION	JOIN TYPE	OWNER	MDM	COMPLIANT	REGISTERED	ACTIVITY
GoogleAndroid S...	Yes	Android	8.1.0	Azure AD registered	John Doe	None	N/A	5/26/2019, 5:25:50 PM	5/26/2019, ...

Running the sample app

The Sample Application is a simple app that will call the Graph API of your organization. On first run you'll be prompted to consent as the application is new to your employee account.



Next steps

Learn more about working with the Microsoft Authentication Library and shared device mode on Android devices:

[Shared device mode for Android devices](#)

Tutorial: Sign in users and call Microsoft Graph from an iOS or macOS app

4/12/2022 • 16 minutes to read • [Edit Online](#)

In this tutorial, you build an iOS or macOS app that integrates with the Microsoft identity platform to sign users and get an access token to call 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. This tutorial is applicable to both iOS and macOS apps. Some steps are different between the two platforms.

In this tutorial:

- Create an iOS or macOS app project in *Xcode*
- Register the app in the Azure portal
- Add code to support user sign-in and sign-out
- Add code to call the Microsoft Graph API
- Test the app

Prerequisites

- [Xcode 11.x+](#)

How tutorial app works



The app in this tutorial can sign in users and get data from Microsoft Graph 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).

If you'd like to download a completed version of the app you build in this tutorial, you can find both versions on GitHub:

- [iOS code sample \(GitHub\)](#)

- [macOS code sample](#) (GitHub)

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 select **Create**.

Register your application

1. Sign in to the [Azure portal](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application. Users of your app might see this name, and you can change it later.
6. Select **Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)** under **Supported account types**.
7. Select **Register**.
8. Under **Manage**, select **Authentication > Add a platform > iOS/macOS**.
9. 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.
10. Select **Configure** and save the **MSAL Configuration** that appears in the **MSAL configuration** page so you can enter it when you configure your app later.
11. Select **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`, as well as `AppDelegate.swift` or `SceneDelegate.swift` files:

```
import MSAL
```

Then Add the following code to `ViewController.swift` prior to `viewDidLoad()`:

```
// Update the below to your client ID you received in the portal. The below is for running the demo only
let kClientID = "Your_Application_Id_Here"
let kGraphEndpoint = "https://graph.microsoft.com/" // the Microsoft Graph endpoint
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

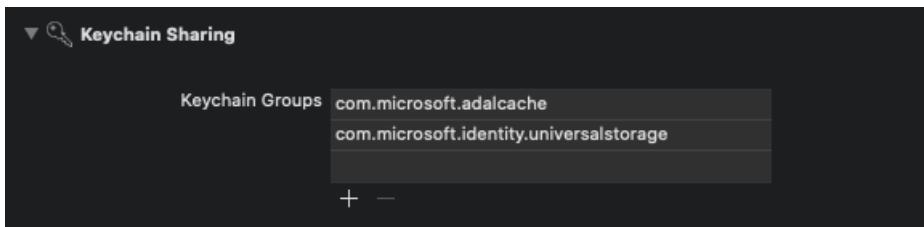
let kScopes: [String] = ["user.read"] // request permission to read the profile of the signed-in user

var accessToken = String()
var applicationContext : MSALPublicClientApplication?
var webViewParameters : MSALWebviewParameters?
var currentAccount: MSALAccount?
```

The only value you 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.

Configure Xcode project settings

Add a new keychain group to your project [Signing & Capabilities](#). The keychain group should be `com.microsoft.adalcache` on iOS and `com.microsoft.identity.universalstorage` on macOS.



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. If you downloaded the code, the bundle identifier 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!
var usernameLabel: UILabel!

func initUI() {

    usernameLabel = UILabel()
    usernameLabel.translatesAutoresizingMaskIntoConstraints = false
    usernameLabel.text = ""
    usernameLabel.textColor = .darkGray
    usernameLabel.textAlignment = .right

    self.view.addSubview(usernameLabel)
```

```

usernameLabel.topAnchor.constraint(equalTo: view.topAnchor, constant: 50.0).isActive = true
usernameLabel.rightAnchor.constraint(equalTo: view.rightAnchor, constant: -10.0).isActive = true
usernameLabel.widthAnchor.constraint(equalToConstant: 300.0).isActive = true
usernameLabel.heightAnchor.constraint(equalToConstant: 50.0).isActive = true

// 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: 120.0).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

let deviceModeButton = UIButton()
deviceModeButton.translatesAutoresizingMaskIntoConstraints = false
deviceModeButton.setTitle("Get device info", for: .normal);
deviceModeButton.setTitleColor(.blue, for: .normal);
deviceModeButton.addTarget(self, action: #selector(getDeviceMode(_:)), for: .touchUpInside)
self.view.addSubview(deviceModeButton)

deviceModeButton.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
deviceModeButton.topAnchor.constraint(equalTo: signOutButton.bottomAnchor, constant: 10.0).isActive =
true
deviceModeButton.widthAnchor.constraint(equalToConstant: 150.0).isActive = true
deviceModeButton.heightAnchor.constraint(equalToConstant: 50.0).isActive = true

// Add logging textfield
loggingText = UITextView()
loggingText.isUserInteractionEnabled = false
loggingText.translatesAutoresizingMaskIntoConstraints = false

self.view.addSubview(loggingText)

loggingText.topAnchor.constraint(equalTo: deviceModeButton.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
}

func platformViewDidLoadSetup() {

NotificationCenter.default.addObserver(self,
    selector: #selector(appCameToForeground(notification:)),
    name: UIApplication.willEnterForegroundNotification,
    object: nil)

}

@objc func appCameToForeground(notification: Notification) {
}

```

```
    self.loadCurrentAccount()  
}
```

macOS UI

```

var callGraphButton: NSButton!
var loggingText: NSTextView!
var signOutButton: NSButton!

var usernameLabel: NSTextField!

func initUI() {

    usernameLabel = NSTextField()
    usernameLabel.translatesAutoresizingMaskIntoConstraints = false
    usernameLabel.stringValue = ""
    usernameLabel.setEditable = false
    usernameLabel.isBezeled = false
    self.view.addSubview(usernameLabel)

    usernameLabel.topAnchor.constraint(equalTo: view.topAnchor, constant: 30.0).isActive = true
    usernameLabel.rightAnchor.constraint(equalTo: view.rightAnchor, constant: -10.0).isActive = true

    // 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: 50.0).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
}

func platformViewDidLoadSetup() {}

```

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)")
    }

    self.loadCurrentAccount()
    self.platformViewDidLoadSetup()
}

```

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(authPresentationViewController: self)
}

```

macOS code:

```

func initWebViewParams() {
    self.webViewParameters = MSALWebviewParameters()
}

```

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. `acquireTokenSilently()` requires providing a valid `MSALAccount` which can be retrieved by using one of MSAL account enumeration APIs. This sample uses `applicationContext.getCurrentAccount(with: msalParameters, completionBlock: {})` to retrieve current account.
- `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:

```

func getGraphEndpoint() -> String {
    return kGraphEndpoint.hasSuffix("/") ? (kGraphEndpoint + "v1.0/me/") : (kGraphEndpoint +
"/v1.0/me/");
}

@objc func callGraphAPI(_ sender: AnyObject) {
    self.loadCurrentAccount { (account) in

        guard let currentAccount = account else {

            // We check to see if we have a current logged in account.
            // If we don't, then we need to sign someone in.
            self.acquireTokenInteractively()
            return
        }

        self.acquireTokenSilently(currentAccount)
    }
}

typealias AccountCompletion = (MSALAccount?) -> Void

func loadCurrentAccount(completion: AccountCompletion? = nil) {

    guard let applicationContext = self.applicationContext else { return }

    let msalParameters = MSALParameters()
    msalParameters.completionBlockQueue = DispatchQueue.main

    applicationContext.getCurrentAccount(with: msalParameters, completionBlock: { (currentAccount,
previousAccount, error) in

        if let error = error {
            self.updateLogging(text: "Couldn't query current account with error: \(error)")
            return
        }

        if let currentAccount = currentAccount {

            self.updateLogging(text: "Found a signed in account \(String(describing:
currentAccount.username)). Updating data for that account...")

            self.updateCurrentAccount(account: currentAccount)

            if let completion = completion {
                completion(self.currentAccount)
            }
        }

        return
    })

    self.updateLogging(text: "Account signed out. Updating UX")
    self.accessToken = ""
    self.updateCurrentAccount(account: nil)

    if let completion = completion {
        completion(nil)
    }
})
}

```

Get a token interactively

The following code snippet 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)
    parameters.promptType = .selectAccount

    // #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.updateCurrentAccount(account: result.account)
        self.getContentWithToken()
    }
}
```

The `promptType` property of `MSALInteractiveTokenParameters` configures the authentication and consent prompt behavior. The following values are supported:

- `.promptIfNecessary` (default) - The user is prompted only if necessary. The SSO experience is determined by the presence of cookies in the webview, and the account type. If multiple users are signed in, account selection experience is presented. *This is the default behavior.*
- `.selectAccount` - If no user is specified, the authentication webview presents a list of currently signed-in accounts for the user to select from.
- `.login` - Requires the user to authenticate in the webview. Only one account may be signed-in at a time if you specify this value.
- `.consent` - Requires the user to consent to the current set of scopes for the request.

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 }

    /**
     Acquire a token for an existing account silently

     - forScopes:           Permissions you want included in the access token received
     in the result in the completionBlock. Not all scopes are
     guaranteed to be included in the access token returned.
     - account:             An account object that we retrieved from the application object before that
the
authentication flow will be locked down to.
     - completionBlock:     The completion block that will be called when the authentication
flow completes, or encounters an error.
    */

    let parameters = MSALSilentTokenParameters(scopes: kScopes, account: account)

    applicationContext.acquireTokenSilent(with: parameters) { (result, error) in

        if let error = error {

            let nsError = error as NSError

            // interactionRequired means we need to ask the user to sign-in. This usually happens
            // when the user's Refresh Token is expired or if the user has changed their password
            // among other possible reasons.

            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 graphURI = getGraphEndpoint()
    let url = URL(string: graphURI)
    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

Signing out with MSAL removes all known information about a user from the application, as well as removing an active session on their device when allowed by device configuration. You can also optionally sign user out from the browser.

To add sign-out capability, add the following code inside the `ViewController` class.

```

@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
        */

        let signoutParameters = MSALSignoutParameters(webviewParameters: self.webViewParameters!)
        signoutParameters.signoutFromBrowser = false // set this to true if you also want to signout
        from browser or webview

        applicationContext.signout(with: account, signoutParameters: signoutParameters, completionBlock:
        {(success, error) in

            if let error = error {
                self.updateLogging(text: "Couldn't sign out account with error: \(error)")
                return
            }

            self.updateLogging(text: "Sign out completed successfully")
            self.accessToken = ""
            self.updateCurrentAccount(account: nil)
        })
    }
}

```

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. Select + and enter one of the following Keychain Groups:
 - iOS: com.microsoft.adalcache
 - macOS: 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
        }
    }
}

func updateAccountLabel() {

    guard let currentAccount = self.currentAccount else {
        self.usernameLabel.text = "Signed out"
        return
    }

    self.usernameLabel.text = currentAccount.username
}

func updateCurrentAccount(account: MSALAccount?) {
    self.currentAccount = account
    self.updateAccountLabel()
    self.updateSignOutButton(enabled: account != nil)
}
```

macOS UI:

```

func updateLogging(text : String) {

    if Thread.isMainThread {
        self.loggingText.string = text
    } else {
        DispatchQueue.main.async {
            self.loggingText.string = text
        }
    }
}

func updateSignOutButton(enabled : Bool) {
    if Thread.isMainThread {
        self.signOutButton.isEnabled = enabled
    } else {
        DispatchQueue.main.async {
            self.signOutButton.isEnabled = enabled
        }
    }
}

func updateAccountLabel() {

    guard let currentAccount = self.currentAccount else {
        self.usernameLabel.stringValue = "Signed out"
        return
    }

    self.usernameLabel.stringValue = currentAccount.username ?? ""
    self.usernameLabel.sizeToFit()
}

func updateCurrentAccount(account: MSALAccount?) {
    self.currentAccount = account
    self.updateAccountLabel()
    self.updateSignOutButton(enabled: account != nil)
}

```

For iOS only, get additional device information

Use following code to read current device configuration, including whether device is configured as shared:

```

@objc func getDeviceMode(_ sender: AnyObject) {

    if #available(iOS 13.0, *) {
        self.applicationContext?.getDeviceInformation(with: nil, completionBlock: { (deviceInformation,
error) in

            guard let deviceInfo = deviceInformation else {
                self.updateLogging(text: "Device info not returned. Error: \(String(describing:
error))")
                return
            }

            let isSharedDevice = deviceInfo.deviceMode == .shared
            let modeString = isSharedDevice ? "shared" : "private"
            self.updateLogging(text: "Received device info. Device is in the \(modeString) mode.")
        })
    } else {
        self.updateLogging(text: "Running on older iOS. GetDeviceInformation API is unavailable.")
    }
}

```

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 by querying all accounts from MSAL. For example, see `accountsFromDeviceForParameters:completionBlock:` [API](#)

Test your app

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.

Next steps

Learn more about building mobile apps that call protected web APIs in our multi-part scenario series.

[Scenario: Mobile application that calls web APIs](#)

Microsoft identity platform code samples

4/12/2022 • 9 minutes to read • [Edit Online](#)

These code samples are built and maintained by Microsoft to demonstrate usage of our authentication libraries with the Microsoft identity platform. Common authentication and authorization scenarios are implemented in several [application types](#), development languages, and frameworks.

- Sign in users to web applications and provide authorized access to protected web APIs.
- Protect a web API by requiring an access token to perform API operations.

Each code sample includes a *README.md* file describing how to build the project (if applicable) and run the sample application. Comments in the code help you understand how these libraries are used in the application to perform authentication and authorization by using the identity platform.

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Angular	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Call .NET Core web API• Call .NET Core web API (B2C)• Call Microsoft Graph via OBO• Call .NET Core web API using PoP• Use App Roles for access control• Use Security Groups for access control• Deploy to Azure Storage and App Service	MSAL Angular	<ul style="list-style-type: none">• Authorization code with PKCE• On-behalf-of (OBO)• Proof of Possession (PoP)
Blazor WebAssembly	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Deploy to Azure App Service	MSAL.js	Implicit Flow

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
JavaScript	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API via OBO and CA • Deploy to Azure Storage and App Service 	MSAL.js	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA)
React	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API using PoP • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service • Deploy to Azure Static Web Apps 	MSAL React	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA) • Proof of Possession (PoP)

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Customize token cache • Call Graph (multi-tenant) • Call Azure REST APIs • Protect web API • Protect web API (B2C) • Protect multi-tenant web API • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service 	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	<ul style="list-style-type: none"> • OpenID connect • Authorization code • On-Behalf-Of

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Blazor	Blazor Server Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call web API • Call web API (B2C) 	MSAL.NET	Authorization code Grant Flow
ASP.NET Core	Advanced Token Cache Scenarios	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	On-Behalf-Of (OBO)
ASP.NET Core	Use the Conditional Access auth context to perform step-up authentication	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	Authorization code
ASP.NET Core	Active Directory FS to Azure AD migration	MSAL.NET	<ul style="list-style-type: none"> • SAML • OpenID connect
ASP.NET	<ul style="list-style-type: none"> • Microsoft Graph Training Sample • Sign in users and call Microsoft Graph • Sign in users and call Microsoft Graph with admin restricted scope • Quickstart: Sign in users 	MSAL.NET	<ul style="list-style-type: none"> • OpenID connect • Authorization code
Java Spring	Azure AD Spring Boot Starter Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Groups for access control • Deploy to Azure App Service 	<ul style="list-style-type: none"> • MSAL Java • Azure AD Boot Starter 	Authorization code
Java Servlets	Spring-less Servlet Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure App Service 	MSAL Java	Authorization code
Java	Sign in users and call Microsoft Graph	MSAL Java	Authorization code
Java Spring	Sign in users and call Microsoft Graph via OBO • Web API	MSAL Java	<ul style="list-style-type: none"> • Authorization code • On-Behalf-Of (OBO)

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js Express	Express web app series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service • Use App Roles for access control • Use Security Groups for access control • Web app that sign in users 	MSAL Node	Authorization code
Python Flask	Flask Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Sign in users and call Microsoft Graph • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Python Django	Django Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Deploy to Azure App Service 	MSAL Python	Authorization code
Ruby	Graph Training <ul style="list-style-type: none"> • Sign in users and call Microsoft Graph 	OmniAuth OAuth2	Authorization code

Web API

The following samples show how to protect a web API with the Microsoft identity platform, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET	Call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
ASP.NET Core	Sign in users and call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
Java	Sign in users	MSAL Java	On-Behalf-Of (OBO)
Node.js	<ul style="list-style-type: none"> • Protect a Node.js web API • Protect a Node.js Web API with Azure AD B2C 	MSAL Node	Authorization bearer

Desktop

The following samples show public client desktop applications that access the Microsoft Graph API, or your own

web API in the name of the user. Apart from the *Desktop (Console) with Web Authentication Manager (WAM)* sample, all these client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> • Call Microsoft Graph • Call Microsoft Graph with token cache • Call Microsoft Graph with custom web UI HTML • Call Microsoft Graph with custom web browser • Sign in users with device code flow 	MSAL.NET	<ul style="list-style-type: none"> • Authorization code with PKCE • Device code
.NET	<ul style="list-style-type: none"> • Call Microsoft Graph with daemon console • Call web API with daemon console 	MSAL.NET	Authorization code with PKCE
.NET	Invoke protected API with integrated Windows authentication	MSAL.NET	Integrated Windows authentication
Java	Call Microsoft Graph	MSAL Java	Integrated Windows authentication
Node.js	Sign in users	MSAL Node	Authorization code with PKCE
PowerShell	Call Microsoft Graph by signing in users using username/password	MSAL.NET	Resource owner password credentials
Python	Sign in users	MSAL Python	Resource owner password credentials
Universal Window Platform (UWP)	Call Microsoft Graph	MSAL.NET	Web account manager
Windows Presentation Foundation (WPF)	Sign in users and call Microsoft Graph	MSAL.NET	Authorization code with PKCE
XAML	<ul style="list-style-type: none"> • Sign in users and call ASP.NET core web API • Sign in users and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Mobile

The following samples show public client mobile applications that access the Microsoft Graph API, or your own web API in the name of the user. These client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
iOS	<ul style="list-style-type: none"> Call Microsoft Graph native Call Microsoft Graph with Azure AD nxauth 	MSAL iOS	Authorization code with PKCE
Java	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Kotlin	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Xamarin	<ul style="list-style-type: none"> Sign in users and call Microsoft Graph Sign in users with broker and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Service / daemon

The following samples show an application that accesses the Microsoft Graph API with its own identity (with no user).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> Call Microsoft Graph Call web API Call own web API Using managed identity and Azure key vault 	MSAL.NET	Client credentials grant
ASP.NET	Multi-tenant with Microsoft identity platform endpoint	MSAL.NET	Client credentials grant
Java	Call Microsoft Graph	MSAL Java	Client credentials grant
Node.js	Sign in users and call web API	MSAL Node	Client credentials grant
Python	<ul style="list-style-type: none"> Call Microsoft Graph with secret Call Microsoft Graph with certificate 	MSAL Python	Client credentials grant

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, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET	.NET Azure function web API secured by Azure AD	MSAL.NET	Authorization code

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Node.js Azure function web API secured by Azure AD	MSAL Node	Authorization bearer
Node.js	Call Microsoft Graph API on behalf of a user	MSAL Node	On-Behalf-Of (OBO)
Python	Python Azure function web API secured by Azure AD	MSAL Python	Authorization code

Headless

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 the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET core	Invoke protected API from text-only device	MSAL.NET	Device code
Java	Sign in users and invoke protected API	MSAL Java	Device code
Python	Call Microsoft Graph	MSAL Python	Device code

Microsoft Teams applications

The following sample illustrates Microsoft Teams Tab application that signs in users. Additionally it demonstrates how to call Microsoft Graph API with the user's identity using the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Teams Tab app: single sign-on (SSO) and call Microsoft Graph	MSAL Node	On-Behalf-Of (OBO)

Multi-tenant SaaS

The following samples show how to configure your application to accept sign-ins from any Azure Active Directory (Azure AD) tenant. Configuring your application to be *multi-tenant* means that you can offer a **Software as a Service** (SaaS) application to many organizations, allowing their users to be able to sign-in to your application after providing consent.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core MVC web application calls Microsoft Graph API	MSAL.NET	OpenID connect
ASP.NET Core	ASP.NET Core MVC web application calls ASP.NET Core Web API	MSAL.NET	Authorization code

Next steps

If you'd like to delve deeper into more sample code, see:

- [Sign in users and call the Microsoft Graph API from an Angular](#)
- [Sign in users in a Nodejs and Express web app](#)
- [Call the Microsoft Graph API from a Universal Windows Platform](#)

Support single sign-on and app protection policies in mobile apps you develop

4/12/2022 • 4 minutes to read • [Edit Online](#)

Single sign-on (SSO) is a key offering of the Microsoft identity platform and Azure Active Directory, providing easy and secure logins for users of your app. In addition, app protection policies (APP) enable support of the key security policies that keep your user's data safe. Together, these features enable secure user logins and management of your app's data.

This article explains why SSO and APP are important and provides the high-level guidance for building mobile applications that support these features. This applies for both phone and tablet apps. If you're an IT administrator that wants to deploy SSO across your organization's Azure Active Directory tenant, check out our [guidance for planning a single sign-on deployment](#)

About single sign-on and app protection policies

[Single sign-on \(SSO\)](#) allows a user to sign in once and get access to other applications without re-entering credentials. This makes accessing apps easier and eliminates the need for users to remember long lists of usernames and passwords. Implementing it in your app makes accessing and using your app easier.

In addition, enabling single sign-on in your app unlocks new authentication mechanisms that come with modern authentication, like [passwordless logins](#). Usernames and passwords are one of the most popular attack vectors against applications, and enabling SSO allows you to mitigate this risk by enforcing conditional access or passwordless logins that add additional security or rely on more secure authentication mechanisms. Finally, enabling single sign-on also enables [single sign-out](#). This is useful in situations like work applications that will be used on shared devices.

[App protection policies \(APP\)](#) ensure that an organization's data remains safe and contained. They allow companies to manage and protect their data within an app and allow control over who can access the app and its data. Implementing app protection policies enables your app to connect users to resources protected by Conditional Access policies and securely transfer data to and from other protected apps. Scenarios unlocked by app protection policies include requiring a PIN to open an app, controlling the sharing of data between apps, and preventing company app data from being saved to personal storage locations.

Implementing single sign-on

We recommend the following to enable your app to take advantage of single sign-on.

Use the Microsoft Authentication Library (MSAL)

The best choice for implementing single sign-on in your application is to use [the Microsoft Authentication Library \(MSAL\)](#). By using MSAL you can add authentication to your app with minimal code and API calls, get the full features of the [Microsoft identity platform](#), and let Microsoft handle the maintenance of a secure authentication solution. By default, MSAL adds SSO support for your application. In addition, using MSAL is a requirement if you also plan to implement app protection policies.

NOTE

It is possible to configure MSAL to use an embedded web view. This will prevent single sign-on. Use the default behavior (that is, the system web browser) to ensure that SSO will work.

If you're currently using the [ADAL library](#) in your application, then we highly recommend that you [migrate it to MSAL](#), as [ADAL is being deprecated](#).

For iOS applications, we have a [quickstart](#) that shows you how to set up sign-ins using MSAL, as well as [guidance for configuring MSAL for various SSO scenarios](#).

For Android applications, we have a [quickstart](#) that shows you how to set up sign-ins using MSAL, and guidance for [how to enable cross-app SSO on Android using MSAL](#).

Use the system web browser

A web browser is required for interactive authentication. For mobile apps that use modern authentication libraries other than MSAL (that is, other OpenID Connect or SAML libraries), or if you implement your own authentication code, you should use the system browser as your authentication surface to enable SSO.

Google has guidance for doing this in Android Applications: [Chrome Custom Tabs - Google Chrome](#).

Apple has guidance for doing this in iOS applications: [Authenticating a User Through a Web Service | Apple Developer Documentation](#).

TIP

The [SSO plug-in for Apple devices](#) allows SSO for iOS apps that use embedded web views on managed devices using Intune. We recommend MSAL and system browser as the best option for developing apps that enable SSO for all users, but this will allow SSO in some scenarios where it otherwise is not possible.

Enable App Protection Policies

To enable app protection policies, use the [Microsoft Authentication Library \(MSAL\)](#). MSAL is the Microsoft identity platform's authentication and authorization library and the Intune SDK is developed to work in tandem with it.

In addition, you must use a broker app for authentication. The broker requires the app to provide application and device information to ensure app compliance. iOS users will use the [Microsoft Authenticator app](#) and Android users will use either the Microsoft Authenticator app or the [Company Portal app for brokered authentication](#). By default, MSAL uses a broker as its first choice for fulfilling an authentication request, so using the broker to authenticate will be enabled for your app automatically when using MSAL out-of-the-box.

Finally, [add the Intune SDK to your app](#) to enable app protection policies. The SDK for the most part follows an intercept model and will automatically apply app protection policies to determine if actions the app is taking are allowed or not. There are also APIs you can call manually to tell the app if there are restrictions on certain actions.

Additional resources

- [Plan an Azure Active Directory single sign-on deployment](#)
- [How to: Configure SSO on macOS and iOS](#)
- [Microsoft Enterprise SSO plug-in for Apple devices \(Preview\)](#)
- [Brokered authentication in Android](#)
- [Authorization agents and how to enable them](#)

- [Get started with the Microsoft Intune App SDK](#)
- [Configure settings for the Intune App SDK](#)
- [Microsoft Intune protected apps](#)

Overview of shared device mode

4/12/2022 • 4 minutes to read • [Edit Online](#)

Shared device mode is a feature of Azure Active Directory that allows you to build applications that support frontline workers and enable shared device mode on the devices deployed to them.

IMPORTANT

Shared device mode for iOS is in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Some features might be unsupported or have constrained capabilities. For more information, see [Supplemental terms of use for Microsoft Azure previews](#).

What are frontline workers?

Frontline workers are retail employees, maintenance and field agents, medical personnel, and other users that don't sit in front of a computer or use corporate email for collaboration. The following sections introduce the aspects and challenges of supporting frontline workers, followed by an introduction to the features provided by Microsoft that enable your application for use by an organization's frontline workers.

Challenges of supporting frontline workers

Enabling frontline worker workflows includes challenges not usually presented by typical information workers. Such challenges can include high turnover rate and less familiarity with an organization's core productivity tools. To empower their frontline workers, organizations are adopting different strategies. Some are adopting a bring-your-own-device (BYOD) strategy in which their employees use business apps on their personal phone, while others provide their employees with shared devices like iPads or Android tablets.

Supporting multiple users on devices designed for one user

Because mobile devices running iOS or Android were designed for single users, most applications optimize their experience for use by a single user. Part of this optimized experience means enabling single sign-on across applications and keeping users signed in on their device. When a user removes their account from an application, the app typically doesn't consider it a security-related event. Many apps even keep a user's credentials around for quick sign-in. You may even have experienced this yourself when you've deleted an application from your mobile device and then reinstalled it, only to discover you're still signed in.

Global sign-in and sign-out (SSO)

To allow an organization's employees to use its apps across a pool of devices shared by those employees, developers need to enable the opposite experience. Employees should be able to pick a device from the pool and perform a single gesture to "make it theirs" for the duration of their shift. At the end of their shift, they should be able to perform another gesture to sign out globally on the device, with all their personal and company information removed so they can return it to the device pool. Furthermore, if an employee forgets to sign out, the device should be automatically signed out at the end of their shift and/or after a period of inactivity.

Azure Active Directory enables these scenarios with a feature called **shared device mode**.

Introducing shared device mode

As mentioned, shared device mode is a feature of Azure Active Directory that enables you to:

- Build applications that support frontline workers
- Deploy devices to frontline workers and turn on shared device mode

Build applications that support frontline workers

You can support frontline workers in your applications by using the Microsoft Authentication Library (MSAL) and [Microsoft Authenticator app](#) to enable a device state called *shared device mode*. When a device is in shared device mode, Microsoft provides your application with information to allow it to modify its behavior based on the state of the user on the device, protecting user data.

Supported features are:

- **Sign in a user device-wide** through any supported application.
- **Sign out a user device-wide** through any supported application.
- **Query the state of the device** to determine if your application is on a device that's in shared device mode.
- **Query the device state of the user** on the device to determine if anything has changed since the last time your application was used.

Supporting shared device mode should be considered a feature upgrade for your application, and can help increase its adoption in environments where the same device is used among multiple users.

Your users depend on you to ensure their data isn't leaked to another user. Share Device Mode provides helpful signals to indicate to your application that a change you should manage has occurred. Your application is responsible for checking the state of the user on the device every time the app is used, clearing the previous user's data. This includes if it is reloaded from the background in multi-tasking. On a user change, you should ensure both the previous user's data is cleared and that any cached data being displayed in your application is removed. We recommend you always perform a thorough security review process after adding shared device mode capability to your app.

For details on how to modify your applications to support shared device mode, see the [Next steps](#) section at the end of this article.

Deploy devices to frontline workers and turn on shared device mode

Once your applications support shared device mode and include the required data and security changes, you can advertise them as being usable by frontline workers.

An organization's device administrators are able to deploy their devices and your applications to their stores and workplaces through a mobile device management (MDM) solution like Microsoft Intune. Part of the provisioning process is marking the device as a *Shared Device*. Administrators configure shared device mode by deploying the [Microsoft Authenticator app](#) and setting shared device mode through configuration parameters. After performing these steps, all applications that support shared device mode will use the Microsoft Authenticator application to manage its user state and provide security features for the device and organization.

Next steps

We support iOS and Android platforms for shared device mode. Review the documentation below for your platform to begin supporting frontline workers in your applications.

- [Supporting shared device mode for iOS](#)
- [Supporting shared device mode for Android](#)

Scenario: Mobile application that calls web APIs

4/12/2022 • 2 minutes to read • [Edit Online](#)

Learn how to build a mobile app that calls web APIs.

Getting started

If you haven't already, create your first application by completing 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 and Android app \(GitHub\)](#)

Overview

A personalized, seamless user experience is essential for mobile apps. The 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. It can also acquire tokens to call a web API on their behalf. To implement these flows, we'll use the Microsoft Authentication Library (MSAL). MSAL 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 you ask for sign-in. Request only the required permissions.
- **Support all user configurations:** Many mobile business users must adhere to conditional-access policies and device-compliance policies. Be sure to support these key scenarios.
- **Implement single sign-on (SSO):** By using MSAL and Microsoft identity platform, you can enable single sign-on through the device's browser or Microsoft Authenticator (and Intune Company Portal on Android).
- **Implement shared device mode:** Enable your application to be used in shared-device scenarios, for example hospitals, manufacturing, retail, and finance. [Read more about supporting shared device mode](#).

Specifics

Keep in mind the following considerations when you build a mobile app on Microsoft identity platform:

- Depending on the platform, some user interaction might be required the first time that users sign in. For example, iOS requires apps to show user interaction when they use SSO for the first time through Microsoft Authenticator (and Intune Company Portal on Android).
- On iOS and Android, MSAL might use an external browser to sign in users. The external browser might appear on top of your app.
- Never use a secret in a mobile application. In these applications, secrets are accessible to all users.

Recommended reading

If you're new to identity and access management (IAM) with OAuth 2.0 and OpenID Connect, or even just new to IAM on the Microsoft identity platform, the following set of articles should be high on your reading list.

Although not required reading before completing your first quickstart or tutorial, they cover topics integral to the platform, and familiarity with them will help you on your path as you build more complex scenarios.

Authentication and authorization

- [Authentication basics](#)
- [ID tokens](#)
- [Access tokens](#)

Microsoft identity platform

- [Audiences](#)
- [Applications and service principals](#)
- [Permissions and consent](#)

Next steps

Move on to the next article in this scenario, [App registration](#).

Register mobile apps that call web APIs

4/12/2022 • 2 minutes to read • [Edit Online](#)

This article contains instructions to help you register a mobile application that you're creating.

Supported account types

The account types that your mobile applications support depend on the experience that you want to enable and the flows that you want to use.

Audience for interactive token acquisition

Most mobile applications use interactive authentication. If your app uses this form of authentication, you can sign in users from any [account type](#).

Audience for integrated Windows authentication, username-password, and B2C

If you have a Universal Windows Platform (UWP) app, you can use integrated Windows authentication (IWA) to sign in users. To use IWA or username-password authentication, your application needs to sign in users in your own line-of-business (LOB) developer tenant. In an independent software vendor (ISV) scenario, your application can sign in users in Azure Active Directory organizations. These authentication flows aren't supported for Microsoft personal accounts.

You can also sign in users by using social identities that pass a B2C authority and policy. To use this method, you can use only interactive authentication and username-password authentication. Username-password authentication is currently supported only on Xamarin.iOS, Xamarin.Android, and UWP.

For more information, see [Scenarios and supported authentication flows](#) and [Scenarios and supported platforms and languages](#).

Platform configuration and redirect URIs

Interactive authentication

When you build a mobile app that uses interactive authentication, the most critical registration step is the redirect URI. You can set interactive authentication through the [platform configuration on the Authentication blade](#).

This experience will enable your app to get single sign-on (SSO) through Microsoft Authenticator (and Intune Company Portal on Android). It will also support device management policies.

The app registration portal provides a preview experience to help you compute the brokered reply URI for iOS and Android applications:

1. In the app registration portal, select **Authentication > Try out the new experience**.

Microsoft Azure

my ios app - Authentication

TRY OUT THE NEW EXPERIENCE

\Redirect URIs

TYPE REDIRECT URI

Web e.g. https://myapp.com/auth

Suggested Redirect URIs for public clients (mobile, desktop)

Implicit grant

Advanced settings

Logout URL

Access tokens

2. Select Add a platform.

Microsoft Azure

my ios app - Platform configurations

+ ADD A PLATFORM

Platform configurations

Supported account types

Who can use this application or access this API?

Accounts in this organizational directory only (msidentity-samples-testing)

Accounts in any organizational directory

Help me decide...

⚠ 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.

Advanced settings

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

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. The main area is titled 'my ios app - Platform configurations'. It includes sections for 'Platform configurations', 'Supported account types', and 'Advanced settings'. A large blue box highlights the 'Mobile applications' section, which contains options for 'iOS' (selected) and 'Android'. Below this, another blue box highlights the 'iOS' entry under 'Objective-C, Swift, Xamarin'.

4. Enter your bundle ID, and then select Register.

This screenshot shows the 'Configure your iOS app' step. The 'Bundle ID' field is populated with 'com.yourcompany.xforms'. At the bottom right, there are 'Configure' and 'Cancel' buttons.

When you complete the steps, the redirect URI is computed for you, as in the following image.

If you prefer to manually configure the redirect URI, you can do so through the application manifest. Here's the recommended format for the manifest:

- **iOS:** `msauth.<Bundle_ID>://auth`
 - For example, enter `msauth.com.yourcompany.appName://auth`
- **Android:** `msauth://<PACKAGE_NAME>/<SIGNATURE_HASH>`
 - You can generate the Android signature hash by using the release key or debug key through the KeyTool command.

Username-password authentication

If your app uses only username-password authentication, you don't need to register a redirect URI for your application. This flow does a round trip to the Microsoft identity platform. Your application won't be called back on any specific URI.

However, identify your application as a public client application. To do so:

1. Still in the [Azure portal](#), select your app in **App registrations**, and then select **Authentication**.
2. In **Advanced settings > Allow public client flows > Enable the following mobile and desktop flows:**, select **Yes**.

Advanced settings

Allow public client flows ①

Enable the following mobile and desktop flows:

Yes

No

- App collects plaintext password (Resource Owner Password Credential Flow) [Learn more](#)
- No keyboard (Device Code Flow) [Learn more](#)
- SSO for domain-joined Windows (Windows Integrated Auth Flow) [Learn more](#)

API permissions

Mobile applications call APIs for the signed-in user. Your app needs to request delegated permissions. These

permissions are also called scopes. Depending on the experience that you want, you can request delegated permissions statically through the Azure portal. Or you can request them dynamically at runtime.

By statically registering permissions, you allow administrators to easily approve your app. Static registration is recommended.

Next steps

Move on to the next article in this scenario, [App code configuration](#).

Configure a mobile app that calls web APIs

4/12/2022 • 8 minutes to read • [Edit Online](#)

After you create your application, you'll learn how to configure the code by using the app registration parameters. Mobile applications present some complexities related to fitting into their creation framework.

Microsoft libraries supporting mobile apps

The following Microsoft libraries support mobile apps:

PLATFORM	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
Android (Java)	MSAL Android	MSAL	Quickstart	✓	✓	GA
Android (Kotlin)	MSAL Android	MSAL	—	✓	✓	GA
iOS (Swift/Obj-C)	MSAL for iOS and macOS	MSAL	Tutorial	✓	✓	GA
Xamarin (.NET)	MSAL.NET	Microsoft.Identity.Client	—	✓	✓	GA

¹ [Supplemental terms of use for Microsoft Azure Previews](#) apply to libraries in *Public preview*.

Instantiate the application

Android

Mobile applications use the `PublicClientApplication` class. Here's 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. To instantiate the class, use the following code.

```
NSError *msalError = nil;  
  
MSALPublicClientApplicationConfig *config = [[MSALPublicClientApplicationConfig alloc] initWithClientId:@"  
<your-client-id-here>"];  
MSALPublicClientApplication *application = [[MSALPublicClientApplication alloc] initWithConfiguration:config  
error:&msalError];
```

```
let config = MSALPublicClientApplicationConfig(clientId: "<your-client-id-here>")
if let application = try? MSALPublicClientApplication(configuration: config){ /* Use application */}
```

Additional [MSALPublicClientApplicationConfig](#) properties can override the default authority, specify a redirect URI, or change the behavior of MSAL token caching.

Xamarin or UWP

This section explains how to instantiate the application for Xamarin.iOS, Xamarin.Android, and UWP apps.

Instantiate the application

In Xamarin or UWP, the simplest way to instantiate the application is by using the following code. In this code, `ClientId` is the GUID of your registered app.

```
var app = PublicClientApplicationBuilder.Create(clientId)
    .Build();
```

Additional `With<Parameter>` methods set the UI parent, override the default authority, specify a client name and version for telemetry, specify a redirect URI, and specify the HTTP factory to use. The HTTP factory might be used, for instance, to handle proxies and to specify telemetry and logging.

The following sections provide more information about instantiating the application.

Specify the parent UI, window, or activity

On Android, pass the parent activity before you do the interactive authentication. On iOS, when you use a broker, pass-in `ViewController`. In the same way on UWP, you might want to pass-in the parent window. You pass it in when you acquire the token. But when you're creating the app, you can also specify a callback as a delegate that returns `UIParent`.

```
IPublicClientApplication application = PublicClientApplicationBuilder.Create(clientId)
    .ParentActivityOrWindowFunc(() => parentUi)
    .Build();
```

On Android, we recommend that you use [CurrentActivityPlugin](#). The resulting `PublicClientApplication` builder code looks like this example:

```
// Requires MSAL.NET 4.2 or above
var pca = PublicClientApplicationBuilder
    .Create("<your-client-id-here>")
    .WithParentActivityOrWindow(() => CrossCurrentActivity.Current)
    .Build();
```

Find more app-building parameters

For a list of all methods that are available on `PublicClientApplicationBuilder`, see the [Methods list](#).

For a description of all options that are exposed in `PublicClientApplicationOptions`, see the [reference documentation](#).

Tasks for Xamarin iOS

If you use MSAL.NET on Xamarin iOS, do the following tasks.

- [Override and implement the `openUrl` function in `AppDelegate`](#)
- [Enable keychain groups](#)
- [Enable token cache sharing](#)
- [Enable keychain access](#)

For more information, see [Xamarin iOS considerations](#).

Tasks for MSAL for iOS and macOS

These tasks are necessary when you use MSAL for iOS and macOS:

- [Implement the `openURL` callback](#)
- [Enable keychain access groups](#)
- [Customize browsers and WebViews](#)

Tasks for Xamarin.Android

If you use Xamarin.Android, do the following tasks:

- [Ensure control goes back to MSAL after the interactive portion of the authentication flow ends](#)
- [Update the Android manifest](#)
- [Use the embedded web view \(optional\)](#)
- [Troubleshoot as necessary](#)

For more information, see [Xamarin.Android considerations](#).

For considerations about the browsers on Android, see [Xamarin.Android-specific considerations with MSAL.NET](#).

Tasks for UWP

On UWP, you can use corporate networks. The following sections explain the tasks that you should complete in the corporate scenario.

For more information, see [UWP-specific considerations with MSAL.NET](#).

Configure the application to use the broker

On Android and iOS, brokers enable:

- **Single sign-on (SSO):** You can use SSO for devices that are registered with Azure Active Directory (Azure AD). When you use SSO, your users don't need to sign in to each application.
- **Device identification:** This setting enables conditional-access policies that are related to Azure AD devices. The authentication process uses the device certificate that was created when the device was joined to the workplace.
- **Application identification verification:** When an application calls the broker, it passes its redirect URL. Then the broker verifies it.

Enable the broker on Xamarin

To enable the broker on Xamarin, use the `.WithBroker()` parameter when you call the `PublicClientApplicationBuilder.CreateApplication` method. By default, `.WithBroker()` is set to true.

To enable brokered authentication for Xamarin.iOS, follow the steps in the [Xamarin.iOS section](#) in this article.

Enable the broker for MSAL for Android

For information about enabling a broker on Android, see [Brokered authentication on Android](#).

Enable the broker for MSAL for iOS and macOS

Brokered authentication is enabled by default for Azure AD scenarios in MSAL for iOS and macOS.

The following sections provide instructions to configure your application for brokered authentication support for either MSAL for Xamarin.iOS or MSAL for iOS and macOS. In the two sets of instructions, some of the steps differ.

Enable brokered authentication for Xamarin iOS

Follow the steps in this section to enable your Xamarin.iOS app to talk with the [Microsoft Authenticator](#) app.

Step 1: Enable broker support

Broker support is disabled by default. You enable it for an individual `PublicClientApplication` class. Use the `WithBroker()` parameter when you create the `PublicClientApplication` class through `PublicClientApplicationBuilder`. The `WithBroker()` parameter is set to true by default.

```
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 then calls back to your application. It calls back by using the `AppDelegate.OpenUrl` method. Because MSAL waits for the response from the broker, your application needs to cooperate to call MSAL.NET back. You set up this behavior by updating the `AppDelegate.cs` file to override the method, as the following code shows.

```
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 an opportunity to process the response from the broker and to complete the authentication process that MSAL.NET started.

Step 3: Set a UIViewController()

For Xamarin iOS, you don't normally need to set an object window. But in this case you should set it so that you can send and receive responses from a broker. To set an object window, in `AppDelegate.cs`, you set a `ViewController`.

To set the object window, follow these steps:

1. In `AppDelegate.cs`, set the `App.RootViewController` to a new `UIViewController()`. This setting ensures that the call to the broker includes `UIViewController`. 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)`. Pass in the reference to the object window that you'll use. Here's an example:

In `App.cs`:

```
public static object RootViewController { get; set; }
```

In `AppDelegate.cs`:

```
LoadApplication(new App());
App.RootViewController = new UIViewController();
```

In the `AcquireToken` 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 your app's URL scheme in the `Info.plist` file.

To register your app's URL scheme, follow these steps:

1. Prefix `CFBundleURLSchemes` with `msauth`.
2. Add `CFBundleURLName` to the end. Follow this pattern:

```
$"msauth.(BundleId)"
```

Here, `BundleId` uniquely identifies your device. For example, if `BundleId` is `yourcompany.xforms`, your URL scheme is `msauth.com.yourcompany.xforms`.

This URL scheme will become part of the redirect URI that uniquely identifies your app when it receives the broker's response.

```
<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 to the `LSApplicationQueriesSchemes` section

MSAL uses `-canOpenURL:` to check if the broker is installed on the device. In iOS 9, Apple locked down the schemes that an application can query for.

Add `msauthv2` to the `LSApplicationQueriesSchemes` section of the `Info.plist` file, as in the following code example:

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

Brokered authentication for MSAL for iOS and macOS

Brokered authentication is enabled by default for Azure AD scenarios.

Step 1: Update AppDelegate to handle the callback

When MSAL for iOS and macOS calls the broker, the broker calls back to your application by using the `openURL` method. Because MSAL waits for the response from the broker, your application needs to cooperate to call back MSAL. Set up this capability by updating the `AppDelegate.m` file to override the method, as the following code examples show.

```
- (BOOL)application:(UIApplication *)app
    openURL:(NSURL *)url
    options:(NSDictionary<UIApplicationOpenURLOptionsKey,id> *)options
{
    return [MSALPublicClientApplication handleMSALResponse:url
    sourceApplication:options[UIApplicationOpenURLOptionsSourceApplicationKey]];
}
```

```
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)
}
```

If you adopted `UISceneDelegate` on iOS 13 or later, then place the MSAL callback into the `scene:openURLContexts:` of `UISceneDelegate` instead. MSAL `handleMSALResponse:sourceApplication:` must be called only once for each URL.

For more information, see the [Apple documentation](#).

Step 2: Register a URL scheme

MSAL for iOS and macOS uses URLs to invoke the broker and then return the broker response to your app. To finish the round trip, register a URL scheme for your app in the `Info.plist` file.

To register a scheme for your app:

1. Prefix your custom URL scheme with `msauth`.
2. Add your bundle identifier to the end of your scheme. Follow this pattern:

```
$"msauth.(BundleId)"
```

Here, `BundleId` uniquely identifies your device. For example, if `BundleId` is `yourcompany.xforms`, your URL scheme is `msauth.com.yourcompany.xforms`.

This URL scheme will become part of the redirect URI that uniquely identifies your app when it receives the broker's response. Make sure that the redirect URI in the format `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: Add LSApplicationQueriesSchemes

Add `LSApplicationQueriesSchemes` to allow calls to the Microsoft Authenticator app, if it's installed.

NOTE

The `msauthv3` scheme is needed when your app is compiled by using Xcode 11 and later.

Here's an example of how to add `LSApplicationQueriesSchemes`:

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

Brokered authentication for Xamarin.Android

For information about enabling a broker on Android, see [Brokered authentication on Xamarin.Android](#).

Next steps

Move on to the next article in this scenario, [Acquiring a token](#).

Xamarin Android system browser considerations for using MSAL.NET

4/12/2022 • 2 minutes to read • [Edit Online](#)

This article discusses what you should consider when you use 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. It no longer requires Chrome be installed on the Android device for authentication.

We recommend that you use browsers that support custom tabs. Here are some examples of these browsers:

BROWSERS THAT HAVE 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 identifying browsers that offer custom tabs support, our testing indicates that a few browsers that don't support custom tabs also work for authentication. These browsers include Opera, Opera Mini, InBrowser, and Maxthon.

Tested devices and browsers

The following table lists the devices and browsers that have been tested for authentication compatibility.

DEVICE	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

DEVICE	BROWSER	RESULT
Huawei	OperaMini	Pass
Huawei/One+	InBrowser	Pass
One+	Maxthon	Pass
Huawei/One+	DuckDuckGo	User canceled authentication
Huawei/One+	UC Browser	User canceled authentication
One+	Dolphin	User canceled authentication
One+	CM Browser	User canceled authentication
Huawei/One+	None installed	AndroidActivityNotFound exception

* Supports custom tabs

Known issues

If the user has no browser enabled on the device, MSAL.NET will throw an `AndroidActivityNotFound` exception.

- **Mitigation:** Ask the user to enable a browser on their device. Recommend a browser that supports custom tabs.

If authentication fails (for example, if authentication launches with DuckDuckGo), MSAL.NET will return

`AuthenticationCanceled MsalClientException`.

- **Root problem:** A browser that supports custom tabs wasn't enabled on the device. Authentication launched with a browser that couldn't complete authentication.
- **Mitigation:** Ask the user to enable a browser on their device. Recommend a browser that supports custom tabs.

Next steps

For more information and code examples, see [Choosing between an embedded web browser and a system browser on Xamarin Android](#) and [Embedded versus system web UI](#).

Configuration requirements and troubleshooting tips for Xamarin Android with MSAL.NET

4/12/2022 • 6 minutes to read • [Edit Online](#)

There are several configuration changes you're required to make in your code when using Xamarin Android with the Microsoft Authentication Library for .NET (MSAL.NET). The following sections describe the required modifications, followed by a [Troubleshooting](#) section to help you avoid some of the most common issues.

Set the parent activity

On Xamarin Android, set the parent activity so the token returns after the interaction:

```
var authResult = AcquireTokenInteractive(scopes)
    .WithParentActivityOrWindow(parentActivity)
    .ExecuteAsync();
```

In MSAL.NET 4.2 and later, you can also set this functionality at the level of [PublicClientApplication](#). To do so, use a callback:

```
// Requires MSAL.NET 4.2 or later
var pca = PublicClientApplicationBuilder
    .Create("<your-client-id-here>")
    .WithParentActivityOrWindow(() => parentActivity)
    .Build();
```

If you use [CurrentActivityPlugin](#), then your [PublicClientApplication](#) builder code should look similar to this code snippet:

```
// Requires MSAL.NET 4.2 or later
var pca = PublicClientApplicationBuilder
    .Create("<your-client-id-here>")
    .WithParentActivityOrWindow(() => CrossCurrentActivity.Current)
    .Build();
```

Ensure that control returns to MSAL

When the interactive portion of the authentication flow ends, return control to MSAL by overriding the [Activity.OnActivityResult\(\)](#) method.

In your override, call MSAL.NET's [AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs\(\)](#) method to return control to MSAL at the end of the interactive portion of the authentication flow.

```

protected override void OnActivityResult(int requestCode,
                                       Result resultCode,
                                       Intent data)
{
    base.OnActivityResult(requestCode, resultCode, data);

    // Return control to MSAL
    AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(requestCode,
                                                                           resultCode,
                                                                           data);
}

```

Update the Android manifest for System WebView support

To support System WebView, the *AndroidManifest.xml* file should contain the following values:

```

<activity android:name="microsoft.identity.client.BrowserTabActivity"
          android:configChanges="orientation|screenSize">
    <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>

```

The `android:scheme` value is created from the redirect URI that's configured in the application portal. For example, if your redirect URI is `msal4a1aa1d5-c567-49d0-ad0b-cd957a47f842://auth`, the `android:scheme` entry in the manifest would look like this example:

```
<data android:scheme="msal4a1aa1d5-c567-49d0-ad0b-cd957a47f842" android:host="auth" />
```

Alternatively, [create the activity in code](#) rather than manually editing *AndroidManifest.xml*. To create the activity in code, first create a class that includes the `Activity` attribute and the `IntentFilter` attribute.

Here's an example of a class that represents the values of the XML file:

```

[Activity]
[IntentFilter(new[] { Intent.ActionView },
             Categories = new[] { Intent.CategoryBrowsable, Intent.CategoryDefault },
             DataHost = "auth",
             DataScheme = "msal{client_id}")]
public class MsalActivity : BrowserTabActivity
{
}

```

Use System WebView in brokered authentication

To use System WebView as a fallback for interactive authentication when you've configured your application to use brokered authentication and the device doesn't have a broker installed, enable MSAL to capture the authentication response by using the broker's redirect URI. MSAL will try to authenticate by using the default System WebView on the device when it detects that the broker is unavailable. Using this default, it will fail because the redirect URI is configured to use a broker, and System WebView doesn't know how to use it to return to MSAL. To resolve this, create an *intent filter* by using the broker redirect URI that you configured earlier. Add the intent filter by modifying your application's manifest like this example:

```

<!--Intent filter to capture System WebView or Authenticator calling back to our app after sign-in-->
<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="msauth"
            android:host="Enter_the_Package_Name"
            android:path="/Enter_the_Signature_Hash" />
    </intent-filter>
</activity>

```

Substitute the package name that you registered in the Azure portal for the `android:host=` value. Substitute the key hash that you registered in the Azure portal for the `android:path=` value. The signature hash should **not** be URL-encoded. Ensure that a leading forward slash (`/`) appears at the beginning of your signature hash.

Xamarin.Forms 4.3.x manifest

Xamarin.Forms 4.3.x generates code that sets the `package` attribute to `com.companyname.{appName}` in `AndroidManifest.xml`. If you use `Datascheme` as `msal{client_id}`, then you might want to change the value to match the value of the `MainActivity.cs` namespace.

Android 11 support

To use the system browser and brokered authentication in Android 11, you must first declare these packages, so they are visible to the app. Apps that target Android 10 (API 29) and earlier can query the OS for a list of packages that are available on the device at any given time. To support privacy and security, Android 11 reduces package visibility to a default list of OS packages and the packages that are specified in the app's `AndroidManifest.xml` file.

To enable the application to authenticate by using both the system browser and the broker, add the following section to `AndroidManifest.xml`:

```

<!-- Required for API Level 30 to make sure the app can detect browsers and other apps where communication
is needed.-->
<!--https://developer.android.com/training/basics/intents/package-visibility-use-cases-->
<queries>
    <package android:name="com.azure.authenticator" />
    <package android:name="{Package Name}" />
    <package android:name="com.microsoft.windowsintune.companyportal" />
    <!-- Required for API Level 30 to make sure the app detect browsers
        (that don't support custom tabs) -->
<intent>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="https" />
</intent>
    <!-- Required for API Level 30 to make sure the app can detect browsers that support custom tabs -->
    <!-- https://developers.google.com/web/updates/2020/07/custom-tabs-android-
11#detecting_browsers_that_support_custom_tabs -->
    <intent>
        <action android:name="android.support.customtabs.action.CustomTabsService" />
    </intent>
</queries>

```

Replace `{Package Name}` with the application package name.

Your updated manifest, which now includes support for the system browser and brokered authentication, should look similar to this example:

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1"
    android:versionName="1.0" package="com.companyname.XamarinDev">
    <uses-sdk android:minSdkVersion="21" android:targetSdkVersion="30" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <application android:theme="@android:style/Theme.NoTitleBar">
        <activity android:name="microsoft.identity.client.BrowserTabActivity"
            android:configChanges="orientation|screenSize">
            <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="msal4a1aa1d5-c567-49d0-ad0b-cd957a47f842" android:host="auth" />
            </intent-filter>
            <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="com.companyname.XamarinDev" />
            </intent-filter>
        </activity>
    </application>
    <!-- Required for API Level 30 to make sure we can detect browsers and other apps we want to
        be able to talk to.-->
    <!--https://developer.android.com/training/basics/intents/package-visibility-use-cases-->
    <queries>
        <package android:name="com.azure.authenticator" />
        <package android:name="com.companyname.xamarindev" />
        <package android:name="com.microsoft.windowsintune.companyportal" />
    <!-- Required for API Level 30 to make sure we can detect browsers
        (that don't support custom tabs) -->
    <intent>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="https" />
    </intent>
    <!-- Required for API Level 30 to make sure we can detect browsers that support custom tabs -->
    <!-- https://developers.google.com/web/updates/2020/07/custom-tabs-android-11#detecting_browsers_that_support_custom_tabs -->
    <intent>
        <action android:name="android.support.customtabs.action.CustomTabsService" />
    </intent>
    </queries>
</manifest>

```

Use the embedded web view (optional)

By default, MSAL.NET uses the system web browser. This browser enables you to get single sign-on (SSO) by using web applications and other apps. In some rare cases, you might want your system to use an embedded web view.

This code example shows how to set up an embedded web view:

```

bool useEmbeddedWebView = !app.IsSystemWebViewAvailable;

var authResult = AcquireTokenInteractive(scopes)
    .WithParentActivityOrWindow(parentActivity)
    .WithEmbeddedWebView(useEmbeddedWebView)
    .ExecuteAsync();

```

For more information, see [Use web browsers for MSAL.NET](#) and [Xamarin Android system browser](#)

considerations.

Troubleshooting

General tips

- Update the existing MSAL.NET NuGet package to the [latest version of MSAL.NET](#).
- Verify that Xamarin.Forms is on the latest version.
- Verify that Xamarin.Android.Support.v4 is on the latest version.
- Ensure all the Xamarin.Android.Support packages target the latest version.
- Clean or rebuild the application.
- In Visual Studio, try setting the maximum number of parallel project builds to 1. To do that, select **Options > Projects and Solutions > Build and Run > Maximum number of parallel projects builds**.
- If you're building from the command line and your command uses `/m`, try removing this element from the command.

Error: The name AuthenticationContinuationHelper doesn't exist in the current context

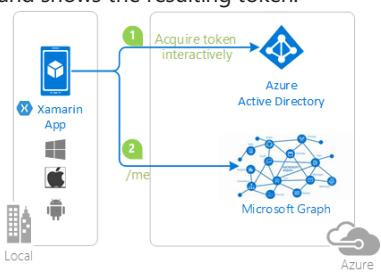
If an error indicates that `AuthenticationContinuationHelper` doesn't exist in the current context, Visual Studio might have incorrectly updated the `Android.csproj*` file. Sometimes the file path in the `<HintPath>` element incorrectly contains `netstandard13` instead of `monoandroid90`.

This example contains a correct file path:

```
<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

For more information, see the sample of a [Xamarin mobile application that uses Microsoft identity platform](#). The following table summarizes the relevant information in the README file.

SAMPLE	PLATFORM	DESCRIPTION
https://github.com/Azure-Samples/active-directory-xamarin-native-v2	Xamarin.iOS, Android, UWP	A simple Xamarin.Forms app that shows how to use MSAL to authenticate Microsoft personal accounts and Azure AD through the Azure AD 2.0 endpoint. The app also shows how to access Microsoft Graph and shows the resulting token.  The diagram illustrates the authentication and authorization process. It starts with a 'Xamarin App' icon containing a smartphone and tablet icon, labeled 'Local'. An arrow leads from the app to a box labeled 'Acquire token interactively' with a diamond icon. This leads to another box labeled 'Azure Active Directory' with a network graph icon. A final arrow points from the Azure AD box to a 'Microsoft Graph' box with a similar network graph icon. The path is numbered 1 and 2, corresponding to the steps in the text above.

Shared device mode for Android devices

4/12/2022 • 4 minutes to read • [Edit Online](#)

Frontline workers such as retail associates, flight crew members, and field service workers often use a shared mobile device to do their work. That becomes problematic when they start sharing passwords or pin numbers to access customer and business data on the shared device.

Shared device mode allows you to configure an Android device so that it can be easily shared by multiple employees. Employees can sign in and access customer information quickly. When they're finished with their shift or task, they can sign out of the device and it will be immediately ready for the next employee to use.

Shared device mode also provides Microsoft identity backed management of the device.

To create a shared device mode app, developers and cloud device admins work together:

- Developers write a single-account app (multiple-account apps aren't supported in shared device mode), add `"shared_device_mode_supported": true` to the app's configuration, and write code to handle things like shared device sign-out.
- Device admins prepare the device to be shared by installing the authenticator app, and setting the device to shared mode using the authenticator app. Only users who are in the [Cloud Device Administrator](#) role can put a device into shared mode by using the [Authenticator app](#). You can configure the membership of your organizational roles in the Azure portal via: [Azure Active Directory > Roles and Administrators > Cloud Device Administrator](#).

This article focuses primarily what developers should think about.

Single vs multiple-account applications

Applications written using the Microsoft Authentication Library (MSAL) SDK can manage a single account or multiple accounts. For details, see [single-account mode or multiple-account mode](#).

The Microsoft identity platform features available to your app vary depending on whether the application is running in single-account mode or multiple-account mode.

Shared device mode apps only work in single-account mode.

IMPORTANT

Applications that only support multiple-account mode can't run on a shared device. If an employee loads an app that doesn't support single-account mode, it won't run on the shared device.

Apps written before the MSAL SDK was released run in multiple-account mode and must be updated to support single-account mode before they can run on a shared mode device.

Supporting both single-account and multiple-accounts

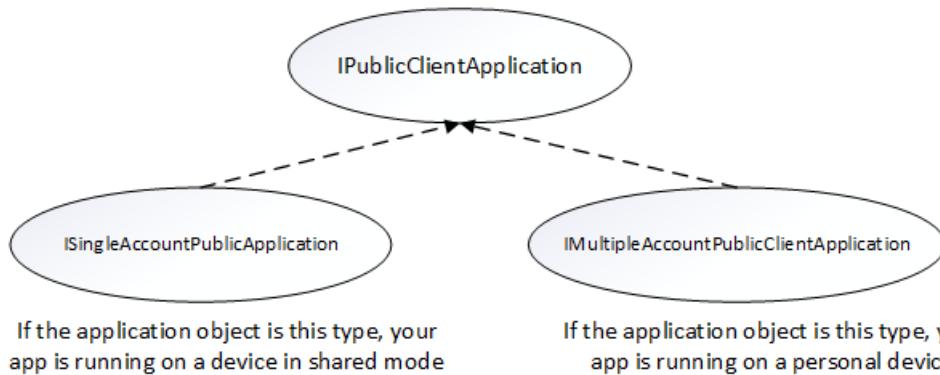
Your app can be built to support running on both personal devices and shared devices. If your app currently supports multiple accounts and you want to support shared device mode, add support for single account mode.

You may also want your app to change its behavior depending on the type of device it's running on. Use `ISingleAccountPublicClientApplication.isSharedDevice()` to determine when to run in single-account mode.

There are two different interfaces that represent the type of device your application is on. When you request an

application instance from MSAL's application factory, the correct application object is provided automatically.

The following object model illustrates the type of object you may receive and what it means in the context of a shared device:



You'll need to do a type check and cast to the appropriate interface when you get your `IPublicClientApplication` object. The following code checks for multiple account mode or single account mode, and casts the application object appropriately:

```
private IPublicClientApplication mApplication;

    // Running in personal-device mode?
    if (mApplication instanceof IMultipleAccountPublicClientApplication) {
        IMultipleAccountPublicClientApplication multipleAccountApplication =
            (IMultipleAccountPublicClientApplication) mApplication;
        ...
    }
    // Running in shared-device mode?
    } else if (mApplication instanceof ISingleAccountPublicClientApplication) {
        ISingleAccountPublicClientApplication singleAccountApplication =
            (ISingleAccountPublicClientApplication) mApplication;
        ...
    }
}
```

The following differences apply depending on whether your app is running on a shared or personal device:

	SHARED MODE DEVICE	PERSONAL DEVICE
Accounts	Single account	Multiple accounts
Sign-in	Global	Global
Sign-out	Global	Each application can control if the sign-out is local to the app or for the family of applications.
Supported account types	Work accounts only	Personal and work accounts supported

Why you may want to only support single-account mode

If you're writing an app that will only be used for frontline workers using a shared device, we recommend you write your application to only support single-account mode. This includes most applications that are task focused such as medical records apps, invoice apps, and most line-of-business apps. Only supporting single-account mode simplifies development because you won't need to implement the additional features that are part of multiple-account apps.

What happens when the device mode changes

If your application is running in multiple-account mode, and an administrator puts the device in shared device mode, all of the accounts on the device are cleared from the application and the application transitions to single-account mode.

Microsoft applications that support shared device mode

These Microsoft applications support Azure AD's shared device mode:

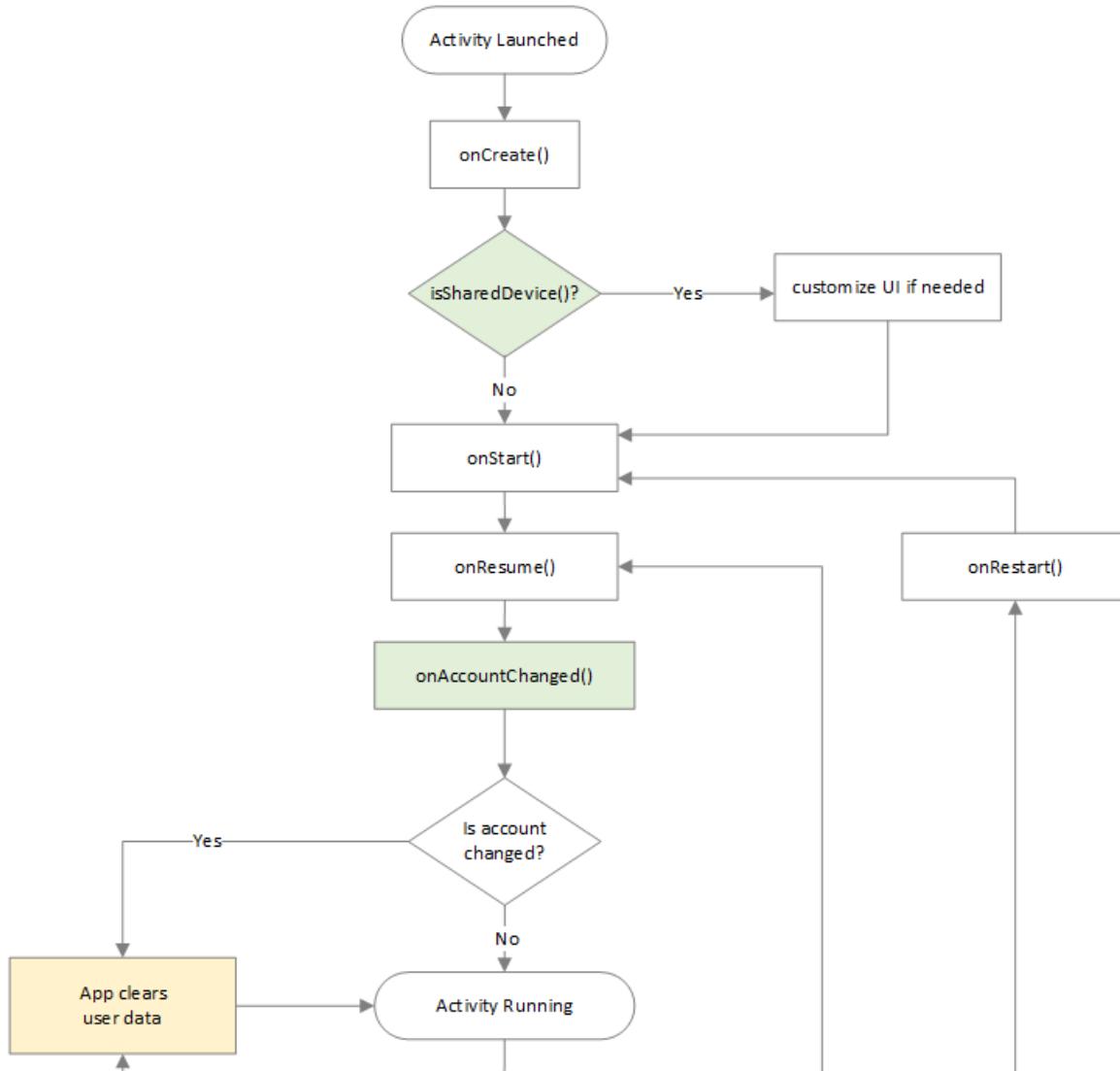
- Microsoft Teams
 - Microsoft Managed Home Screen app for Android Enterprise

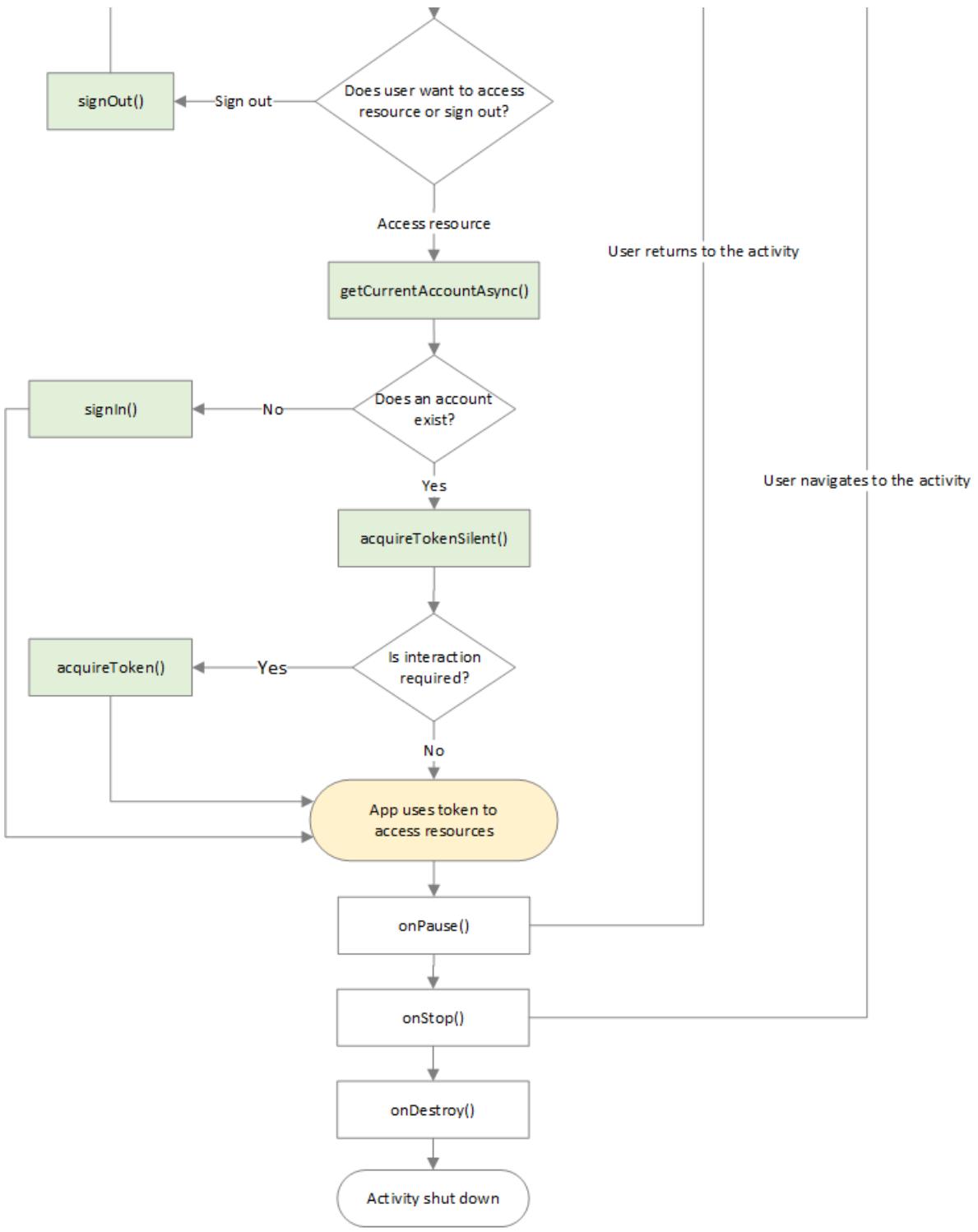
Shared device sign-out and the overall app lifecycle

When a user signs out, you'll need to take action to protect the privacy and data of the user. For example, if you're building a medical records app you'll want to make sure that when the user signs out previously displayed patient records are cleared. Your application must be prepared for data privacy and check every time it enters the foreground.

When your app uses MSAL to sign out the user in an app running on device that is in shared mode, the signed-in account and cached tokens are removed from both the app and the device.

The following diagram shows the overall app lifecycle and common events that may occur while your app runs. The diagram covers from the time an activity launches, signing in and signing out an account, and how events such as pausing, resuming, and stopping the activity fit in.





Next steps

For more information on how to run a frontline worker app on a shared mode on Android device, see:

- [Use shared-device mode in your Android application](#)

Microsoft Enterprise SSO plug-in for Apple devices (preview)

4/12/2022 • 13 minutes to read • [Edit Online](#)

IMPORTANT

This feature is in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Some features might be unsupported or have constrained capabilities. For more information, see [Supplemental terms of use for Microsoft Azure previews](#).

The *Microsoft Enterprise SSO plug-in for Apple devices* provides single sign-on (SSO) for Azure Active Directory (Azure AD) accounts on macOS, iOS, and iPadOS across all applications that support Apple's [enterprise single sign-on](#) feature. The plug-in provides SSO for even old applications that your business might depend on but that don't yet support the latest identity libraries or protocols. Microsoft worked closely with Apple to develop this plug-in to increase your application's usability while providing the best protection available.

The Enterprise SSO plug-in is currently a built-in feature of the following apps:

- [Microsoft Authenticator](#): iOS, iPadOS
- Microsoft Intune [Company Portal](#): macOS

Features

The Microsoft Enterprise SSO plug-in for Apple devices offers the following benefits:

- It provides SSO for Azure AD accounts across all applications that support the Apple Enterprise SSO feature.
- It can be enabled by any mobile device management (MDM) solution.
- It extends SSO to applications that don't yet use Microsoft identity platform libraries.
- It extends SSO to applications that use OAuth 2, OpenID Connect, and SAML.

Requirements

To use the Microsoft Enterprise SSO plug-in for Apple devices:

- The device must *support* and have an installed app that has the Microsoft Enterprise SSO plug-in for Apple devices:
 - iOS 13.0 and later: [Microsoft Authenticator app](#)
 - iPadOS 13.0 and later: [Microsoft Authenticator app](#)
 - macOS 10.15 and later: [Intune Company Portal app](#)
- The device must be *enrolled in MDM*, for example, through Microsoft Intune.
- Configuration must be *pushed to the device* to enable the Enterprise SSO plug-in. Apple requires this security constraint.

iOS requirements

- iOS 13.0 or higher must be installed on the device.
- A Microsoft application that provides the Microsoft Enterprise SSO plug-in for Apple devices must be installed on the device. For Public Preview, these applications are the [Microsoft Authenticator app](#).

macOS requirements

- macOS 10.15 or higher must be installed on the device.
- A Microsoft application that provides the Microsoft Enterprise SSO plug-in for Apple devices must be installed on the device. For Public Preview, these applications include the [Intune Company Portal app](#).

Enable the SSO plug-in

Use the following information to enable the SSO plug-in by using MDM.

Microsoft Intune configuration

If you use Microsoft Intune as your MDM service, you can use built-in configuration profile settings to enable the Microsoft Enterprise SSO plug-in:

1. Configure the [SSO app extension](#) settings of a configuration profile.
2. If the profile isn't already assigned, [assign the profile to a user or device group](#).

The profile settings that enable the SSO plug-in are automatically applied to the group's devices the next time each device checks in with Intune.

Manual configuration for other MDM services

If you don't use Intune for MDM, you can configure an Extensible Single Sign On profile payload for Apple devices. Use the following parameters to configure the Microsoft Enterprise SSO plug-in and its configuration options.

iOS settings:

- **Extension ID:** `com.microsoft.azureauthenticator.ssoextension`
- **Team ID:** This field isn't needed for iOS.

macOS settings:

- **Extension ID:** `com.microsoft.CompanyPortalMac.ssoextension`
- **Team ID:** `UBF8T346G9`

Common settings:

- **Type:** Redirect
 - `https://login.microsoftonline.com`
 - `https://login.microsoft.com`
 - `https://sts.windows.net`
 - `https://login.partner.microsoftonline.cn`
 - `https://login.chinacloudapi.cn`
 - `https://login.microsoftonline.de`
 - `https://login.microsoftonline.us`
 - `https://login.usgovcloudapi.net`
 - `https://login-us.microsoftonline.com`

More configuration options

You can add more configuration options to extend SSO functionality to other apps.

Enable SSO for apps that don't use a Microsoft identity platform library

The SSO plug-in allows any application to participate in SSO even if it wasn't developed by using a Microsoft SDK like Microsoft Authentication Library (MSAL).

The SSO plug-in is installed automatically by devices that have:

- Downloaded the Authenticator app on iOS or iPadOS, or downloaded the Intune Company Portal app on

macOS.

- Registered their device with your organization.

Your organization likely uses the Authenticator app for scenarios like multifactor authentication (MFA), passwordless authentication, and conditional access. By using an MDM provider, you can turn on the SSO plug-in for your applications. Microsoft has made it easy to configure the plug-in inside the Microsoft Endpoint Manager in Intune. An allowlist is used to configure these applications to use the SSO plug-in.

IMPORTANT

The Microsoft Enterprise SSO plug-in supports only apps that use native Apple network technologies or webviews. It doesn't support applications that ship their own network layer implementation.

Use the following parameters to configure the Microsoft Enterprise SSO plug-in for apps that don't use a Microsoft identity platform library.

Enable SSO for all managed apps

- **Key:** `Enable_SSO_On_All_ManagedApps`
- **Type:** `Integer`
- **Value:** 1 or 0 .

When this flag is on (its value is set to `1`), all MDM-managed apps not in the `AppBlockList` may participate in SSO.

Enable SSO for specific apps

- **Key:** `AppAllowList`
- **Type:** `String`
- **Value:** Comma-delimited list of application bundle IDs for the applications that are allowed to participate in SSO.
- **Example:** `com.contoso.workapp,com.contoso.travelapp`

NOTE

Safari and Safari View Service are allowed to participate in SSO by default. Can be configured *not* to participate in SSO by adding the bundle IDs of Safari and Safari View Service in AppBlockList. iOS Bundle IDs : [com.apple.mobilesafari, com.apple.SafariViewService] , macOS BundleID : com.apple.Safari

Enable SSO for all apps with a specific bundle ID prefix

- **Key:** `AppPrefixAllowList`
- **Type:** `String`
- **Value:** Comma-delimited list of application bundle ID prefixes for the applications that are allowed to participate in SSO. This parameter allows all apps that start with a particular prefix to participate in SSO.
- **Example:** `com.contoso.,com.fabrikam.`

Disable SSO for specific apps

- **Key:** `AppBlockList`
- **Type:** `String`
- **Value:** Comma-delimited list of application bundle IDs for the applications that are allowed not to participate in SSO.
- **Example:** `com.contoso.studyapp,com.contoso.travelapp`

To *disable* SSO for Safari or Safari View Service, you must explicitly do so by adding their bundle IDs to the

AppBlockList :

- iOS: `com.apple.mobilesafari`, `com.apple.SafariViewService`
- macOS: `com.apple.Safari`

Enable SSO through cookies for a specific application

Some apps that have advanced network settings might experience unexpected issues when they're enabled for SSO. For example, you might see an error indicating that a network request was canceled or interrupted.

If your users have problems signing in to an application even after you've enabled it through the other settings, try adding it to the `AppCookieSSOAllowList` to resolve the issues.

- **Key:** `AppCookieSSOAllowList`
- **Type:** `String`
- **Value:** Comma-delimited list of application bundle ID prefixes for the applications that are allowed to participate in the SSO. All apps that start with the listed prefixes will be allowed to participate in SSO.
- **Example:** `com.contoso.myapp1,com.fabrikam.myapp2`

Other requirements: To enable SSO for applications by using `AppCookieSSOAllowList`, you must also add their bundle ID prefixes `AppPrefixAllowList`.

Try this configuration only for applications that have unexpected sign-in failures.

Summary of keys

KEY	TYPE	VALUE
<code>Enable_SSO_On_All_ManagedApps</code>	Integer	<code>1</code> to enable SSO for all managed apps, <code>0</code> to disable SSO for all managed apps.
<code>AppAllowList</code>	String (comma-delimited list)	Bundle IDs of applications allowed to participate in SSO.
<code>AppBlockList</code>	String (comma-delimited list)	Bundle IDs of applications not allowed to participate in SSO.
<code>AppPrefixAllowList</code>	String (comma-delimited list)	Bundle ID prefixes of applications allowed to participate in SSO.
<code>AppCookieSSOAllowList</code>	String (comma-delimited list)	Bundle ID prefixes of applications allowed to participate in SSO but that use special network settings and have trouble with SSO using the other settings. Apps you add to <code>AppCookieSSOAllowList</code> must also be added to <code>AppPrefixAllowList</code> .

Settings for common scenarios

- **Scenario:** I want to enable SSO for most managed applications, but not for all of them.

KEY	VALUE
<code>Enable_SSO_On_All_ManagedApps</code>	<code>1</code>
<code>AppBlockList</code>	The bundle IDs (comma-delimited list) of the apps you want to prevent from participating in SSO.

- *Scenario*: I want to disable SSO for Safari, which is enabled by default, but enable SSO for all managed apps.

KEY	VALUE
Enable_SSO_On_All_ManagedApps	1
AppBlockList	The bundle IDs (comma-delimited list) of the Safari apps you want to prevent from participating in SSO. <ul style="list-style-type: none"> • For iOS: com.apple.mobilesafari , com.apple.SafariViewService • For macOS: com.apple.Safari

- *Scenario*: I want to enable SSO on all managed apps and few unmanaged apps, but disable SSO for a few other apps.

KEY	VALUE
Enable_SSO_On_All_ManagedApps	1
AppAllowList	The bundle IDs (comma-delimited list) of the apps you want to enable for participation in for SSO.
AppBlockList	The bundle IDs (comma-delimited list) of the apps you want to prevent from participating in SSO.

Find app bundle identifiers on iOS devices

Apple provides no easy way to get bundle IDs from the App Store. The easiest way to get the bundle IDs of the apps you want to use for SSO is to ask your vendor or app developer. If that option isn't available, you can use your MDM configuration to find the bundle IDs:

1. Temporarily enable the following flag in your MDM configuration:

- **Key:** admin_debug_mode_enabled
- **Type:** Integer
- **Value:** 1 or 0

2. When this flag is on, sign in to iOS apps on the device for which you want to know the bundle ID.

3. In the Authenticator app, select **Help > Send logs > View logs**.

4. In the log file, look for following line:

[ADMIN MODE] SSO extension has captured following app bundle identifiers . This line should capture all application bundle IDs that are visible to the SSO extension.

Use the bundle IDs to configure SSO for the apps.

Allow users to sign in from unknown applications and the Safari browser

By default, the Microsoft Enterprise SSO plug-in provides SSO for authorized apps only when a user has signed in from an app that uses a Microsoft identity platform library like MSAL. The Microsoft Enterprise SSO plug-in can also acquire a shared credential when it's called by another app that uses a Microsoft identity platform library during a new token acquisition.

When you enable the browser_sso_interaction_enabled flag, apps that don't use a Microsoft identity platform library can do the initial bootstrapping and get a shared credential. The Safari browser can also do the initial bootstrapping and get a shared credential.

If the Microsoft Enterprise SSO plug-in doesn't have a shared credential yet, it will try to get one whenever a sign-in is requested from an Azure AD URL inside the Safari browser, ASWebAuthenticationSession, SafariViewController, or another permitted native application.

Use these parameters to enable the flag:

- **Key:** `browser_sso_interaction_enabled`
- **Type:** `Integer`
- **Value:** 1 or 0

macOS requires this setting so it can provide a consistent experience across all apps. iOS and iPadOS don't require this setting because most apps use the Authenticator application for sign-in. But we recommend that you enable this setting because if some of your applications don't use the Authenticator app on iOS or iPadOS, this flag will improve the experience. The setting is disabled by default.

Disable asking for MFA during initial bootstrapping

By default, the Microsoft Enterprise SSO plug-in always prompts the user for MFA during the initial bootstrapping and while getting a shared credential. The user is prompted for MFA even if it's not required for the application that the user has opened. This behavior allows the shared credential to be easily used across all other applications without the need to prompt the user if MFA is required later. Because the user gets fewer prompts overall, this setup is generally a good decision.

Enabling `browser_sso_disable_mfa` turns off MFA during initial bootstrapping and while getting the shared credential. In this case, the user is prompted only when MFA is required by an application or resource.

To enable the flag, use these parameters:

- **Key:** `browser_sso_disable_mfa`
- **Type:** `Integer`
- **Value:** 1 or 0

We recommend keeping this flag disabled because it reduces the number of times the user is prompted to sign in. If your organization rarely uses MFA, you might want to enable the flag. But we recommend that you use MFA more frequently instead. For this reason, the flag is disabled by default.

Disable OAuth 2 application prompts

If an application prompts your users to sign in even though the Microsoft Enterprise SSO plug-in works for other applications on the device, the app might be bypassing SSO at the protocol layer. Shared credentials are also ignored by such applications because the plug-in provides SSO by appending the credentials to network requests made by allowed applications.

These parameters specify whether the SSO extension should prevent native and web applications from bypassing SSO at the protocol layer and forcing the display of a sign-in prompt to the user.

For a consistent SSO experience across all apps on the device, we recommend you enable one of these settings, which are disabled by default.

Disable the app prompt and display the account picker:

- **Key:** `disable_explicit_app_prompt`
- **Type:** `Integer`
- **Value:** 1 or 0

Disable app prompt and select an account from the list of matching SSO accounts automatically:

- **Key:** `disable_explicit_app_prompt_and_autologin`
- **Type:** `Integer`

- **Value:** 1 or 0

Use Intune for simplified configuration

You can use Intune as your MDM service to ease configuration of the Microsoft Enterprise SSO plug-in. For example, you can use Intune to enable the plug-in and add old apps to an allowlist so they get SSO.

For more information, see the [Intune configuration documentation](#).

Use the SSO plug-in in your application

[MSAL for Apple devices](#) versions 1.1.0 and later supports the Microsoft Enterprise SSO plug-in for Apple devices. It's the recommended way to add support for the Microsoft Enterprise SSO plug-in. It ensures you get the full capabilities of the Microsoft identity platform.

If you're building an application for frontline-worker scenarios, see [Shared device mode for iOS devices](#) for setup information.

Understand how the SSO plug-in works

The Microsoft Enterprise SSO plug-in relies on the [Apple Enterprise SSO framework](#). Identity providers that join the framework can intercept network traffic for their domains and enhance or change how those requests are handled. For example, the SSO plug-in can show more UIs to collect end-user credentials securely, require MFA, or silently provide tokens to the application.

Native applications can also implement custom operations and communicate directly with the SSO plug-in. For more information, see this [2019 Worldwide Developer Conference video from Apple](#).

Applications that use MSAL

[MSAL for Apple devices](#) versions 1.1.0 and later supports the Microsoft Enterprise SSO plug-in for Apple devices natively for work and school accounts.

You don't need any special configuration if you followed [all recommended steps](#) and used the default [redirect URI format](#). On devices that have the SSO plug-in, MSAL automatically invokes it for all interactive and silent token requests. It also invokes it for account enumeration and account removal operations. Because MSAL implements a native SSO plug-in protocol that relies on custom operations, this setup provides the smoothest native experience to the end user.

If the SSO plug-in isn't enabled by MDM but the Microsoft Authenticator app is present on the device, MSAL instead uses the Authenticator app for any interactive token requests. The SSO plug-in shares SSO with the Authenticator app.

Applications that don't use MSAL

Applications that don't use a Microsoft identity platform library, like MSAL, can still get SSO if an administrator adds these applications to the allowlist.

You don't need to change the code in those apps as long as the following conditions are satisfied:

- The application uses Apple frameworks to run network requests. These frameworks include [WKWebView](#) and [NSURLSession](#), for example.
- The application uses standard protocols to communicate with Azure AD. These protocols include, for example, OAuth 2, SAML, and WS-Federation.
- The application doesn't collect plaintext usernames and passwords in the native UI.

In this case, SSO is provided when the application creates a network request and opens a web browser to sign the user in. When a user is redirected to an Azure AD sign-in URL, the SSO plug-in validates the URL and checks for an SSO credential for that URL. If it finds the credential, the SSO plug-in passes it to Azure AD, which

authorizes the application to complete the network request without asking the user to enter credentials. Additionally, if the device is known to Azure AD, the SSO plug-in passes the device certificate to satisfy the device-based conditional access check.

To support SSO for non-MSAL apps, the SSO plug-in implements a protocol similar to the Windows browser plug-in described in [What is a primary refresh token?](#).

Compared to MSAL-based apps, the SSO plug-in acts more transparently for non-MSAL apps. It integrates with the existing browser sign-in experience that apps provide.

The end user sees the familiar experience and doesn't have to sign in again in each application. For example, instead of displaying the native account picker, the SSO plug-in adds SSO sessions to the web-based account picker experience.

Next steps

Learn about [Shared device mode for iOS devices](#).

Considerations for using Xamarin iOS with MSAL.NET

4/12/2022 • 5 minutes to read • [Edit Online](#)

When you use the Microsoft Authentication Library for .NET (MSAL.NET) on Xamarin iOS, you should:

- Override and implement the `OpenUrl` function in `AppDelegate`.
- Enable keychain groups.
- Enable token cache sharing.
- Enable keychain access.
- Understand known issues with iOS 12 and iOS 13 and authentication.

Implement OpenUrl

Override the `OpenUrl` method of the `FormsApplicationDelegate` derived class and call `AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs`. Here's an example:

```
public override bool OpenUrl(UIApplication app, NSURL url, NSDictionary options)
{
    AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(url);
    return true;
}
```

Also, perform the following tasks:

- Define a redirect URI scheme.
- Require permissions for your app to call another app.
- Have a specific form for the redirect URI.
- [Register a redirect URI](#) in the Azure portal.

Enable keychain access

To enable keychain access, make sure that your application has a keychain access group. You can set the keychain access group when you create your application by using the `WithIosKeychainSecurityGroup()` API.

To benefit from the cache and single sign-on (SSO), set the keychain access group to the same value in all of your applications.

This example of the setup uses MSAL 4.x:

```
var builder = PublicClientApplicationBuilder
    .Create(ClientId)
    .WithIosKeychainSecurityGroup("com.microsoft.adalcache")
    .Build();
```

Also enable keychain access in the `Entitlements.plist` file. Use either the following access group or your own access group.

```

<dict>
  <key>keychain-access-groups</key>
  <array>
    <string>$(AppIdentifierPrefix)com.microsoft.adalcache</string>
  </array>
</dict>

```

When you use the `WithIosKeychainSecurityGroup()` API, MSAL automatically appends your security group to the end of the application's *team ID* (`AppIdentifierPrefix`). MSAL adds your security group because when you build your application in Xcode, it will do the same. That's why the entitlements in the `Entitlements.plist` file need to include `$(AppIdentifierPrefix)` before the keychain access group.

For more information, see the [iOS entitlements documentation](#).

Troubleshooting KeyChain access

If you get an error message similar to "The application cannot access the iOS keychain for the application publisher (the TeamId is null)", this means MSAL is not able to access the KeyChain. This is a configuration issue. To troubleshoot, try to access the KeyChain on your own, for example:

```

var queryRecord = new SecRecord(SecKind.GenericPassword)
{
  Service = "",
  Account = "SomeTeamId",
  Accessible = SecAccessible.Always
};

SecRecord match = SecKeyChain.QueryAsRecord(queryRecord, out SecStatusCode resultCode);

if (resultCode == SecStatusCode.ItemNotFound)
{
  SecKeyChain.Add(queryRecord);
  match = SecKeyChain.QueryAsRecord(queryRecord, out resultCode);
}

// Make sure that resultCode == SecStatusCode.Success

```

Enable token cache sharing across iOS applications

Starting in MSAL 2.x, you can specify a keychain access group to persist the token cache across multiple applications. This setting enables you to share the token cache among several applications that have the same keychain access group. You can share the token cache among [ADAL.NET](#) applications, MSAL.NET Xamarin.iOS applications, and native iOS applications that were developed in [ADALobjc](#) or [MSALobjc](#).

By sharing the token cache, you allow single sign-on (SSO) among all of the applications that use the same keychain access group.

To enable this cache sharing, use the `WithIosKeychainSecurityGroup()` method to set the keychain access group to the same value in all applications that share the same cache. The first code example in this article shows how to use the method.

Earlier in this article, you learned that MSAL adds `$(AppIdentifierPrefix)` whenever you use the `WithIosKeychainSecurityGroup()` API. MSAL adds this element because the team ID `AppIdentifierPrefix` ensures that only applications that are made by the same publisher can share keychain access.

NOTE

The `KeychainSecurityGroup` property is deprecated. Use the `iosKeychainSecurityGroup` property instead. The `TeamId` prefix is not required when you use `iosKeychainSecurityGroup`.

Use Microsoft Authenticator

Your application can use Microsoft Authenticator as a broker to enable:

- **SSO:** When you enable SSO, your users don't need to sign in to each application.
- **Device identification:** Use device identification to authenticate by accessing the device certificate. This certificate is created on the device when it's joined to the workplace. Your application will be ready if the tenant administrators enable conditional access related to the devices.
- **Application identification verification:** When an application calls the broker, it passes its redirect URL. The broker verifies the redirect URL.

For details about how to enable a broker, see [Use Microsoft Authenticator or Microsoft Intune Company Portal on Xamarin iOS and Android applications](#).

Known issues with iOS 12 and authentication

Microsoft released a [security advisory](#) about an incompatibility between iOS 12 and some types of authentication. The incompatibility breaks social, WSFed, and OIDC sign-ins. The security advisory helps you understand how to remove ASP.NET security restrictions from your applications to make them compatible with iOS 12.

When you develop MSAL.NET applications on Xamarin iOS, you might see an infinite loop when you try to sign in to websites from iOS 12. Such behavior is similar to this ADAL issue on GitHub: [Infinite loop when trying to login to website from iOS 12 #1329](#).

You might also see a break in ASP.NET Core OIDC authentication with iOS 12 Safari. For more information, see this [WebKit issue](#).

Known issues with iOS 13 and authentication

If your app requires conditional access or certificate authentication support, enable your app to communicate with the Microsoft Authenticator broker app. MSAL is then responsible for handling requests and responses between your application and Microsoft Authenticator.

On iOS 13, Apple made a breaking API change by removing the application's ability to read the source application when receiving a response from an external application through custom URL schemes.

Apple's documentation for [UIApplicationOpenURLOptionsSourceApplicationKey](#) states:

If the request originated from another app belonging to your team, UIKit sets the value of this key to the ID of that app. If the team identifier of the originating app is different than the team identifier of the current app, the value of the key is nil.

This change is breaking for MSAL because it relied on `UIApplication.SharedApplication.OpenUrl` to verify communication between MSAL and the Microsoft Authenticator app.

Additionally, on iOS 13, the developer is required to provide a presentation controller when using `ASWebAuthenticationSession`.

Your app is impacted if you're building with Xcode 11 and you use either iOS broker or `ASWebAuthenticationSession`.

In such cases, use [MSAL.NET 4.4.0+](#) to enable successful authentication.

Additional requirements

- When using the latest MSAL libraries, ensure that **Microsoft Authenticator version 6.3.19+** is installed on the device.

- When updating to MSAL.NET 4.4.0+, update the your `LSApplicationQueriesSchemes` in the `Info.plist` file and add `msauthv3`:

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

Adding `msauthv3` to `Info.plist` is necessary to detect the presence of the latest Microsoft Authenticator app on the device that supports iOS 13.

Report an issue

If you have questions or would like to report an issue you've found in MSAL.NET, open an issue in the [AzureAD/microsoft-authentication-library-for-dotnet](#) repository on GitHub.

Next steps

For information about properties for Xamarin iOS, see the [iOS-specific considerations](#) paragraph of the following sample's README.md file:

SAMPLE	PLATFORM	DESCRIPTION
https://github.com/Azure-Samples/active-directory-xamarin-native-v2	Xamarin iOS, Android, Universal Windows Platform (UWP)	A simple Xamarin Forms app that shows how to use MSAL to authenticate Microsoft personal accounts and Azure AD via the Azure AD 2.0 endpoint. The app also shows how to use the resulting token to access Microsoft Graph.

Shared device mode for iOS devices

4/12/2022 • 6 minutes to read • [Edit Online](#)

IMPORTANT

This feature is in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Some features might be unsupported or have constrained capabilities. For more information, see [Supplemental terms of use for Microsoft Azure previews](#).

Frontline workers such as retail associates, flight crew members, and field service workers often use a shared mobile device to perform their work. These shared devices can present security risks if your users share their passwords or PINs, intentionally or not, to access customer and business data on the shared device.

Shared device mode allows you to configure an iOS 13 or higher device to be more easily and securely shared by employees. Employees can sign in and access customer information quickly. When they're finished with their shift or task, they can sign out of the device and it's immediately ready for use by the next employee.

Shared device mode also provides Microsoft identity-backed management of the device.

This feature uses the [Microsoft Authenticator app](#) to manage the users on the device and to distribute the [Microsoft Enterprise SSO plug-in for Apple devices](#).

Create a shared device mode app

To create a shared device mode app, developers and cloud device admins work together:

1. **Application developers** write a single-account app (multiple-account apps are not supported in shared device mode) and write code to handle things like shared device sign-out.
2. **Device administrators** prepare the device to be shared by using a mobile device management (MDM) provider like Microsoft Intune to manage the devices in their organization. The MDM pushes the Microsoft Authenticator app to the devices and turns on "Shared Mode" for each device through a profile update to the device. This Shared Mode setting is what changes the behavior of the supported apps on the device. This configuration from the MDM provider sets the shared device mode for the device and enables the [Microsoft Enterprise SSO plug-in for Apple devices](#) which is required for shared device mode.
3. **[Required during Public Preview only]** A user with [Cloud Device Administrator](#) role must then launch the [Microsoft Authenticator app](#) and join their device to the organization.

To configure the membership of your organizational roles in the Azure portal: **Azure Active Directory** > **Roles and Administrators** > **Cloud Device Administrator**

The following sections help you update your application to support shared device mode.

Use Intune to enable shared device mode & SSO extension

NOTE

The following step is required only during public preview.

Your device needs to be configured to support shared device mode. It must have iOS 13+ installed and be

MDM-enrolled. MDM configuration also needs to enable [Microsoft Enterprise SSO plug-in for Apple devices](#). To learn more about SSO extensions, see the [Apple video](#).

1. In the Intune Configuration Portal, tell the device to enable the [Microsoft Enterprise SSO plug-in for Apple devices](#) with the following configuration:

- **Type:** Redirect
- **Extension ID:** com.microsoft.azureauthenticator.ssoextension
- **Team ID:** (this field is not needed for iOS)
- **URLs:**
 - <https://login.microsoftonline.com>
 - <https://login.microsoft.com>
 - <https://sts.windows.net>
 - <https://login.partner.microsoftonline.cn>
 - <https://login.chinacloudapi.cn>
 - <https://login.microsoftonline.de>
 - <https://login.microsoftonline.us>
 - <https://login.usgovcloudapi.net>
 - <https://login-us.microsoftonline.com>
- **Additional Data to configure:**
 - Key: sharedDeviceMode
 - Type: Boolean
 - Value: true

For more information about configuring with Intune, see the [Intune configuration documentation](#).

2. Next, configure your MDM to push the Microsoft Authenticator app to your device through an MDM profile.

Set the following configuration options to turn on Shared Device mode:

- Configuration 1:
 - Key: sharedDeviceMode
 - Type: Boolean
 - Value: true

Modify your iOS application to support shared device mode

Your users depend on you to ensure their data isn't leaked to another user. The following sections provide helpful signals to indicate to your application that a change has occurred and should be handled.

You are responsible for checking the state of the user on the device every time your app is used, and then clearing the previous user's data. This includes if it is reloaded from the background in multi-tasking.

On a user change, you should ensure both the previous user's data is cleared and that any cached data being displayed in your application is removed. We highly recommend you and your company conduct a security review process after updating your app to support shared device mode.

Detect shared device mode

Detecting shared device mode is important for your application. Many applications will require a change in their user experience (UX) when the application is used on a shared device. For example, your application might have a "Sign-Up" feature, which isn't appropriate for a frontline worker because they likely already have an account. You may also want to add extra security to your application's handling of data if it's in shared device mode.

Use the `getDeviceInformationWithParameters:completionBlock:` API in the `MSALPublicClientApplication` to determine if an app is running on a device in shared device mode.

The following code snippets show examples of using the `getDeviceInformationWithParameters:completionBlock:` API.

Swift

```
application.getDeviceInformation(with: nil, completionBlock: { (deviceInformation, error) in

    guard let deviceInfo = deviceInformation else {
        return
    }

    let isSharedDevice = deviceInfo.deviceMode == .shared
    // Change your app UX if needed
})
```

Objective-C

```
[application getDeviceInformationWithParameters:nil
                                         completionBlock:^(MSALDeviceInformation * _Nullable deviceInformation,
NSError * _Nullable error)
{
    if (!deviceInformation)
    {
        return;
    }

    BOOL isSharedDevice = deviceInformation.deviceMode == MSALDeviceModeShared;
    // Change your app UX if needed
}];
```

Get the signed-in user and determine if a user has changed on the device

Another important part of supporting shared device mode is determining the state of the user on the device and clearing application data if a user has changed or if there is no user at all on the device. You are responsible for ensuring data isn't leaked to another user.

You can use `getCurrentAccountWithParameters:completionBlock:` API to query the currently signed-in account on the device.

Swift

```
let msalParameters = MSALParameters()
msalParameters.completionBlockQueue = DispatchQueue.main

application.getCurrentAccount(with: msalParameters, completionBlock: { (currentAccount, previousAccount,
error) in

    // currentAccount is the currently signed in account
    // previousAccount is the previously signed in account if any
})
```

Objective-C

```
MSALParameters *parameters = [MSALParameters new];
parameters.completionBlockQueue = dispatch_get_main_queue();

[application getCurrentAccountWithParameters:parameters
    completionBlock:^(MSALAccount * _Nullable account, MSALAccount * _Nullable previousAccount, NSError * _Nullable error)
{
    // currentAccount is the currently signed in account
    // previousAccount is the previously signed in account if any
}];
```

Globally sign in a user

When a device is configured as a shared device, your application can call the

`acquireTokenWithParameters:completionBlock:` API to sign in the account. The account will be available globally for all eligible apps on the device after the first app signs in the account.

Objective-C

```
MSALInteractiveTokenParameters *parameters = [[MSALInteractiveTokenParameters alloc]
initWithScopes:@[@"api://myapi/scopes"] webViewParameters:[self msalTestWebViewParameters]];

parameters.loginHint = self.loginHintTextField.text;

[application acquireTokenWithParameters:parameters completionBlock:completionBlock];
```

Globally sign out a user

The following code removes the signed-in account and clears cached tokens from not only the app, but also from the device that's in shared device mode. It does not, however, clear the *data* from your application. You must clear the data from your application, as well as clear any cached data your application may be displaying to the user.

Clear browser state

NOTE

The following step is required only during public preview.

In this public preview version, the [Microsoft Enterprise SSO plug-in for Apple devices](#) clears state only for applications. It does not clear state on the Safari browser. We recommend you manually clear browser session to ensure no traces of user state are left behind. You can use the optional `signoutFromBrowser` property shown below to clear any cookies. This will cause the browser to briefly launch on the device.

Swift

```

let account = .... /* account retrieved above */

let signoutParameters = MSALSignoutParameters(webviewParameters: self.webViewParamaters!)
signoutParameters.signoutFromBrowser = true // Only needed for Public Preview.

application.signout(with: account, signoutParameters: signoutParameters, completionBlock: {(success, error)
in

    if let error = error {
        // Signout failed
        return
    }

    // Sign out completed successfully
})

```

Objective-C

```

MSALAccount *account = ... /* account retrieved above */;

MSALSignoutParameters *signoutParameters = [[MSALSignoutParameters alloc]
initWithWebviewParameters:webViewParameters];
signoutParameters.signoutFromBrowser = YES; // Only needed for Public Preview.

[application signoutWithAccount:account signoutParameters:signoutParameters completionBlock:^(BOOL success,
NSError * _Nullable error)
{
    if (!success)
    {
        // Signout failed
        return;
    }

    // Sign out completed successfully
}];

```

Next steps

To see shared device mode in action, the following code sample on GitHub includes an example of running a frontline worker app on an iOS device in shared device mode:

[MSAL iOS Swift Microsoft Graph API Sample](#)

Get a token for a mobile app that calls web APIs

4/12/2022 • 6 minutes to read • [Edit Online](#)

Before your app can call protected web APIs, it needs an access token. This article walks you through the process to get a token by using the Microsoft Authentication Library (MSAL).

Define a scope

When you request a token, define a scope. The scope determines what data your app can access.

The easiest way to define a scope is to combine the desired web API's `App ID URI` with the scope `.default`. This definition tells the Microsoft identity platform that your app requires all scopes that are 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. When you call `AcquireTokenSilent()` or `AcquireTokenInteractive()`, MSAL returns an access token for the requested scopes. The correct pattern is to make a silent request and then 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 requests.
sampleApp.acquireToken(getActivity(), SCOPES, getAuthInteractiveCallback());

```

iOS

First try acquiring a token silently:

```

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 isEqualToString:MSALErrorDomain] && error.code == MSALErrorInteractionRequired)
    {
        // Interactive auth will be required, call acquireTokenWithParameters:error:
        return;
    }
}];

```

```

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
}

```

If MSAL returns `MSALErrorInteractionRequired`, then try acquiring tokens interactively:

```

UIViewController *viewController = ...; // Pass a reference to the view controller that should be used when
getting a token interactively
MSALWebviewParameters *webParameters = [[MSALWebviewParameters alloc]
initWithAuthPresentationViewController: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;
}
}];

```

```

let viewController = ... // Pass a reference to the view controller that should be used when getting a token
interactively
let webviewParameters = MSALWebviewParameters(authPresentationViewController: 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 to get a token interactively or silently:

- [Common parameters for getting a token](#)
- [Parameters for getting an interactive token](#)
- [Parameters for getting a silent token](#)

Xamarin

The following example shows the minimal code to get a token interactively. The example uses Microsoft Graph to read the user's profile.

```

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`. The `scopes` parameter enumerates strings that define the scopes for which a token is required. If the token is for Microsoft Graph, you can find the required scopes in the API reference of each Microsoft Graph API. In the reference, go to the "Permissions" section.

For example, to [list the user's contacts](#), use the scope "User.Read", "Contacts.Read". For more information, see [Microsoft Graph permissions reference](#).

On Android, you can specify parent activity when you create the app by using `PublicClientApplicationBuilder`. If you don't specify the parent activity at that time, later you can specify it by using `.WithParentActivityOrWindow` as in the following section. If you specify parent activity, then the token gets back to that parent activity after the interaction. If you don't specify it, then the `.ExecuteAsync()` call throws an exception.

Specific optional parameters in MSAL.NET

The following sections explain the optional parameters in MSAL.NET.

WithPrompt

The `withPrompt()` parameter controls 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 field is preceded by a small blue icon representing a class or struct.

The class defines the following constants:

- `SelectAccount` forces the security token service (STS) to present the account-selection dialog box. The dialog box contains the accounts for which the user has a session. You can use this option when you want to let the user choose among different identities. This option drives MSAL to send `prompt=select_account` to the identity provider.

The `SelectAccount` constant is the default, and it effectively provides the best possible experience based on the available information. The available information might include account, presence of a session for the user, and so on. Don't change this default unless you have a good reason to do it.

- `Consent` enables you to prompt the user for consent even if consent was granted before. In this case, MSAL sends `prompt=consent` to the identity provider.

You might want to use the `Consent` constant in security-focused applications where the organization governance requires users to see the consent dialog box each time they use the application.

- `ForceLogin` enables the service to prompt the user for credentials even if the prompt isn't needed.

This option can be useful if the token acquisition fails and you want to let the user sign in again. In this case, MSAL sends `prompt=login` to the identity provider. You might want to use this option in security-focused applications where the organization governance requires the user to sign in each time they access specific parts of the application.

- `Never` is for only .NET 4.5 and Windows Runtime (WinRT). This constant won't prompt the user, but it will try to use the cookie that's stored in the hidden embedded web view. For more information, see [Using web browsers with MSAL.NET](#).

If this option fails, then `AcquireTokenInteractive` throws an exception to notify you that a UI interaction is needed. Then use another `Prompt` parameter.

- `NoPrompt` doesn't send a prompt to the identity provider.

This option is useful only for edit-profile policies in Azure Active Directory B2C. For more information, see [B2C specifics](#).

`WithExtraScopeToConsent`

Use the `WithExtraScopeToConsent` modifier in an advanced scenario where you want the user to provide upfront consent to several resources. You can use this modifier when you don't want to use incremental consent, which is normally used with MSAL.NET or the Microsoft identity platform. For more information, see [Have the user consent upfront for several resources](#).

Here's a code example:

```
var result = await app.AcquireTokenInteractive(scopesForCustomerApi)
    .WithExtraScopeToConsent(scopesForVendorApi)
    .ExecuteAsync();
```

Other optional parameters

To learn about the other optional parameters for `AcquireTokenInteractive`, see the [reference documentation for `AcquireTokenInteractiveParameterBuilder`](#).

Acquire tokens via the protocol

We don't recommend directly using the protocol to get tokens. If you do, then the app won't support some scenarios that involve single sign-on (SSO), device management, and conditional access.

When you use the protocol to get tokens for mobile apps, make two requests:

- Get an authorization code.
- Exchange the code for a token.

Get an 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 the 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=0AAABAAAiL9Kn2Z27UubvWFPbm0gLWQJVzCTE9UkP3pSx1aXxUjq3n8b2JRLk40xVxr...
&redirect_uri=<ENCODED_REDIRECT_URI>
&grant_type=authorization_code
```

Next steps

Move on to the next article in this scenario, [Calling a web API](#).

Use Microsoft Authenticator or Intune Company Portal on Xamarin applications

4/12/2022 • 10 minutes to read • [Edit Online](#)

On Android and iOS, brokers like Microsoft Authenticator and the Android-specific Microsoft Intune Company Portal enable:

- **Single sign-on (SSO)**: Users don't need to sign in to each application.
- **Device identification**: The broker accesses the device certificate. This certificate is created on the device when it's joined to the workplace.
- **Application identification verification**: When an application calls the broker, it passes its redirect URL. The broker verifies the URL.

To enable one of these features, use the `.WithBroker()` parameter when you call the `PublicClientApplicationBuilder.CreateApplication` method. The `.WithBroker()` parameter is set to true by default.

The setup of brokered authentication in the Microsoft Authentication Library for .NET (MSAL.NET) varies by platform:

- [iOS applications](#)
- [Android applications](#)

Brokered authentication for iOS

Use the following steps to enable your Xamarin.iOS app to communicate with the [Microsoft Authenticator](#) app. If you're targeting iOS 13, consider reading about [Apple's breaking API change](#).

Step 1: Enable broker support

You must enable broker support for individual instances of `PublicClientApplication`. Support is disabled by default. When you create `PublicClientApplication` through `PublicClientApplicationBuilder`, use the `.WithBroker()` parameter as the following example shows. The `.WithBroker()` parameter is set to true by default.

```
var app = PublicClientApplicationBuilder
    .Create(ClientId)
    .WithBroker()
    .WithReplyUri(redirectUriOnIos) // $"msauth.{Bundle.Id}://auth" (see step 6 below)
    .Build();
```

Step 2: Enable keychain access

To enable keychain access, you must have a keychain access group for your application. You can use the `.WithIosKeychainSecurityGroup()` API to set your keychain access group when you create your application:

```
var builder = PublicClientApplicationBuilder
    .Create(ClientId)
    .WithIosKeychainSecurityGroup("com.microsoft.adalcache")
    .Build();
```

For more information, see [Enable keychain access](#).

Step 3: Update AppDelegate to handle the callback

When MSAL.NET calls the broker, the broker calls back to your application through the `OpenUrl` method of the `AppDelegate` class. Because MSAL waits for the broker's response, 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 started. It's used as an opportunity to process the response from the broker and complete the authentication process that MSAL.NET started.

Step 4: Set UIViewController()

Still in the `AppDelegate.cs` file, set an object window. You don't typically need to set the object window for Xamarin iOS, but you do need an object window to send and receive responses from the broker.

To set up the object window:

1. In the `AppDelegate.cs` file, set `App.RootViewController` to a new `UIViewController()`. This assignment ensures that the call to the broker includes `UIViewController`. If this setting is assigned incorrectly, 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 then pass in the reference to the object window you'll use.

In `App.cs`:

```
public static object RootViewController { get; set; }
```

In `AppDelegate.cs`:

```
LoadApplication(new App());
App.RootViewController = new UIViewController();
```

In the `AcquireToken` call:

```
result = await app.AcquireTokenInteractive(scopes)
    .WithParentActivityOrWindow(App.RootViewController)
    .ExecuteAsync();
```

Step 5: Register a URL scheme

MSAL.NET uses URLs to invoke the broker and then return the broker response to your app. To complete the round trip, register a URL scheme for your app in the *Info.plist* file.

The `CFBundleURLSchemes` name must include `msauth.` as a prefix. Follow the prefix with `CFBundleURLName`.

In the URL scheme, `BundleId` uniquely identifies the app: `$"msauth.(BundleId)"`. So if `BundleId` is `com.yourcompany.xforms`, then the URL scheme is `msauth.com.yourcompany.xforms`.

NOTE

This URL scheme becomes part of the redirect URI that uniquely identifies 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 6: Add the broker identifier to the `LSApplicationQueriesSchemes` section

MSAL uses `-canOpenURL:` to check whether the broker is installed on the device. In iOS 9, Apple locked down the schemes that an application can query for.

Add `msauthv2` to the `LSApplicationQueriesSchemes` section of the *Info.plist* file, as in the following example:

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

Step 7: Add a redirect URI to your app registration

When you use the broker, your redirect URI has an extra requirement. The redirect URI *must* have the following format:

```
$"msauth.{BundleId}://auth"
```

Here's an example:

```
public static string redirectUriOnIos = "msauth.com.yourcompany.XForms://auth";
```

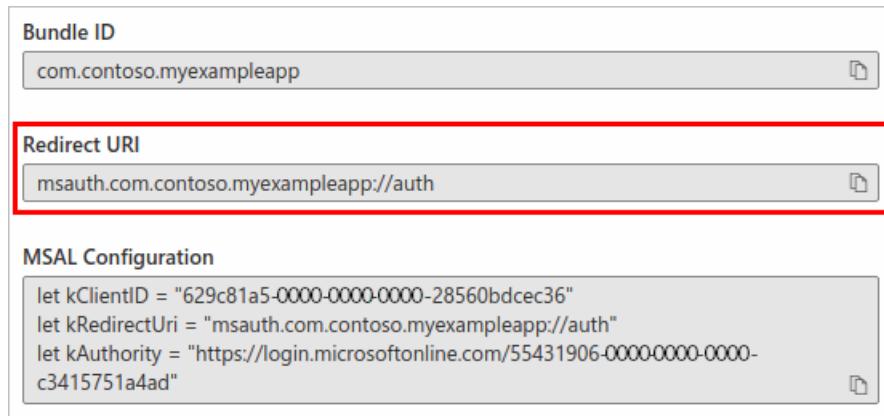
Notice that the redirect URI matches the `CFBundleURLSchemes` name that you included in the *Info.plist* file.

Add the redirect URI to the app's registration in the [Azure portal](#). To generate a properly formatted redirect URI, use **App registrations** in the Azure portal to generate the brokered redirect URI from the bundle ID.

To generate the redirect URI:

1. Sign in to the [Azure portal](#).
2. Select **Azure Active Directory > App registrations > your registered app**
3. Select **Authentication > Add a platform > iOS / macOS**
4. Enter your bundle ID, and then select **Configure**.

Copy the generated redirect URI that appears in the **Redirect URI** text box for inclusion in your code:



5. Select **Done** to complete generation of the redirect URI.

Brokered authentication for Android

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 creating the `IPublicClientApplication` through the `PublicClientApplicationBuilder`.

```
var app = PublicClientApplicationBuilder  
    .Create(ClientId)  
    .WithBroker()  
    .WithRedirectUri(redirectUriOnAndroid) // See step #4  
    .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 with the `OnActivityResult()` method. Since MSAL will wait for the response from the broker, your application needs to route the result to MSAL.NET.

Route the result to the

`SetAuthenticationContinuationEventArgs(int requestCode, Result resultCode, Intent data)` method by overriding the `OnActivityResult()` method as shown here:

```
protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)  
{  
    base.OnActivityResult(requestCode, resultCode, data);  
    AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(requestCode, resultCode, data);  
}
```

This method is invoked every time the broker application is launched, and is used as an opportunity to process the response from the broker and complete the authentication process started by MSAL.NET.

Step 3: Set an Activity

To enable brokered authentication, set an activity so that MSAL can send and receive the response to and from the broker. To do so, provide the activity (usually the `MainActivity`) to `WithParentActivityOrWindow(object parent)` the parent object.

For example, in the call to `AcquireTokenInteractive()`:

```
result = await app.AcquireTokenInteractive(scopes)
    .WithParentActivityOrWindow((Activity)context)
    .ExecuteAsync();
```

Step 4: Add a redirect URI to your app registration

MSAL uses URLs to invoke the broker and then return to your app. To complete that round trip, register a **Redirect URI** for your app by using the [Azure portal](#).

The format of the redirect URI for your application depends on the certificate used to sign the APK. For example:

```
msauth://com.microsoft.xforms.testApp/hgbUYHVBYUTvuvT&Y6tr554365466=
```

The last part of the URI, `hgbUYHVBYUTvuvT&Y6tr554365466=`, is the Base64-encoded version of the signature that the APK is signed with. While developing your app in Visual Studio, if you're debugging your code without signing the APK with a specific certificate, Visual Studio signs the APK for you for debugging purposes. When Visual Studio signs the APK for you in this way, it gives it a unique signature for the machine it's built on. Thus, each time you build your app on a different machine, you'll need to update the redirect URI in the application's code and the application's registration in the Azure portal in order to authenticate with MSAL.

While debugging, you may encounter an MSAL exception (or log message) stating the redirect URI provided is incorrect. **The exception or log message also indicates the redirect URI you should be using** with the current machine you're debugging on. You can use the provided redirect URI to continue developing your app as long as you update redirect URI in code and add the provided redirect URI to the app's registration in the Azure portal.

Once you're ready to finalize your code, update the redirect URI in the code and the application's registration in the Azure portal to use the signature of the certificate you sign the APK with.

In practice, this means you should consider adding a redirect URI for each member of your development team, *plus* a redirect URI for the production signed version of the APK.

You can compute the signature yourself, similar to how MSAL does it:

```

private string GetRedirectUriForBroker()
{
    string packageName = Application.Context.PackageName;
    string signatureDigest = this.GetCurrentSignatureForPackage(packageName);
    if (!string.IsNullOrEmpty(signatureDigest))
    {
        return string.Format(CultureInfo.InvariantCulture, "{0}://{1}/{2}", RedirectUriScheme,
            packageName.ToLowerInvariant(), signatureDigest);
    }

    return string.Empty;
}

private string GetCurrentSignatureForPackage(string packageName)
{
    PackageInfo info = Application.Context.PackageManager.GetPackageInfo(packageName,
        PackageInfoFlags.Signatures);
    if (info != null && info.Signatures != null && info.Signatures.Count > 0)
    {
        // First available signature. Applications can be signed with multiple signatures.
        // The order of Signatures is not guaranteed.
        Signature signature = info.Signatures[0];
        MessageDigest md = MessageDigest.GetInstance("SHA");
        md.Update(signature.ToByteArray());
        return Convert.ToBase64String(md.Digest()), Base64FormattingOptions.None);
        // Server side needs to register all other tags. ADAL will
        // send one of them.
    }
}

```

You also have the option of acquiring the signature for your package by using `keytool` with the following commands:

- Windows:

```
keytool.exe -list -v -keystore "%LocalAppData%\Xamarin\Mono for Android\debug.keystore" -alias androiddebugkey -storepass android -keypass android
```

- macOS:

```
keytool -exportcert -alias androiddebugkey -keystore ~/.android/debug.keystore | openssl sha1 -binary
| openssl base64
```

Step 5 (optional): Fall back to the system browser

If MSAL is configured to use the broker but the broker is not installed, MSAL will fall back to using a web view (a browser). MSAL will try to authenticate using the default system browser on the device, which fails because the redirect URI is configured for the broker and the system browser doesn't know how to use it to navigate back to MSAL. To avoid the failure, you can configure an *intent filter* with the broker redirect URI you used in step 4.

Modify your application's manifest to add the intent filter:

```

<!-- NOTE the SLASH (required) that prefixes the signature value in the path attribute.
     The signature value is the Base64-encoded signature discussed above. -->
<intent-filter>
    <data android:scheme="msauth"
          android:host="Package Name"
          android:path="/Package Signature"/>

```

For example, if you have a redirect URI of

`msauth://com.microsoft.xforms.testApp/hgbUYHVBUTvuvT&Y6tr554365466=`, your manifest should look like the following XML snippet.

The forward-slash (/) in front of the signature in the `android:path` value is required.

```
<!-- NOTE the SLASH (required) that prefixes the signature value in the path attribute.  
The signature value is the Base64-encoded signature discussed above. -->  
<intent-filter>  
    <data android:scheme="msauth"  
          android:host="com.microsoft.xforms.testApp"  
          android:path="/hgbUYHVBUTvuvT&Y6tr554365466="/>
```

For more information about configuring your application for system browser and Android 11 support, see [Update the Android manifest for system browser support](#).

As an alternative, you can configure MSAL to fall back to the embedded browser, which doesn't rely on a redirect URI:

```
.WithUseEmbeddedWebUi(true)
```

Troubleshooting tips for Android brokered authentication

Here are a few tips on avoiding issues when you implement brokered authentication on Android:

- **Redirect URI** - Add a redirect URI to your application registration in the [Azure portal](#). A missing or incorrect redirect URI is a common issue encountered by developers.
- **Broker version** - Install the minimum required version of the broker apps. Either of these two apps can be used for brokered authentication on Android.
 - [Intune Company Portal](#) (version 5.0.4689.0 or greater)
 - [Microsoft Authenticator](#) (version 6.2001.0140 or greater).
- **Broker precedence** - MSAL communicates with the *first broker installed* on the device when multiple brokers are installed.

Example: If you first install Microsoft Authenticator and then install Intune Company Portal, brokered authentication will *only* happen on the Microsoft Authenticator.

- **Logs** - If you encounter an issue with brokered authentication, viewing the broker's logs might help you diagnose the cause.
 - Get Microsoft Authenticator logs:
 1. Select the menu button in the top-right corner of the app.
 2. Select **Send Feedback > Having Trouble?**.
 3. Under **What are you trying to do?**, select an option and add a description.
 4. To send the logs, select the arrow in the top-right corner of the app.

After you send the logs, a dialog box displays the incident ID. Record the incident ID, and include it when you request assistance.

- Get Intune Company Portal logs:
 1. Select the menu button on the top-left corner of the app.
 2. Select **Help > Email Support**.
 3. To send the logs, select **Upload Logs Only**.

After you send the logs, a dialog box displays the incident ID. Record the incident ID, and include it

when you request assistance.

Next steps

Learn about [Considerations for using Universal Windows Platform with MSAL.NET](#).

Call a web API from a mobile app

4/12/2022 • 6 minutes to read • [Edit Online](#)

After your app signs in a user and receives tokens, the Microsoft Authentication Library (MSAL) exposes information about the user, the user's environment, and the issued tokens. Your app can use these values to call a web API or display a welcome message to the user.

In this article, we'll first look at the MSAL result. Then we'll look at how to use an access token from `AuthenticationResult` or `result` to call a protected web API.

MSAL result

MSAL provides the following values:

- `AccessToken` calls protected web APIs in an HTTP bearer request.
- `IdToken` contains useful information about the signed-in user. This information includes the user's name, the home tenant, and a unique identifier for storage.
- `ExpiresOn` is the expiration time of the token. MSAL handles an app's automatic refresh.
- `TenantId` is the identifier of the tenant where the user signed in. For guest users in Azure Active Directory (Azure AD) B2B, this value identifies the tenant where the user signed in. The value doesn't identify the user's home tenant.
- `Scopes` indicates 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` value. An `Account` value represents the current user's signed-in account:

- `HomeAccountId` identifies the user's home tenant.
- `UserName` is the user's preferred username. This value might be empty for Azure AD B2C users.
- `AccountIdentifier` identifies the signed-in user. In most cases, this value is the same as the `HomeAccountId` value unless the user is a guest in another tenant.

Call an API

After you have the access token, you can call a web API. Your app will use the token to build 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. You can use `accessToken` to call a web API. Add this property to the HTTP authorization header before you call to access the protected web API.

```

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];

```

```

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

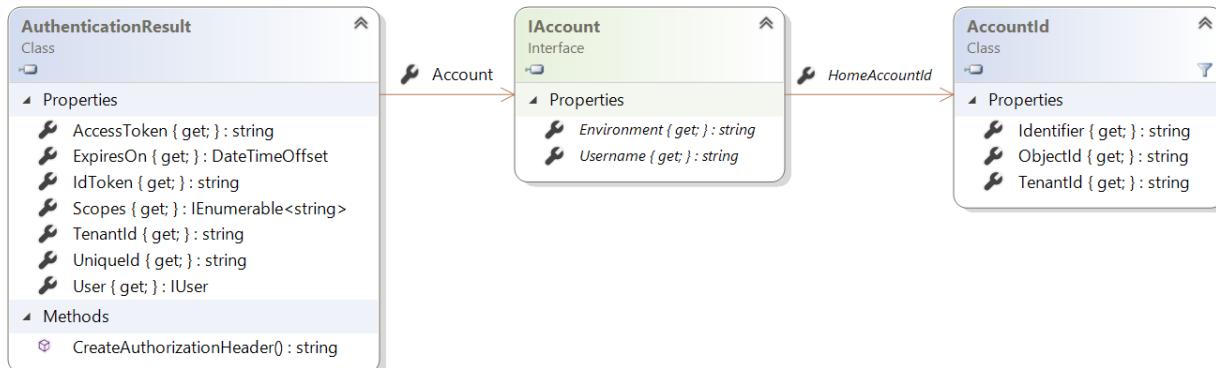
The methods to acquire tokens return `AuthenticationResult`. For async methods, `Task<AuthenticationResult>` returns.

In MSAL.NET, `AuthenticationResult` exposes:

- `AccessToken` for the web API to access resources. This parameter is a string, usually a Base-64-encoded JWT. 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. Writing code that depends on access token content on the client is one of the biggest sources of errors and client logic breaks. For more information, see [Access tokens](#).
- `IdToken` for the user. This parameter is an encoded JWT. For more information, see [ID tokens](#).
- `ExpiresOn` tells the date and time when the token expires.
- `TenantId` contains the tenant in which the user was found. For guest users in Azure Active Directory (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 an account through the `IAccount` interface. This breaking change provides the right semantics. The same user can have several accounts, in different Azure AD directories. Also, MSAL.NET provides better information in the case of guest scenarios because 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 with the properties shown in the following table.

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. They're not Base 64 encoded.

The `IAccount` interface represents information about a single account. The same user can be present in different tenants, which means that a user can have multiple accounts. Its members are shown in the following table.

PROPERTY	DESCRIPTION
<code>Username</code>	A string that contains the displayable value in UserPrincipalName (UPN) format, for example, john.doe@contoso.com. This string can be null, unlike HomeAccountId and HomeAccountId.Identifier, which 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 that contains 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. Here, the value is only the host.
<code>HomeAccountId</code>	The account ID of the home account for the user. This property uniquely identifies the user across Azure AD tenants.

Use the token to call a protected API

After `AuthenticationResult` is returned by MSAL in `result`, add it to the HTTP authorization header before you make the call to access the protected web API.

```
httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer",
result.AccessToken);

// Call the web API.
HttpResponseMessage response = await _httpClient.GetAsync(apiUri);
...
}
```

Make several API requests

To call the same API several times, or call multiple APIs, then consider the following subjects when you build your app:

- **Incremental consent:** The Microsoft identity platform allows apps to get user consent when 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 that it needs.
- **Conditional access:** When you make several API requests, in certain scenarios you might have to meet additional conditional-access requirements. Requirements can increase in this way if the first request has no conditional-access policies and your app attempts to silently access a new API that requires conditional access. To handle this problem, be sure to catch errors from silent requests, and be prepared to make an interactive request. For more information, see [Guidance for conditional access](#).

Call several APIs by using incremental consent and conditional access

To call several APIs for the same user, after you acquire a token for the 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();
```

Interaction is required when:

- The user consented for the first API but now needs to consent for more scopes. In this case, you use incremental consent.
- The first API doesn't require [multi-factor authentication](#), but the next API 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 on to the next article in this scenario, [Move to production](#).

Prepare mobile apps for production

4/12/2022 • 2 minutes to read • [Edit Online](#)

This article provides details about how to improve the quality and reliability of your mobile app before you move it to production.

Handle errors

As you prepare a mobile app for production, several error conditions can occur. The main cases you'll handle are silent failures and fallbacks to interaction. Other conditions that you should consider include no-network situations, service outages, requirements for admin consent, and other scenario-specific cases.

For each Microsoft Authentication Library (MSAL) type, you can find sample code and wiki content that describes how to handle error conditions:

- [MSAL Android wiki](#)
- [MSAL iOS wiki](#)
- [MSAL.NET wiki](#)

Enable logging

To help in debugging and authentication failure troubleshooting scenarios, the Microsoft Authentication Library provides built-in logging support. Logging for each library is covered in the following articles:

- [Logging in MSAL.NET](#)
- [Logging in MSAL for Android](#)
- [Logging in MSALjs](#)
- [Logging in MSAL for iOS/macOS](#)
- [Logging in MSAL for Java](#)
- [Logging in MSAL for Python](#)

Here are some suggestions for data collection:

- Users might ask for help when they have problems. A best practice is to capture and temporarily store logs. Provide a location where users can upload the logs. MSAL provides logging extensions to capture detailed information about authentication.
- If telemetry is available, enable it through MSAL to gather data about how users sign in to your app.

Validate your integration

Test your integration by following the [Microsoft identity platform integration checklist](#).

Build for resilience

Learn how to increase resiliency in your app. For details, see [Increase resilience of authentication and authorization applications you develop](#)

Next steps

To try out additional samples, [Mobile public client applications](#).

Migrate Android applications that use a broker from ADAL.NET to MSAL.NET

4/12/2022 • 2 minutes to read • [Edit Online](#)

If you have a Xamarin Android app currently using the Azure Active Directory Authentication Library for .NET (ADAL.NET) and an [authentication broker](#), it's time to migrate to the [Microsoft Authentication Library for .NET](#) (MSAL.NET).

Prerequisites

- A Xamarin Android app already integrated with a broker ([Microsoft Authenticator](#) or [Intune Company Portal](#)) and ADAL.NET that you need to migrate to MSAL.NET.

Step 1: Enable the broker

Current ADAL code:	MSAL counterpart:
In ADAL.NET, broker support is enabled on a per-authentication context basis. To call the broker, you had to set a <code>useBroker</code> to <code>true</code> in the <code>PlatformParameters</code> constructor: <pre>public PlatformParameters(Activity callerActivity, bool useBroker)</pre>	In MSAL.NET, broker support is enabled on a per-PublicClientApplication basis. Use the <code>WithBroker()</code> parameter (which is set to true by default) to call broker: <pre>var app = PublicClientApplicationBuilder .Create(ClientId) .WithBroker() .WithRedirectUri(redirectUriOnAndroid) .Build();</pre>
In the platform-specific page renderer code for Android, you set the <code>useBroker</code> flag to true: <pre>page.BrokerParameters = new PlatformParameters(this, true, PromptBehavior.SelectAccount);</pre>	Then, in the <code>AcquireToken</code> call: <pre>result = await app.AcquireTokenInteractive(scopes) .WithParentActivityOrWindow(App.RootViewControl ler) .ExecuteAsync();</pre>
Then, include the parameters in the acquire token call: <pre>AuthenticationResult result = await AuthContext.AcquireTokenAsync(Resource, ClientId, new Uri(RedirectURI), platformParameters) .ConfigureAwait(false);</pre>	

Step 2: Set an Activity

In ADAL.NET, you passed in an activity (usually the `MainActivity`) as part of the `PlatformParameters` as shown in [Step 1: Enable the broker](#).

MSAL.NET also uses an activity, but it's not required in regular Android usage without a broker. To use the broker,

set the activity to send and receive responses from broker.

Current ADAL code:	MSAL counterpart:
<p>The activity is passed into the PlatformParameters in the Android-specific platform.</p> <pre>page.BrokerParameters = new PlatformParameters(this, true, PromptBehavior.SelectAccount);</pre>	<p>In MSAL.NET, do two things to set the activity for Android:</p> <ol style="list-style-type: none">1. In <code>MainActivity.cs</code>, set the <code>App.RootViewController</code> to the <code>MainActivity</code> to ensure there's an activity with the call to the broker. If it's not set correctly, you may get this error: <code>"Activity_required_for_android_broker":"Activity is null, so MSAL.NET cannot invoke the Android broker. See https://aka.ms/Brokered-Authentication-for-Android"</code>2. On the AcquireTokenInteractive call, use the <code>.WithParentActivityOrWindow(App.RootViewController)</code> and pass in the reference to the activity you will use. This example will use the <code>MainActivity</code>. <p>For example:</p> <p>In <code>App.cs</code>:</p> <pre>public static object RootViewController { get; set; }</pre> <p>In <code>MainActivity.cs</code>:</p> <pre>LoadApplication(new App()); App.RootViewController = this;</pre> <p>In the AcquireToken call:</p> <pre>result = await app.AcquireTokenInteractive(scopes) .WithParentActivityOrWindow(App.RootViewController) .ExecuteAsync();</pre>

Next steps

For more information about Android-specific considerations when using MSAL.NET with Xamarin, see [Configuration requirements and troubleshooting tips for Xamarin Android with MSAL.NET](#).

Migrate iOS applications that use Microsoft Authenticator from ADAL.NET to MSAL.NET

4/12/2022 • 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.RootController)
.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>msauth</p> <pre><key>LSApplicationQueriesSchemes</key> <array> <string>msauth</string> </array></pre>	<p>Uses</p> <p>msauthv2</p> <pre><key>LSApplicationQueriesSchemes</key> <array> <string>msauthv2</string> <string>msauthv3</string> </array></pre>

Step 6: Register your redirect URI in the Azure 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 Azure portal.

Current ADAL code:	MSAL counterpart:
--------------------	-------------------

```
"<app-scheme>://<your.bundle.id>"
```

Example:

```
mytestiosapp://com.mycompany.myapp
```

```
 $"msauth.{BundleId}://auth"
```

Example:

```
public static string redirectUriOnIos =  
    "msauth.com.yourcompany.XForms://auth";
```

For more information about how to register the redirect URI in the Azure portal, see [Step 7: Add a redirect URI to your app registration](#).

Step 7: Set the Entitlements.plist

Enable keychain access in the *Entitlements.plist* file:

```
<key>keychain-access-groups</key>  
  <array>  
    <string>$(@{AppIdentifierPrefix})com.microsoft.adalcache</string>  
  </array>
```

For more information about enabling keychain access, see [Enable keychain access](#).

Next steps

Learn about [Xamarin iOS-specific considerations with MSAL.NET](#).

Android Microsoft Authentication Library configuration file

4/12/2022 • 7 minutes to read • [Edit Online](#)

The Android Microsoft Authentication Library (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>broker_redirect_uri_registered</code>	Boolean	No	Possible values: <code>true</code> , <code>false</code>
<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.

`broker_redirect_uri_registered`

If you want to use brokered authentication, the `broker_redirect_uri_registered` property must be set to `true`. In a brokered authentication scenario, if the application isn't in the correct format to talk to the broker as described in [Redirect URI for public client apps](#), the application validates your redirect URI and throws an exception when it starts.

`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
call
},
// Example AzureAD My Organization
{
    "type": "AAD",
    "audience": {
        "type": "AzureADMyOrg",
        "tenant_id": "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	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.
AAD	AzureADMultipleOrgs			https://login.microsoftonline.com/organizations	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	accounts can use this endpoint.

Type	Audience	Tenant ID	Authority URL	Resulting Endpoint	Notes
B2C			See Resulting Endpoint	https://login.microsoftonline.com/tfp/contoso.onmicrosoft.com	Only applies to B2C policies. The authority URL must be 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. Brokered authentication does not work for Azure AD B2C.

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 type= <code>AAD</code> . Specifies the identity your app targets. Use the value from your app registration
<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>	string	No	Which log messages to output. Supported log levels include <code>ERROR</code> , <code>WARNING</code> , <code>INFO</code> , and <code>VERBOSE</code> .
<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"
      }
    }
  ]
}
```



```

    ],
    "browser_use_customTab" : false
),

{
    "browser_package_name": "com.microsoft.emmx",
    "browser_signature_hashes": [
        "Ivy-Rk6ztai_IudfbYUrSHugzRqAtHwslFvHT0PTvLMsEKLUIgv7ZzbVxygWy_M5mOPpfjZrd3v0x3t-cA6fVQ=="
    ],
    "browser_use_customTab" : false
),

{
    "browser_package_name": "com.opera.browser",
    "browser_signature_hashes": [
        "FIJ3IIeqB7V0qHpRNepYNkhEGA_eJaf7ntca-Oa_6Feev3UkgngpuTnv31JdAmpEFPGNPo0RHqd1U0k-3jWJWw=="
    ],
    "browser_use_customTab" : false
),

{
    "browser_package_name": "com.opera.mini.native",
    "browser_signature_hashes": [
        "TOTyHs086iGIEdxrX_24aAewTzx7Wbi6niS2ZrpPhLkjuzPAh1c3NQ_U4Lx1KdgihQE4BiS36MifP6LbmmUYQ=="
    ],
    "browser_use_customTab" : false
),

{
    "browser_package_name": "mobi.mgeek.TunnyBrowser",
    "browser_signature_hashes": [
        "RMVoXuK1sfJZuGZ8onG1yhMc-sKiAV2NiB_GZfdNlN8XJ78XEE2wPM6LnQiyltF25GkHiPN2iKQ1Gwa02bkyyQ=="
    ],
    "browser_use_customTab" : false
),

{
    "browser_package_name": "org.mozilla.focus",
    "browser_signature_hashes": [
        "L72dT-stFqomSY7sYySrgBJ3VYKbipMzapmUXFTZNqOzN_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%2Bh11ZifsqwyKrY%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);

```

Shared device mode for Android devices

4/12/2022 • 4 minutes to read • [Edit Online](#)

Frontline workers such as retail associates, flight crew members, and field service workers often use a shared mobile device to do their work. That becomes problematic when they start sharing passwords or pin numbers to access customer and business data on the shared device.

Shared device mode allows you to configure an Android device so that it can be easily shared by multiple employees. Employees can sign in and access customer information quickly. When they're finished with their shift or task, they can sign out of the device and it will be immediately ready for the next employee to use.

Shared device mode also provides Microsoft identity backed management of the device.

To create a shared device mode app, developers and cloud device admins work together:

- Developers write a single-account app (multiple-account apps aren't supported in shared device mode), add `"shared_device_mode_supported": true` to the app's configuration, and write code to handle things like shared device sign-out.
- Device admins prepare the device to be shared by installing the authenticator app, and setting the device to shared mode using the authenticator app. Only users who are in the [Cloud Device Administrator](#) role can put a device into shared mode by using the [Authenticator app](#). You can configure the membership of your organizational roles in the Azure portal via: [Azure Active Directory > Roles and Administrators > Cloud Device Administrator](#).

This article focuses primarily what developers should think about.

Single vs multiple-account applications

Applications written using the Microsoft Authentication Library (MSAL) SDK can manage a single account or multiple accounts. For details, see [single-account mode or multiple-account mode](#).

The Microsoft identity platform features available to your app vary depending on whether the application is running in single-account mode or multiple-account mode.

Shared device mode apps only work in single-account mode.

IMPORTANT

Applications that only support multiple-account mode can't run on a shared device. If an employee loads an app that doesn't support single-account mode, it won't run on the shared device.

Apps written before the MSAL SDK was released run in multiple-account mode and must be updated to support single-account mode before they can run on a shared mode device.

Supporting both single-account and multiple-accounts

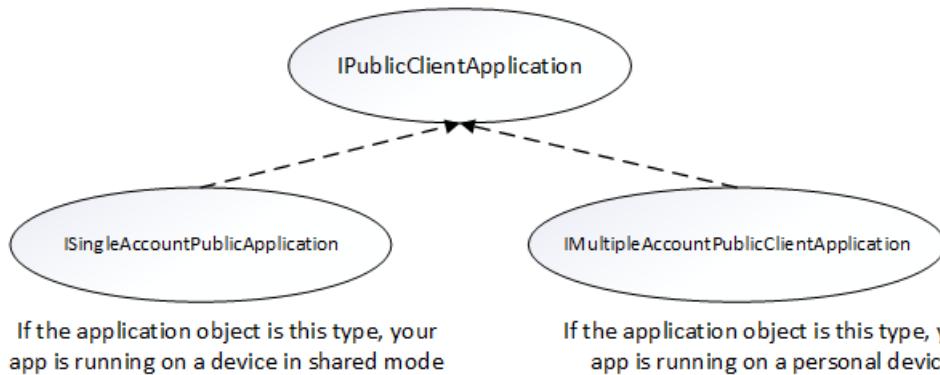
Your app can be built to support running on both personal devices and shared devices. If your app currently supports multiple accounts and you want to support shared device mode, add support for single account mode.

You may also want your app to change its behavior depending on the type of device it's running on. Use `ISingleAccountPublicClientApplication.isSharedDevice()` to determine when to run in single-account mode.

There are two different interfaces that represent the type of device your application is on. When you request an

application instance from MSAL's application factory, the correct application object is provided automatically.

The following object model illustrates the type of object you may receive and what it means in the context of a shared device:



You'll need to do a type check and cast to the appropriate interface when you get your `IPublicClientApplication` object. The following code checks for multiple account mode or single account mode, and casts the application object appropriately:

```
private IPublicClientApplication mApplication;

    // Running in personal-device mode?
    if (mApplication instanceof IMultipleAccountPublicClientApplication) {
        IMultipleAccountPublicClientApplication multipleAccountApplication =
            (IMultipleAccountPublicClientApplication) mApplication;
        ...
    }
    // Running in shared-device mode?
    } else if (mApplication instanceof ISingleAccountPublicClientApplication) {
        ISingleAccountPublicClientApplication singleAccountApplication =
            (ISingleAccountPublicClientApplication) mApplication;
        ...
    }
}
```

The following differences apply depending on whether your app is running on a shared or personal device:

	SHARED MODE DEVICE	PERSONAL DEVICE
Accounts	Single account	Multiple accounts
Sign-in	Global	Global
Sign-out	Global	Each application can control if the sign-out is local to the app or for the family of applications.
Supported account types	Work accounts only	Personal and work accounts supported

Why you may want to only support single-account mode

If you're writing an app that will only be used for frontline workers using a shared device, we recommend you write your application to only support single-account mode. This includes most applications that are task focused such as medical records apps, invoice apps, and most line-of-business apps. Only supporting single-account mode simplifies development because you won't need to implement the additional features that are part of multiple-account apps.

What happens when the device mode changes

If your application is running in multiple-account mode, and an administrator puts the device in shared device mode, all of the accounts on the device are cleared from the application and the application transitions to single-account mode.

Microsoft applications that support shared device mode

These Microsoft applications support Azure AD's shared device mode:

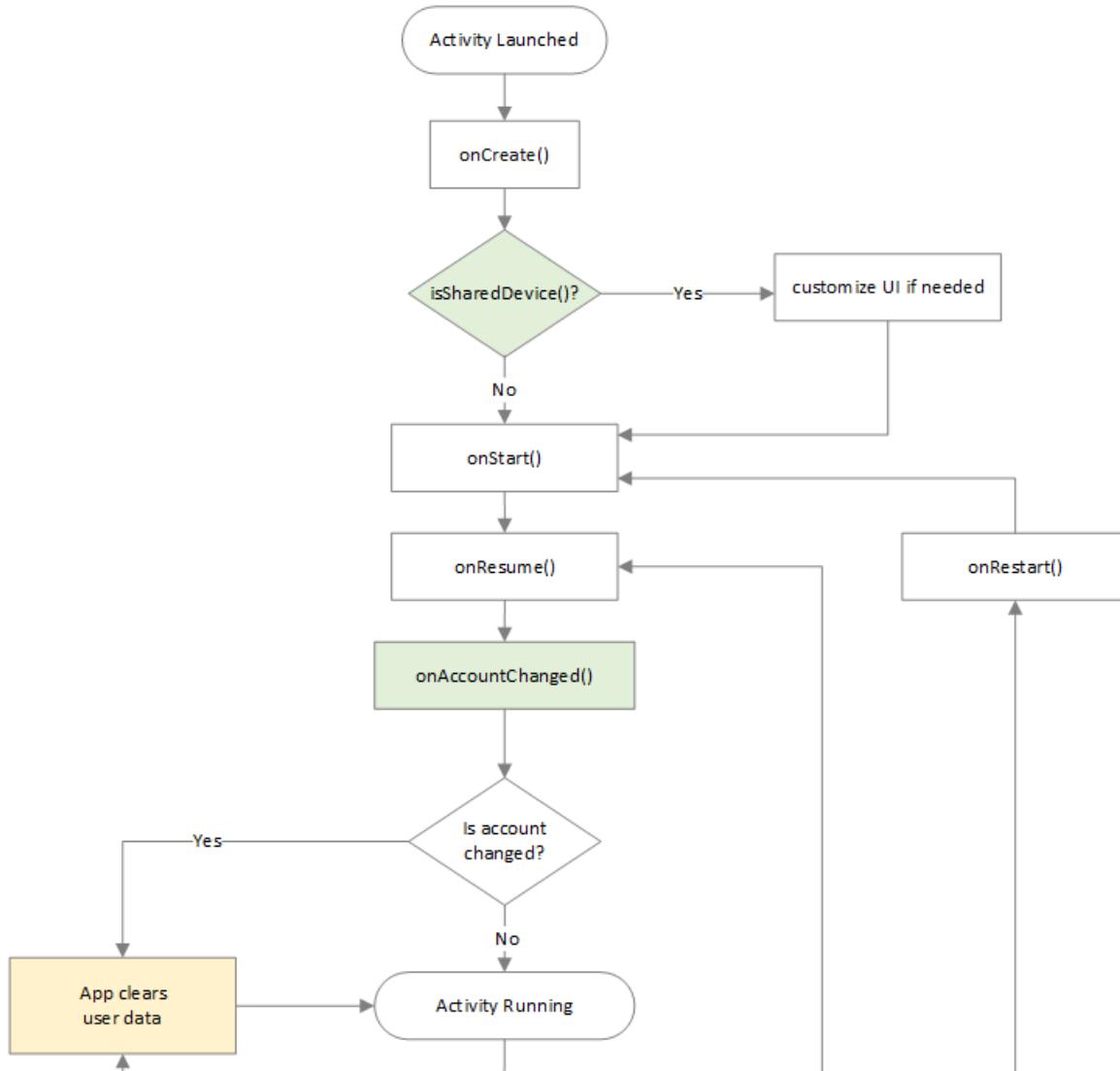
- Microsoft Teams
 - Microsoft Managed Home Screen app for Android Enterprise

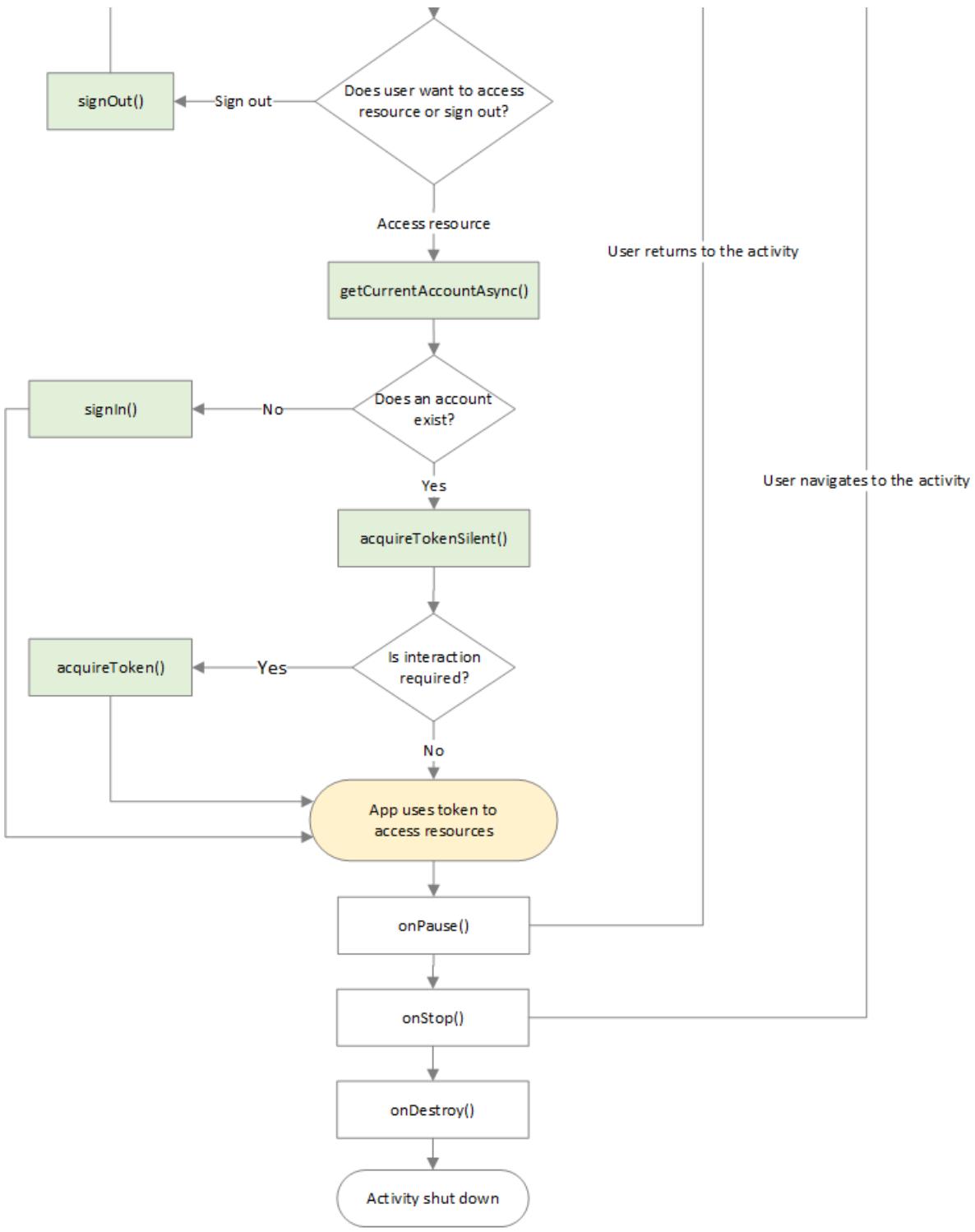
Shared device sign-out and the overall app lifecycle

When a user signs out, you'll need to take action to protect the privacy and data of the user. For example, if you're building a medical records app you'll want to make sure that when the user signs out previously displayed patient records are cleared. Your application must be prepared for data privacy and check every time it enters the foreground.

When your app uses MSAL to sign out the user in an app running on device that is in shared mode, the signed-in account and cached tokens are removed from both the app and the device.

The following diagram shows the overall app lifecycle and common events that may occur while your app runs. The diagram covers from the time an activity launches, signing in and signing out an account, and how events such as pausing, resuming, and stopping the activity fit in.





Next steps

For more information on how to run a frontline worker app on a shared mode on Android device, see:

- [Use shared-device mode in your Android application](#)

Use MSAL for Android with B2C

4/12/2022 • 5 minutes to read • [Edit Online](#)

The 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.

Choosing a compatible authorization_user_agent

The B2C identity management system supports authentication with a number of social account providers such as Google, Facebook, Twitter, and Amazon. If you plan to support such account types in your app, it is recommended that you configure your MSAL public client application to use either the `DEFAULT` or `BROWSER` value when specifying your manifest's `authorization_user_agent` due to restrictions prohibiting use of WebView-based authentication with some external identity providers.

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`.

Note: The `account_mode` must be set to **MULTIPLE** for B2C applications. Refer to the documentation for more information about [multiple account public client apps](#).

`app/src/main/res/raw/msal_config.json`

```
{
  "client_id": "<your_client_id_here>",
  "redirect_uri": "<your_redirect_uri_here>",
  "account_mode" : "MULTIPLE",
  "authorization_user_agent" : "DEFAULT",
  "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.

```
IMultipleAccountPublicClientApplication 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 unsuccessful, 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?](#)

Enable cross-app SSO on Android using MSAL

4/12/2022 • 9 minutes to read • [Edit Online](#)

Single sign-on (SSO) allows users to only enter their credentials once and have those credentials automatically work across applications.

The [Microsoft identity platform](#) and the Microsoft Authentication Library (MSAL) help you enable SSO across your own suite of apps. With the broker capability and Authenticator applications, you can extend SSO across the entire device.

In this how-to, you'll learn how to configure the SDKs used by your application to provide SSO to your customers.

Prerequisites

This how-to assumes you know how to:

- Provision your app using the Azure portal. For more information on this topic, see the instructions for creating an app in [the Android tutorial](#)
- Integrate your application with the [Microsoft Authentication Library for Android](#).

Methods for single sign-on

There are two ways for applications using MSAL for Android to achieve SSO:

- Through a [broker application](#)
- Through the [system browser](#)

It is recommended to use a broker application for benefits like device-wide SSO, account management, and conditional access. However, it requires your users to download additional applications.

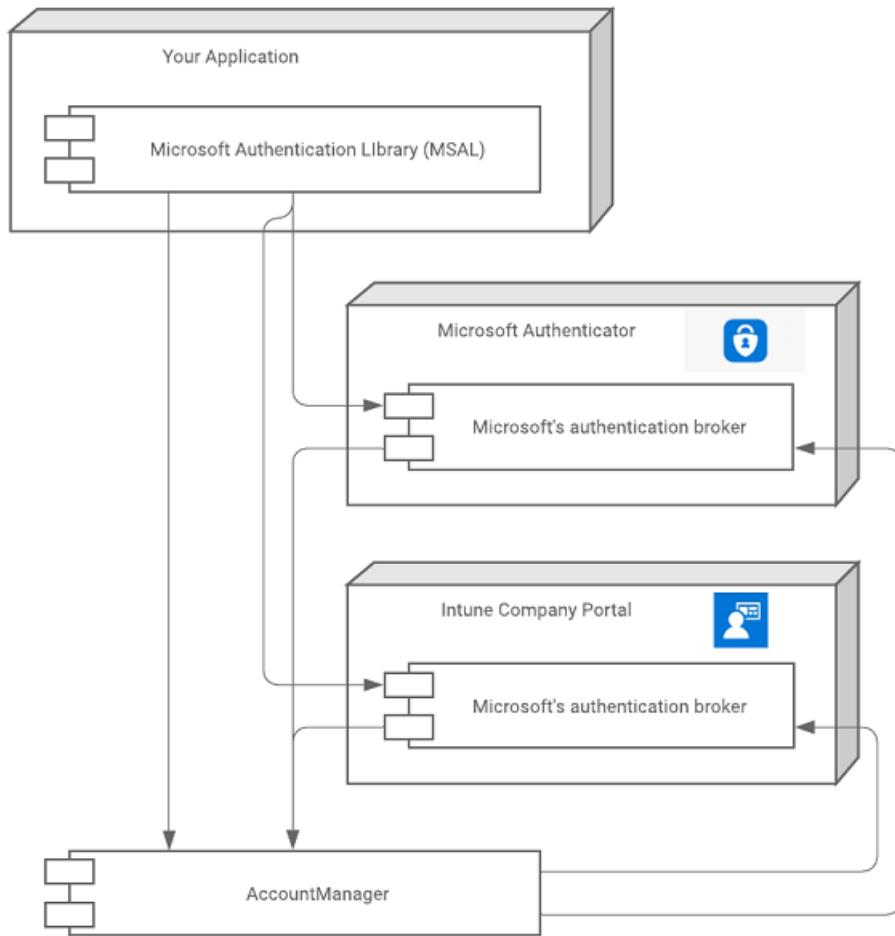
SSO through brokered authentication

We recommend that you 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 in the [Microsoft Authenticator](#) and [Intune Company Portal](#) apps.

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 doesn't 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 doesn't 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` can be thrown for several reasons, and needs to be resolved interactively. For example:

- 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.

Multiple brokers - If multiple brokers are installed on a device, the broker that was installed first is always the active broker. Only a single broker can be active on a device.

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

Generate a redirect URI for a broker

You must register a redirect URI that is compatible with the broker. The redirect URI for the broker should include your app's package name and the Base64-encoded representation of your app's signature.

The format of the redirect URI is: `msauth://<yourpackagename>/<base64urlencodedsignature>`

You can use [keytool](#) to generate a Base64-encoded signature hash using your app's signing keys, and then use the Azure portal to generate your redirect URI using that hash.

Linux and 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
```

Once you've generated a signature hash with *keytool*, use the Azure portal to generate the redirect URI:

1. Sign in to the [Azure portal](#) and select your Android app in **App registrations**.
2. Select **Authentication > Add a platform > Android**.
3. In the **Configure your Android app** pane that opens, enter the **Signature hash** that you generated earlier and a **Package name**.
4. Select the **Configure** button.

The Azure portal generates the redirect URI for you and displays it in the **Android configuration** pane's **Redirect URI** field.

For more information about signing your app, see [Sign your app](#) in the Android Studio User Guide.

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 settings in your MSAL configuration file:

```
"redirect_uri" : "<yourbrokerredirecturi>",
"broker_redirect_uri_registered": true
```

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 so 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

Verify broker integration

It might not be immediately clear that broker integration is working, but you can use the following steps to check:

1. On your Android device, complete a request using the broker.
2. In the settings on your Android device, look for a newly created account corresponding to the account that you authenticated with. The account should be of type *Work account*.

You can remove the account from settings if you want to repeat the test.

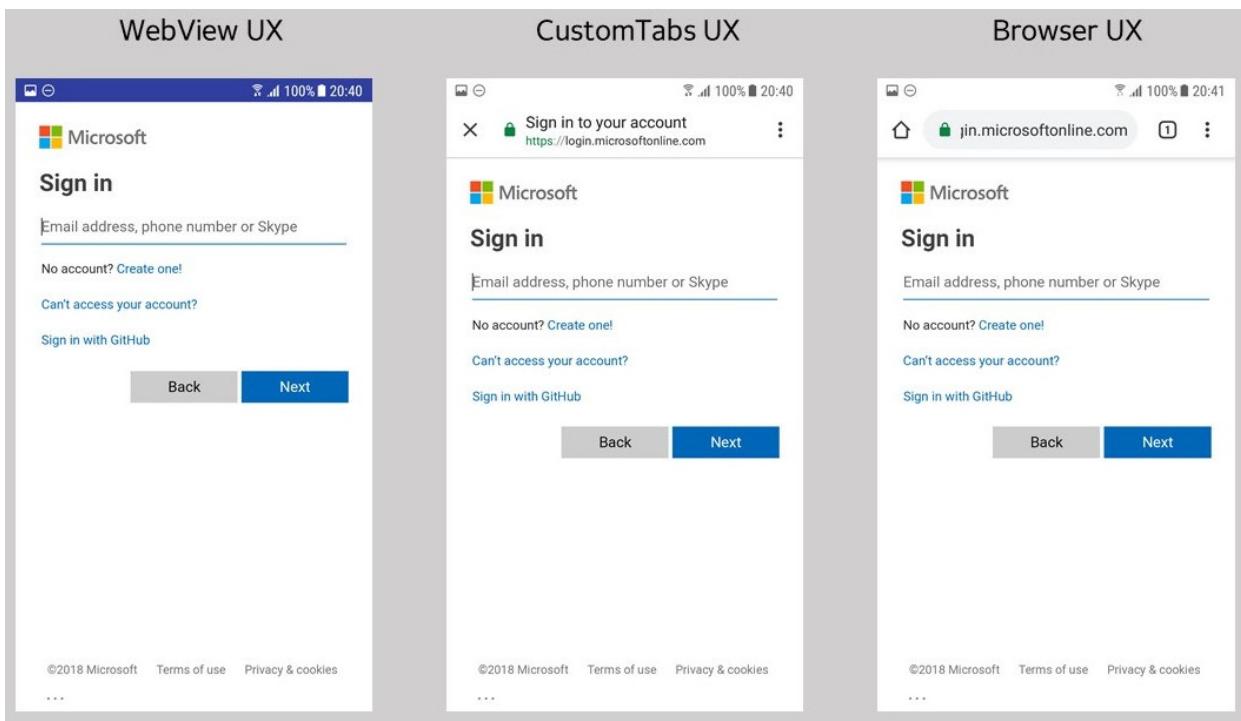
SSO through system browser

Android applications have the option to use the WebView, system browser, or Chrome Custom Tabs for authentication user experience. If the application is not using brokered authentication, it will need to use the system browser rather than the native webview in order to achieve SSO.

Authorization agents

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-on 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 experience across the device or between native apps and web apps.

If the application uses MSAL with a broker like Microsoft Authenticator or Intune Company Portal, then users can have a SSO experience across applications if they have 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 primarily retrieves the default browser from the package manager and checks if it is in a tested list of safe browsers. If not, MSAL falls back on using the Webview rather than launching another non-default browser from the safe list. The default browser 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`. If the device default setting isn't changed, the same browser should be launched for each sign in to ensure a SSO experience.

Tested Browsers

The following browsers have been tested to see if they correctly redirect to the `"redirect_uri"` specified in the configuration file:

DEVICE	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 ¹	pass	pass	pass	fail	pass
Huawei (API 26)	pass ²	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 ³	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

¹Samsung's built-in browser is Samsung Internet.

²Huawei's built-in browser is Huawei Browser.

³The default browser can't be changed inside the Oppo device setting.

Next steps

[Shared device mode for Android devices](#) allows you to configure an Android device so that it can be easily shared by multiple employees.

Accounts & tenant profiles (Android)

4/12/2022 • 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.

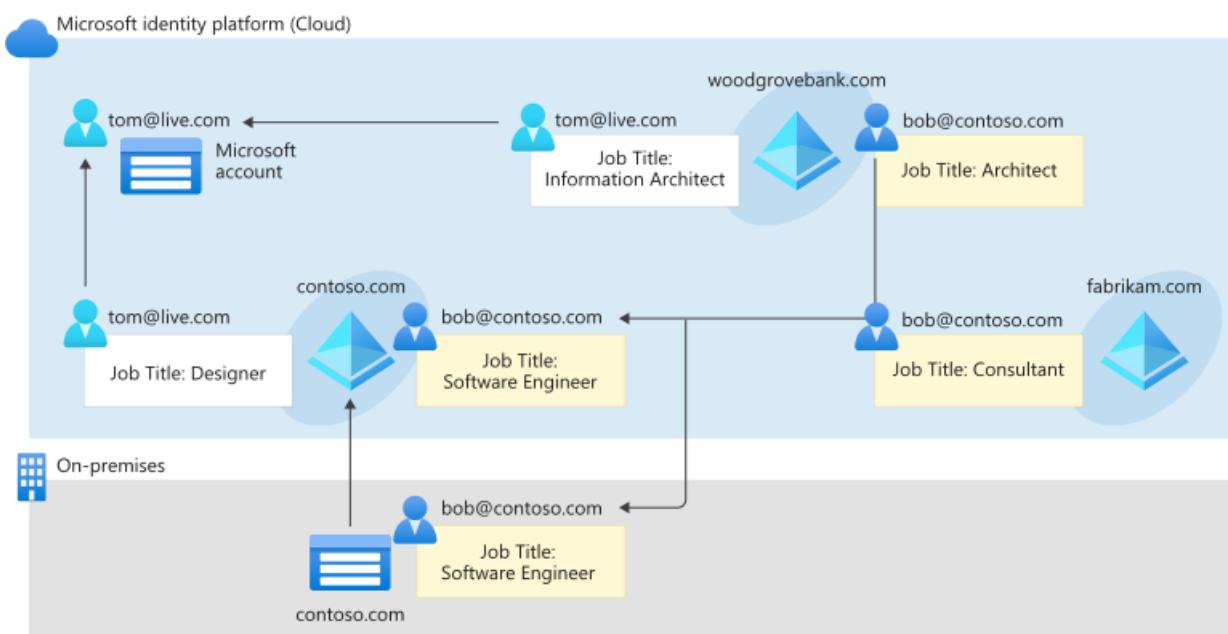
An account in the Microsoft identity platform consists of:

- 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;
}
```

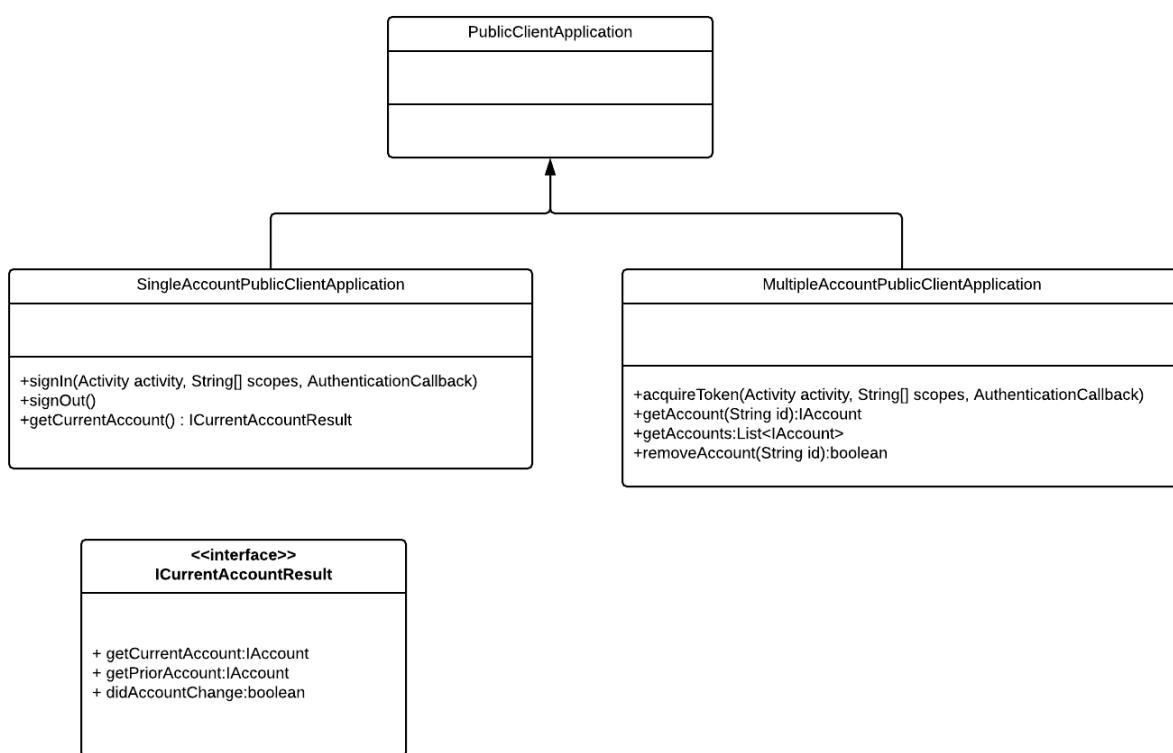
Single and multiple account public client apps

4/12/2022 • 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 the 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)
    {
    }
});
});
```

Logging in MSAL for Android

4/12/2022 • 2 minutes to read • [Edit Online](#)

The 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. Used 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.

The following sections provide more details about MSAL error logging for your application.

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);
```

Next steps

For more code samples, refer to [Microsoft identity platform code samples](#).

Handle exceptions and errors in MSAL for Android

4/12/2022 • 4 minutes to read • [Edit Online](#)

Exceptions in the Microsoft Authentication Library (MSAL) are intended to help app developers troubleshoot their application. 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 the sign-in experience, you may encounter errors about consents, Conditional Access (MFA, Device Management, Location-based restrictions), token issuance and redemption, and user properties.

ERROR CLASS	CAUSE/ERROR STRING	HOW TO HANDLE
<code>MsalUiRequiredException</code>	<ul style="list-style-type: none"><code>INVALID_GRANT</code> : The refresh token used to redeem access token is invalid, expired, or revoked. This exception may be because of a password change.<code>NO_TOKENS_FOUND</code> : Access token doesn't exist and no refresh token can be found to redeem access token.Step-up required<ul style="list-style-type: none">MFAMissing claimsBlocked by Conditional Access (for example, authentication broker installation required)<code>NO_ACCOUNT_FOUND</code> : No account available in the cache for silent authentication.	Call <code>acquireToken()</code> to prompt the user to enter their username and password, and possibly consent and perform multi factor authentication.
<code>MsalDeclinedScopeException</code>	<ul style="list-style-type: none"><code>DECLINED_SCOPE</code> : User or server has not accepted all scopes. The server may decline a scope if the requested scope is not supported, not recognized, or not supported for a particular account.	The developer should decide whether to continue authentication with the granted scopes or end the authentication process. Option to resubmit the acquire token request only for the granted scopes and provide hints for which permissions have been granted by passing <code>silentParametersForGrantedScopes</code> and calling <code>acquireTokenSilent</code> .

ERROR CLASS	CAUSE/ERROR STRING	HOW TO HANDLE
<code>MsalServiceException</code>	<ul style="list-style-type: none"> • <code>INVALID_REQUEST</code> : This request is missing a required parameter, includes an invalid parameter, includes a parameter more than once, or is otherwise malformed. • <code>SERVICE_NOT_AVAILABLE</code> : Represents 500/503/506 error codes due to the service being down. • <code>UNAUTHORIZED_REQUEST</code> : The client is not authorized to request an authorization code. • <code>ACCESS_DENIED</code> : The resource owner or authorization server denied the request. • <code>INVALID_INSTANCE</code> : <code>AuthorityMetadata</code> validation failed • <code>UNKNOWN_ERROR</code> : Request to server failed, but no error and <code>error_description</code> are returned back from the service. 	<p>This exception class represents errors when communicating to the service, can be from the authorize or token endpoints. MSAL reads the error and <code>error_description</code> from the server response. Generally, these errors are resolved by fixing app configurations either in code or in the app registration portal. Rarely a service outage can trigger this warning, which can only be mitigated by waiting for the service to recover.</p>
<code>MsalClientException</code>	<ul style="list-style-type: none"> • <code>MULTIPLE_MATCHING_TOKENS_DETECTED</code> : There are multiple cache entries found and the sdk cannot identify the correct access or refresh token from the cache. This exception usually indicates a bug in the sdk for storing tokens or that the authority is not provided in the silent request and multiple matching tokens are found. • <code>DEVICE_NETWORK_NOT_AVAILABLE</code> : No active network is available on the device. • <code>JSON_PARSE_FAILURE</code> : The sdk failed to parse the JSON format. • <code>IO_ERROR</code> : <code>IOException</code> happened, could be a device or network error. • <code>MALFORMED_URL</code> : The URL is malformed. Likely caused when constructing the auth request, authority, or redirect URI. • <code>UNSUPPORTED_ENCODING</code> : The encoding is not supported by the device. • <code>NO_SUCH_ALGORITHM</code> : The algorithm used to generate <code>PKCE</code> challenge is not supported. • <code>INVALID_JWT</code> : <code>JWT</code> returned 	<p>This exception class represents general errors that are local to the library. These exceptions can be handled by correcting the request.</p>

ERROR CLASS	CAUSE/ERROR STRING	HOW TO HANDLE
	<p>by the server is not valid or is empty or malformed.</p> <ul style="list-style-type: none"> ● <code>STATE_MISMATCH</code> : State from authorization response did not match the state in the authorization request. For authorization requests, the sdk will verify the state returned from redirect and the one sent in the request. ● <code>UNSUPPORTED_URL</code> : Unsupported URL, cannot perform ADFS authority validation. ● <code>AUTHORITY_VALIDATION_NOT_SUPPORTED</code> : The authority is not supported for authority validation. The sdk supports B2C authorities, but doesn't support B2C authority validation. Only well-known host will be supported. ● <code>CHROME_NOT_INSTALLED</code> : Chrome is not installed on the device. The sdk uses chrome custom tab for authorization requests if available, and will fall back to chrome browser. ● <code>USER_MISMATCH</code> : The user provided in the acquire token request doesn't match the user returned from server. 	
<code>MsalUserCancelException</code>	<ul style="list-style-type: none"> ● <code>USER_CANCELED</code> : The user initiated interactive flow and prior to receiving tokens back they canceled the request. 	
<code>MsalArgumentException</code>	<ul style="list-style-type: none"> ● <code>ILLEGAL_ARGUMENT_ERROR_CODE</code> ● <code>AUTHORITY_REQUIRED_FOR_SILENT</code> : Authority must be specified for <code>acquireTokenSilent</code> . 	<p>These errors can be mitigated by the developer correcting arguments and ensuring activity for interactive auth, completion callback, scopes, and an account with a valid ID have been provided.</p>

Catching errors

The following code snippet shows an example of catching errors in the case of silent `acquireToken` calls.

```
/**  
 * Callback used in for silent acquireToken calls.  
 */  
private SilentAuthenticationCallback getAuthSilentCallback() {  
    return new SilentAuthenticationCallback() {  
  
        @Override  
        public void onSuccess(IAuthenticationResult authenticationResult) {  
            Log.d(TAG, "Successfully authenticated");  
  
            /* Successfully got a token, use it to call a protected resource - MSGraph */  
            callGraphAPI(authenticationResult);  
        }  
  
        @Override  
        public void onError(MsalException exception) {  
            /* Failed to acquireToken */  
            Log.d(TAG, "Authentication failed: " + exception.toString());  
            displayError(exception);  
  
            if (exception instanceof MsalClientException) {  
                /* Exception inside MSAL, more info inside MsalError.java */  
            } 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 */  
            }  
        }  
    };  
}
```

Next steps

Learn more about [Logging in MSAL for Android](#).

ADAL to MSAL migration guide for Android

4/12/2022 • 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
- (Azure AD 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 Azure AD 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

Use incremental consent to provide additional context to your users 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.

EXCEPTION	DESCRIPTION
<code>MsalArgumentException</code>	Thrown if one or more inputs arguments are invalid.
<code>MsalClientException</code>	Thrown if the error is client side.

EXCEPTION	DESCRIPTION
MsalDeclinedScopeException	Thrown if one or more requested scopes were declined by the server.
MsalException	Default checked exception thrown by MSAL.
MsalIntuneAppProtectionPolicyRequiredException	Thrown if the resource has MAMCA protection policy enabled.
MsalServiceException	Thrown if the error is server side.
MsalUiRequiredException	Thrown if the token can't be refreshed silently.
MsalUserCancelException	Thrown if the user canceled the authentication flow.

ADALError to MsalException translation

IF YOU'RE CATCHING THESE ERRORS IN ADAL...	...CATCH THESE MSAL EXCEPTIONS:
No equivalent ADALError	MsalArgumentException
<ul style="list-style-type: none"> • ADALError.ANDROIDKEYSTORE_FAILED • ADALError.AUTH_FAILED_USER_MISMATCH • ADALError.DECRYPTION_FAILED • ADALError.DEVELOPER_AUTHORITY_CAN_NOT_BE_VALIDATED • ADALError.DEVELOPER_AUTHORITY_IS_NOT_VALID_INSTANCE • ADALError.DEVELOPER_AUTHORITY_IS_NOT_VALID_URL • ADALError.DEVICE_CONNECTION_IS_NOT_AVAILABLE • ADALError.DEVICE_NO SUCH_ALGORITHM • ADALError.ENCODING_IS_NOT_SUPPORTED • ADALError.ENCRYPTION_ERROR • ADALError.IO_EXCEPTION • ADALError.JSON_PARSE_ERROR • ADALError.NO_NETWORK_CONNECTION_POWER_OPTIMIZATION • ADALError.SOCKET_TIMEOUT_EXCEPTION 	MsalClientException
No equivalent ADALError	MsalDeclinedScopeException
<ul style="list-style-type: none"> • ADALError.APP_PACKAGE_NAME_NOT_FOUND • ADALError.BROKER_APP_VERIFICATION_FAILED • ADALError.PACKAGE_NAME_NOT_FOUND 	MsalException
No equivalent ADALError	MsalIntuneAppProtectionPolicyRequiredException
<ul style="list-style-type: none"> • ADALError.SERVER_ERROR • ADALError.SERVER_INVALID_REQUEST 	MsalServiceException

IF YOU'RE CATCHING THESE ERRORS IN ADAL...

-

`ADALError.AUTH_REFRESH_FAILED_PROMPT_NOT_ALLOWED`

...CATCH THESE MSAL EXCEPTIONS:

`MsalUiRequiredException`

No equivalent ADALError

`MsalUserCancelException`

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');
    }
});

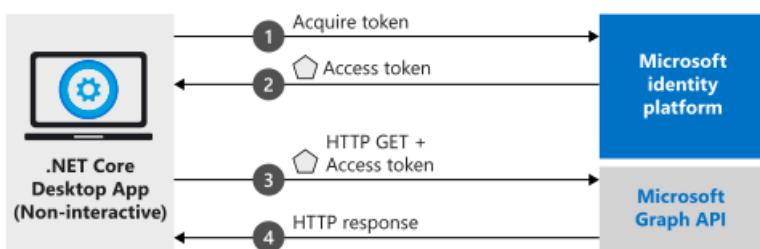
// New Log Levels:
public enum LogLevel
{
    /**
     * Error level logging.
     */
    ERROR,
    /**
     * Warning level logging.
     */
    WARNING,
    /**
     * Info level logging.
     */
    INFO,
    /**
     * Verbose level logging.
     */
    VERBOSE
}
```


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

4/12/2022 • 22 minutes to read • [Edit Online](#)

In this quickstart, you download and run a code sample that demonstrates how a .NET Core console application can get an access token to call the Microsoft Graph API and display a [list of users](#) in the directory. The code sample also demonstrates how a job or a Windows service can run with an application identity, instead of a user's identity. The sample console application in this quickstart is also a daemon application, so it's a confidential client application.

The following diagram shows how the sample app works:



Prerequisites

This quickstart requires [.NET Core 3.1 SDK](#) but will also work with .NET 5.0 SDK.

Register and download the app

You have two options to start building your application: automatic or manual configuration.

Automatic configuration

If you want to register and automatically configure your app and then download the code sample, follow these steps:

1. Go to the [Azure portal page for app registration](#).
2. Enter a name for your application and select **Register**.
3. Follow the instructions to download and automatically configure your new application in one click.

Manual configuration

If you want to manually configure your application and code sample, use the following procedures.

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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. For **Name**, enter a name for your application. For example, enter **Daemon-console**. Users of your app will see this name, and you can change it later.
6. Select **Register** to create the application.

7. Under **Manage**, select **Certificates & secrets**.
8. Under **Client secrets**, select **New client secret**, enter a name, and then select **Add**. Record the secret value in a safe location for use in a later step.
9. Under **Manage**, select **API Permissions > Add a permission**. Select **Microsoft Graph**.
10. Select **Application permissions**.
11. Under the **User** node, select **User.Read.All**, and then select **Add permissions**.

Step 2: Download your Visual Studio project

[Download the Visual Studio project](#)

You can run the provided project in either Visual Studio or Visual Studio for Mac.

TIP

To avoid errors caused by path length limitations in Windows, we recommend extracting the archive or cloning the repository into a directory near the root of your drive.

Step 3: Configure your Visual Studio project

1. Extract the .zip file to a local folder that's close to the root of the disk. For example, extract to *C:\Azure-Samples*.

We recommend extracting the archive into a directory near the root of your drive to avoid errors caused by path length limitations on Windows.

2. Open the solution in Visual Studio: *1-Call-MSGraph\daemon-console.sln* (optional).

3. In *appsettings.json*, replace the values of `Tenant`, `ClientId`, and `ClientSecret`:

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

In that code:

- `Enter_the_Application_Id_Here` is the application (client) ID for the application that you registered. To find the values for the application (client) ID and the directory (tenant) ID, go to the app's **Overview** page in the Azure portal.
- Replace `Enter_the_Tenant_Id_Here` with the tenant ID or tenant name (for example, `contoso.microsoft.com`).
- Replace `Enter_the_Client_Secret_Here` with the client secret that you created in step 1. To generate a new key, go to the **Certificates & secrets** page.

Step 4: Admin consent

If you try to run the application at this point, you'll receive an *HTTP 403 - Forbidden* error: "Insufficient privileges to complete the operation." This error happens because any app-only permission requires a global administrator of your directory to give consent to your application. Select one of the following options, depending on your role.

Global tenant administrator

If you're a global tenant administrator, go to **Enterprise applications** in the Azure portal. Select your app registration, and select **Permissions** from the **Security** section of the left pane. Then select the large button labeled **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, 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
```

In that URL:

- Replace `Enter_the_Tenant_Id_Here` 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 that you registered.

You might see the error "AADSTS50011: No reply address is registered for the application" after you grant consent to the app by using the preceding URL. This error happens because this application and the URL don't have a redirect URI. You can ignore it.

Step 5: Run the application

If you're using Visual Studio or Visual Studio for Mac, press F5 to run the application. Otherwise, run the application via command prompt, console, or terminal:

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

In that code:

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

You should see a list of users in Azure Active Directory as result.

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

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, what the main arguments are, and how to add sign-in to an existing .NET Core console application.

MSAL.NET

Microsoft Authentication Library (MSAL, in the [Microsoft.Identity.Client](#) package) is the library that's used to sign in users and request tokens for accessing an API protected by the Microsoft identity platform. This quickstart requests tokens by using the application's own identity instead of delegated permissions. The authentication flow in this case is known as a [client credentials OAuth flow](#). For more information on how to use MSAL.NET with a client credentials flow, see [this article](#).

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

```
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 by using the following code:

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

ELEMENT	DESCRIPTION
<code>config.ClientSecret</code>	The client secret created for the application in the Azure portal.
<code>config.ClientId</code>	The application (client) ID for the application registered in the Azure portal. You can find this value on the app's Overview page in the Azure portal.
<code>config.Authority</code>	(Optional) The security token service (STS) endpoint for the user to authenticate. It's usually <code>https://login.microsoftonline.com/{tenant}</code> for the public cloud, where <code>{tenant}</code> is the name of your tenant or your tenant ID.

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

Requesting tokens

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

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

ELEMENT	DESCRIPTION
<code>scopes</code>	Contains the requested scopes. For confidential clients, this value should use a format similar to <code>{Application ID URI}/.default</code> . This format indicates that the requested scopes are the ones that are 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 in the Azure portal, under Application Registration (Preview) > Expose an API .

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

Help and support

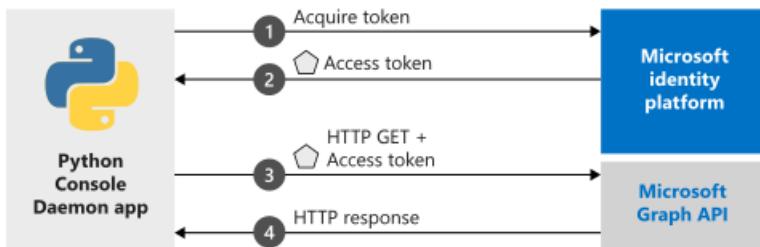
If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

To learn more about daemon applications, see the scenario overview:

[Daemon application that calls web APIs](#)

In this quickstart, you download and run a code sample that demonstrates how a Python application can get an access token using the app's identity to call the Microsoft Graph API and display a [list of users](#) in the directory. The code sample demonstrates how an unattended job or Windows service can 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](#).
2. If you have access to multiple tenants, use the [Directories + subscriptions](#) filter  in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application, for example `Daemon-console`. Users of your app might see this name, and you can change it later.
6. Select **Register**.
7. Under **Manage**, select **Certificates & secrets**.
8. Under **Client secrets**, select **New client secret**, enter a name, and then select **Add**. Record the secret value in a safe location for use in a later step.
9. Under **Manage**, select **API Permissions > Add a permission**. Select **Microsoft Graph**.
10. Select **Application permissions**.
11. Under **User** node, select **User.Read.All**, then select **Add permissions**.

Step 2: Download the Python project

[Download the Python daemon project](#)

Step 3: Configure the 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 **App registrations** in the Azure portal 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, 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:	DESCRIPTION
<code>config["secret"]</code>	Is the client secret created for the application in Azure portal.
<code>config["client_id"]</code>	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.
<code>config["authority"]</code>	The STS endpoint for user to authenticate. Usually <code>https://login.microsoftonline.com/{tenant}</code> 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:	DESCRIPTION
<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 https://graph.microsoft.com). For custom web APIs, <code>{Application ID URI}</code> is defined under the Expose an API section in App registrations in the Azure portal.

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 about your support options, see [Help and support for developers](#).

Next steps

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

[Daemon application that calls web APIs](#)

In this quickstart, you download and run a code sample that demonstrates how a Node.js console application can get an access token using the app's identity to call the Microsoft Graph API and display a [list of users](#) in the directory. The code sample demonstrates how an unattended job or Windows service can run with an application identity, instead of a user's identity.

This quickstart uses the [Microsoft Authentication Library for Node.js \(MSAL Node\)](#) with the [client credentials grant](#).

Prerequisites

- [Node.js](#)
- [Visual Studio Code](#) or another code editor

Register and download the sample application

Follow the steps below to get started.

Step 1: Register the 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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to

- switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
 4. Under **Manage**, select **App registrations > New registration**.
 5. Enter a **Name** for your application, for example `msal-node-cli`. Users of your app might see this name, and you can change it later.
 6. Select **Register**.
 7. Under **Manage**, select **Certificates & secrets**.
 8. Under **Client secrets**, select **New client secret**, enter a name, and then select **Add**. Record the secret value in a safe location for use in a later step.
 9. Under **Manage**, select **API Permissions > Add a permission**. Select **Microsoft Graph**.
 10. Select **Application permissions**.
 11. Under **User** node, select **User.Read.All**, then select **Add permissions**.

Step 2: Download the Node.js sample project

[Download the code sample](#)

Step 3: Configure the Node.js sample project

1. Extract the zip file to a local folder close to the root of the disk, for example, *C:/Azure-Samples*.
2. Edit `.env` and replace the values of the fields `TENANT_ID`, `CLIENT_ID`, and `CLIENT_SECRET` with the following snippet:

```
"TENANT_ID": "Enter_the_Tenant_Id_Here",
"CLIENT_ID": "Enter_the_Application_Id_Here",
"CLIENT_SECRET": "Enter_the_Client_Secret_Here"
```

Where:

- `Enter_the_Application_Id_Here` - is the **Application (client)** ID of the application you registered earlier. Find this ID on the app registration's **Overview** pane in the Azure portal.
- `Enter_the_Tenant_Id_Here` - replace this value with the **Tenant ID** or **Tenant name** (for example, contoso.microsoft.com). Find these values on the app registration's **Overview** pane in the Azure portal.
- `Enter_the_Client_Secret_Here` - replace this value with the client secret you created earlier. To generate a new key, use **Certificates & secrets** in the app registration settings in the Azure portal.

WARNING

Any plaintext secret in source code poses an increased security risk. This article uses a plaintext client secret for simplicity only. Use [certificate credentials](#) instead of client secrets in your confidential client applications, especially those apps you intend to deploy to production.

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 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

Locate the sample's root folder (where `package.json` resides) in a command prompt or console. You'll need to install the dependencies of this sample once:

```
npm install
```

Then, run the application via command prompt or console:

```
node . --op getUsers
```

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

About the code

Below, some of the important aspects of the sample application are discussed.

MSAL Node

[MSAL Node](#) 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 application permissions (using the application's own identity) instead of delegated permissions. The authentication flow used in this case is known as [OAuth 2.0 client credentials flow](#). For more information on how to use MSAL Node with daemon apps, see [Scenario: Daemon application](#).

You can install MSAL Node by running the following npm command.

```
npm install @azure/msal-node --save
```

MSAL initialization

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

```
const msal = require('@azure/msal-node');
```

Then, initialize MSAL using the following code:

```

const msalConfig = {
  auth: {
    clientId: "Enter_the_Application_Id_Here",
    authority: "https://login.microsoftonline.com/Enter_the_Tenant_Id_Here",
    clientSecret: "Enter_the_Client_Secret_Here",
  }
};
const cca = new msal.ConfidentialClientApplication(msalConfig);

```

WHERE:	DESCRIPTION
<code>clientId</code>	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.
<code>authority</code>	The STS endpoint for user to authenticate. Usually <code>https://login.microsoftonline.com/{tenant}</code> for public cloud, where {tenant} is the name of your tenant or your tenant ID.
<code>clientSecret</code>	Is the client secret created for the application in Azure portal.

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

Requesting tokens

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

```

const tokenRequest = {
  scopes: [ 'https://graph.microsoft.com/.default' ],
};

const tokenResponse = await cca.acquireTokenByClientCredential(tokenRequest);

```

WHERE:	DESCRIPTION
<code>tokenRequest</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 Expose an API section in Azure portal's Application Registration.
<code>tokenResponse</code>	The response contains an access token for the scopes requested.

Help and support

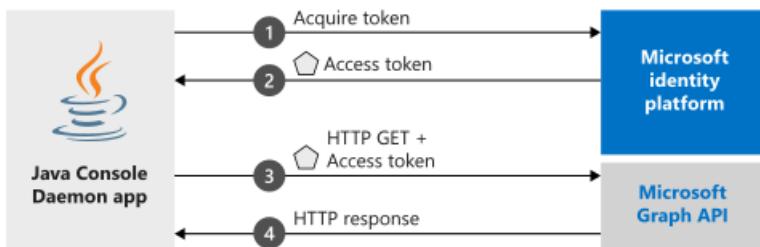
If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

To learn more about daemon/console app development with MSAL Node, see the tutorial:

Daemon application that calls web APIs

In this quickstart, you download and run a code sample that demonstrates how a Java application can get an access token using the app's identity to call the Microsoft Graph API and display a [list of users](#) in the directory. The code sample demonstrates how an unattended job or Windows service can run with an application identity, instead of a user's identity.



Prerequisites

To run this sample, you need:

- [Java Development Kit \(JDK\)](#) 8 or greater
- [Maven](#)

Register and download your quickstart app

You have two options to start your quickstart application: Express (Option 1 below), and Manual (Option 2)

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

1. Go to the [Azure portal - App registrations](#) quickstart experience.
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](#).
2. If you have access to multiple tenants, use the **Directories + subscriptions** filter in the top menu to switch to the tenant in which you want to register the application.
3. Search for and select **Azure Active Directory**.
4. Under **Manage**, select **App registrations > New registration**.
5. Enter a **Name** for your application, for example `Daemon-console`. Users of your app might see this name, and you can change it later.
6. Select **Register**.
7. Under **Manage**, select **Certificates & secrets**.
8. Under **Client secrets**, select **New client secret**, enter a name, and then select **Add**. Record the secret value in a safe location for use in a later step.
9. Under **Manage**, select **API Permissions > Add a permission**. Select **Microsoft Graph**.
10. Select **Application permissions**.
11. Under **User node**, select **User.Read.All**, then select **Add permissions**.

Step 2: Download the Java project

[Download the Java daemon project](#)

Step 3: Configure the Java 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 **msal-client-credential-secret**.
3. Edit *src\main\resources\application.properties* 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 **App registrations** in the Azure portal 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 can test the sample directly by running the main method of **ClientCredentialGrant.java** from your IDE.

From your shell or command line:

```
$ mvn clean compile assembly:single
```

This will generate a **msal-client-credential-secret-1.0.0.jar** file in your */targets* directory. Run this using your Java executable like below:

```
$ java -jar msal-client-credential-secret-1.0.0.jar
```

After running, the application should display the list of users in the configured tenant.

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 `msal-client-credential-certificate`.

More information

MSAL Java

[MSAL Java](#) 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 Java with daemon apps, see [this article](#).

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.

In pom.xml:

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

In build.gradle:

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

MSAL initialization

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

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

Then, initialize MSAL using the following code:

```
IClientCredential credential = ClientCredentialFactory.createFromSecret(CLIENT_SECRET);

ConfidentialClientApplication cca =
    ConfidentialClientApplication
        .builder(CLIENT_ID, credential)
        .authority(AUTHORITY)
        .build();
```

WHERE:	DESCRIPTION
CLIENT_SECRET	Is the client secret created for the application in Azure portal.
CLIENT_ID	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.
AUTHORITY	The STS endpoint for user to authenticate. Usually https://login.microsoftonline.com/{tenant} for public cloud, where {tenant} is the name of your tenant or your tenant Id.

Requesting tokens

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

```
IAuthenticationResult result;
try {
    SilentParameters silentParameters =
        SilentParameters
            .builder(SCOPE)
            .build();

    // try to acquire token silently. This call will fail since the token cache does not
    // have a token for the application you are requesting an access token for
    result = cca.acquireTokenSilently(silentParameters).join();

} catch (Exception ex) {
    if (ex.getCause() instanceof MsalException) {

        ClientCredentialParameters parameters =
            ClientCredentialParameters
                .builder(SCOPE)
                .build();

        // Try to acquire a token. If successful, you should see
        // the token information printed out to console
        result = cca.acquireToken(parameters).join();
    } else {
        // Handle other exceptions accordingly
        throw ex;
    }
}
return result;
```

WHERE:	DESCRIPTION
SCOPE	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 https://graph.microsoft.com). For custom web APIs, <code>{Application ID URI}</code> is defined under the Expose an API section in App registrations in the Azure portal.

Help and support

If you need help, want to report an issue, or want to learn about your support options, see [Help and support for developers](#).

Next steps

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

[Daemon application that calls web APIs](#)

Tutorial: Build a multi-tenant daemon that uses the Microsoft identity platform

4/12/2022 • 11 minutes to read • [Edit Online](#)

In this tutorial, you download and run an ASP.NET daemon web app that demonstrates using the OAuth 2.0 client credentials grant to get an access token to call the Microsoft Graph API.

In this tutorial:

- Integrate a daemon app with the Microsoft identity platform
- Grant application permissions directly to the app by an admin
- Get an access token to call the Microsoft Graph API
- Call the Microsoft Graph API.

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

- [Visual Studio 2017 or 2019](#).
- An Azure AD tenant. For more information, see [How to get an Azure AD tenant](#).
- One or more user accounts in your Azure AD tenant. This sample won't work with a Microsoft account. If you signed in to the [Azure portal](#) with a Microsoft account and have never created a user account in your directory, do that now.

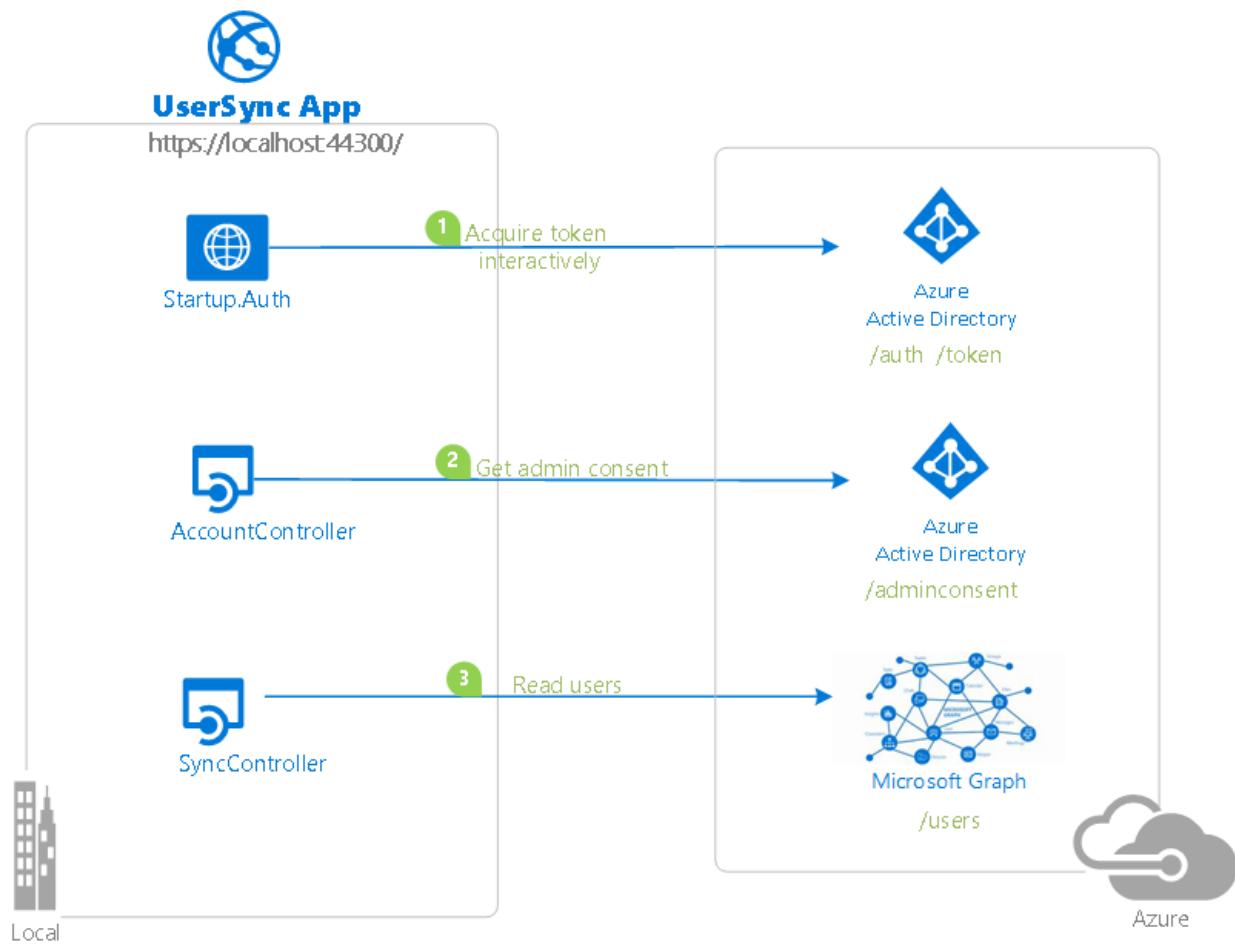
Scenario

The app is built as an ASP.NET MVC application. It uses the OWIN OpenID Connect middleware to sign in users.

The "daemon" component in this sample is an API controller, `SyncController.cs`. When the controller is called, it pulls in a list of users in the customer's Azure Active Directory (Azure AD) tenant from Microsoft Graph.

`SyncController.cs` is triggered by an AJAX call in the web application. It uses the [Microsoft Authentication Library \(MSAL\) for .NET](#) to acquire an access token for Microsoft Graph.

Because the app is a multi-tenant app for Microsoft business customers, it must provide a way for customers to "sign up" or "connect" the application to their company data. During the connection flow, a Global Administrator first grants *application permissions* directly to the app so that it can access company data in a non-interactive fashion, without the presence of a signed-in user. The majority of the logic in this sample shows how to achieve this connection flow by using the identity platform's [admin consent endpoint](#).



For more information on the concepts used in this sample, read the [client credentials protocol documentation for the identity platform](#).

Clone or download this repository

From your shell or command line, enter this command:

```
git clone https://github.com/Azure-Samples/active-directory-dotnet-daemon-v2.git
```

Or [download the sample in a zip file](#).

Register your application

This sample has one project. To register the application with your Azure AD tenant, you can either:

- Follow the steps in [Register the sample with your Azure Active Directory tenant](#) and [Configure the sample to use your Azure AD tenant](#).
- Use PowerShell scripts that:
 - Automatically* create the Azure AD applications and related objects (passwords, permissions, dependencies) for you.
 - Modify the Visual Studio projects' configuration files.

If you want to use the automation:

- On Windows, run PowerShell and go to the root of the cloned directory.
- Run this command:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process -Force
```

- Run the script to create your Azure AD application and configure the code of the sample application accordingly:

```
.\AppCreationScripts\Configure.ps1
```

Other ways of running scripts are described in [App creation scripts](#).

- Open the Visual Studio solution and select **Start** to run the code.

If you don't want to use the automation, use the steps in the following sections.

Choose the Azure AD tenant

- Sign in to the [Azure portal](#).
- If you have access to multiple tenants, use the **Directories + subscriptions** filter  in the top menu to switch to the tenant in which you want to register the application.

Register the client app (`dotnet-web-daemon-v2`)

- Search for and select **Azure Active Directory**.
- Under **Manage**, select **App registrations > New registration**.
- Enter a **Name** for your application, for example `dotnet-web-daemon-v2`. Users of your app might see this name, and you can change it later.
- In the **Supported account types** section, select **Accounts in any organizational directory**.
- In the **Redirect URI (optional)** section, select **Web** in the combo box and enter `https://localhost:44316/` and `https://localhost:44316/Account/GrantPermissions` as Redirect URIs.

If there are more than two redirect URIs, you'll need to add them from the **Authentication** tab later, after the app is created successfully.

- Select **Register** to create the application.
- On the app's **Overview** page, find the **Application (client) ID** value and record it for later use. You'll need it to configure the Visual Studio configuration file for this project.
- Under **Manage**, select **Authentication**.
- Set **Front-channel logout URL** to `https://localhost:44316/Account/EndSession`.
- In the **Implicit grant and hybrid flows** section, select **Access tokens** and **ID tokens**. This sample requires the [implicit grant flow](#) to be enabled to sign in the user and call an API.
- Select **Save**.
- Under **Manage**, select **Certificates & secrets**.
- In the **Client secrets** section, select **New client secret**.
- Enter a key description (for example, **app secret**).
- Select a key duration of either **In 1 year**, **In 2 years**, or **Never Expires**.
- Select **Add**. Record the key value in a safe location. You'll need this key later to configure the project in Visual Studio.

17. Under **Manage**, select **API permissions** > **Add a permission**.
18. In the **Commonly used Microsoft APIs** section, select **Microsoft Graph**.
19. In the **Application permissions** section, ensure that the right permissions are selected: **User.Read.All**.
20. Select **Add permissions**.

Configure the sample to use your Azure AD tenant

In the following steps, **ClientId** is the same as "application ID" or **AppId**.

Open the solution in Visual Studio to configure the projects.

Configure the client project

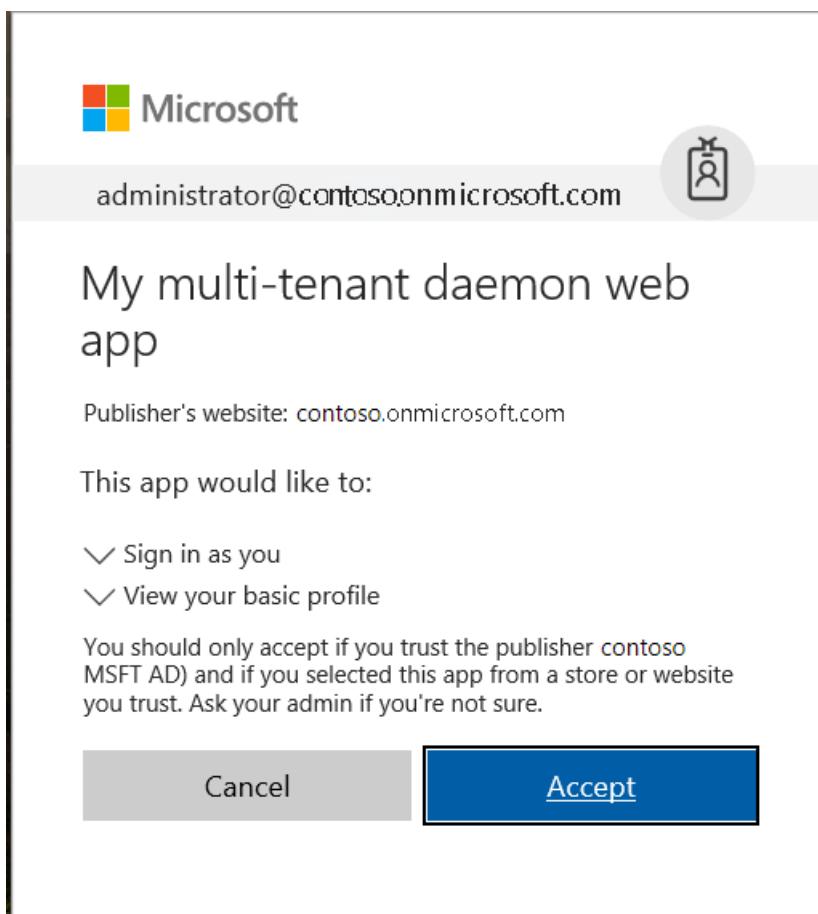
If you used the setup scripts, the following changes will have been applied for you.

1. Open the **UserSync\Web.Config** file.
2. Find the app key **ida:ClientId**. Replace the existing value with the application ID of the **dotnet-web-daemon-v2** application copied from the Azure portal.
3. Find the app key **ida:ClientSecret**. Replace the existing value with the key that you saved during the creation of the **dotnet-web-daemon-v2** app in the Azure portal.

Run the sample

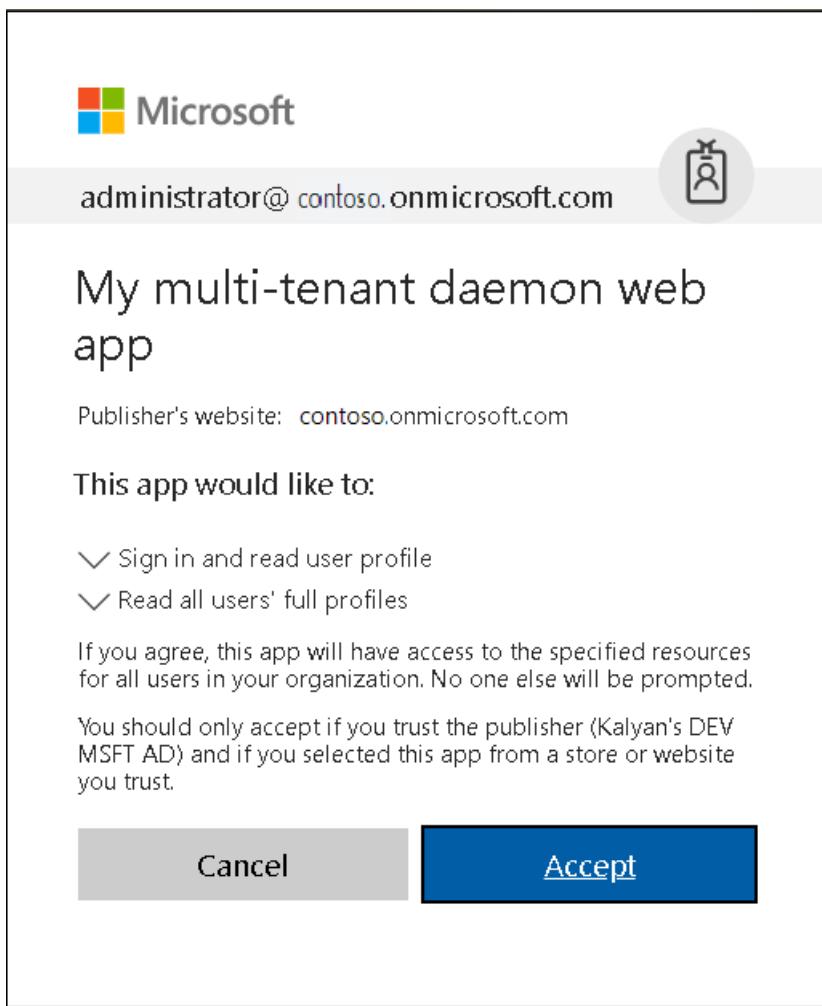
Clean the solution, rebuild the solution, run the UserSync application, and then sign in as an administrator in your Azure AD tenant. If you don't have an Azure AD tenant for testing, you can [follow these instructions](#) to get one.

When you sign in, the app first asks you for permission to sign you in and read your user profile. This consent allows the app to ensure that you're a business user.



The app then tries to sync a list of users from your Azure AD tenant, via Microsoft Graph. If it can't, it asks you (the tenant administrator) to connect your tenant to the app.

The app then asks for permission to read the list of users in your tenant.



After you grant permission, you're signed out from the app. This sign-out ensures that any existing access tokens for Microsoft Graph is removed from the token cache. When you sign in again, the fresh token that's obtained will have the necessary permissions to make calls to Microsoft Graph.

When you grant the permission, the app can then query for users at any point. You can verify this by selecting the **Sync Users** button and refreshing the list of users. Try adding or removing a user and resyncing the list. (But note that the app syncs only the first page of users.)

About the code

The relevant code for this sample is in the following files:

- **App_Start\Startup.Auth.cs, Controllers\AccountController.cs:** Initial sign-in. In particular, the actions on the controller have an **Authorize** attribute, which forces the user to sign in. The application uses the [authorization code flow](#) to sign in the user.
- **Controllers\SyncController.cs:** Syncing the list of users to the local in-memory store.
- **Controllers\UserController.cs:** Displaying the list of users from the local in-memory store.
- **Controllers\AccountController.cs:** Acquiring permissions from the tenant admin by using the admin consent endpoint.

Re-create the sample app

1. In Visual Studio, create a new **Visual C# ASP.NET Web Application (.NET Framework)** project.

2. On the next screen, choose the **MVC** project template. Also add folder and core references for **Web API**, because you'll add a web API controller later. Leave the project's chosen authentication mode as the default: **No Authentication**.
3. Select the project in the **Solution Explorer** window and select the **F4** key.
4. In the project properties, set **SSL Enabled** to **True**. Note the information in **SSL URL**. You'll need it when configuring this application's registration in the Azure portal.
5. Add the following ASP.NET OWIN middleware NuGet packages:
 - Microsoft.Owin.Security.ActiveDirectory
 - Microsoft.Owin.Security.Cookies
 - Microsoft.Owin.Host.SystemWeb
 - Microsoft.IdentityModel.Protocol.Extensions
 - Microsoft.Owin.Security.OpenIdConnect
 - Microsoft.Identity.Client
6. In the **App_Start** folder:
 - a. Create a class called **Startup.Auth.cs**.
 - b. Remove **.App_Start** from the namespace name.
 - c. Replace the code for the **Startup** class with the code from the same file of the sample app. Be sure to take the whole class definition. The definition changes from **public class Startup** to **public partial class Startup**.
7. In **Startup.Auth.cs**, resolve missing references by adding **using** statements as suggested by Visual Studio IntelliSense.
8. Right-click the project, select **Add**, and then select **Class**.
9. In the search box, enter **OWIN**. **OWIN Startup class** appears as a selection. Select it, and name the class **Startup.cs**.
10. In **Startup.cs**, replace the code for the **Startup** class with the code from the same file of the sample app. Again, note that the definition changes from **public class Startup** to **public partial class Startup**.
11. In the **Models** folder, add a new class called **MsGraphUser.cs**. Replace the implementation with the contents of the file of the same name from the sample.
12. Add a new **MVC 5 Controller - Empty** instance called **AccountController**. Replace the implementation with the contents of the file of the same name from the sample.
13. Add a new **MVC 5 Controller - Empty** instance called **UserController**. Replace the implementation with the contents of the file of the same name from the sample.
14. Add a new **Web API 2 Controller - Empty** instance called **SyncController**. Replace the implementation with the contents of the file of the same name from the sample.
15. For the user interface, in the **Views\Account** folder, add three **Empty (without model) Views** instances named **GrantPermissions**, **Index**, and **UserMismatch**. Add one named **Index** in the **Views\User** folder. Replace the implementation with the contents of the file of the same name from the sample.
16. Update **Shared_Layout.cshtml** and **Home\Index.cshtml** to correctly link the various views together.

Deploy the sample to Azure

This project has web app and web API projects. To deploy them to Azure websites, take the following steps for each one:

1. Create an Azure website.
2. Publish the web app and web APIs to the website.
3. Update clients to call the website instead of IIS Express.

Create and publish dotnet-web-daemon-v2 to an Azure website

1. Sign in to the [Azure portal](#).

2. In the upper-left corner, select **Create a resource**.
3. Select **Web > Web App**, and then give your website a name. For example, name it **dotnet-web-daemon-v2-contoso.azurewebsites.net**.
4. Select the information for **Subscription**, **Resource group**, and **App service plan and location**. OS is **Windows**, and **Publish is Code**.
5. Select **Create** and wait for the app service to be created.
6. When you get the **Deployment succeeded** notification, select **Go to resource** to go to the newly created app service.
7. After the website is created, find it in the **Dashboard** and select it to open the app service's **Overview** screen.
8. From the **Overview** tab of the app service, download the publish profile by selecting the **Get publish profile** link and save it. You can use other deployment mechanisms, such as deploying from source control.
9. Switch to Visual Studio and then:
 - a. Go to the **dotnet-web-daemon-v2** project.
 - b. Right-click the project in Solution Explorer, and then select **Publish**.
 - c. Select **Import Profile** on the bottom bar, and import the publish profile that you downloaded earlier.
10. Select **Configure**.
11. On the **Connection** tab, update the destination URL so that it uses "https." For example, use `https://dotnet-web-daemon-v2-contoso.azurewebsites.net`. Select **Next**.
12. On the **Settings** tab, make sure that **Enable Organizational Authentication** is cleared.
13. Select **Save**. Select **Publish** on the main screen.

Visual Studio will publish the project and automatically open a browser to the project's URL. If you see the default webpage of the project, the publication was successful.

Update the Azure AD tenant application registration for **dotnet-web-daemon-v2**

1. Go back to the [Azure portal](#).
2. In the left pane, select the **Azure Active Directory** service, and then select **App registrations**.
3. Select the **dotnet-web-daemon-v2** application.
4. On the **Authentication** page for your application, update the **Front-channel logout URL** fields with the address of your service. For example, use `https://dotnet-web-daemon-v2-contoso.azurewebsites.net/Account/EndSession`.
5. From the **Branding** menu, update the **Home page URL** to the address of your service. For example, use `https://dotnet-web-daemon-v2-contoso.azurewebsites.net`.
6. Save the configuration.
7. Add the same URL in the list of values of the **Authentication > Redirect URIs** menu. If you have multiple redirect URLs, make sure that there's a new entry that uses the app service's URI for each redirect URL.

Clean up resources

When no longer needed, delete the app object that you created in the [Register your application](#) step. To remove the application, follow the instructions in [Remove an application authored by you or your organization](#).

Get help

Use [Microsoft Q&A](#) to get support from the community. Ask your questions on [Microsoft Q&A](#) 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-ad-adal-deprecation," "azure-ad-msal," and "dotnet-standard."

If you find a bug in the sample, please raise the issue on [GitHub Issues](#).

If you find a bug in MSAL.NET, please raise the issue on [MSAL.NET GitHub Issues](#).

To provide a recommendation, go to the [User Voice page](#).

Next steps

Learn more about building daemon apps that use the Microsoft identity platform to access protected web APIs:

[Scenario: Daemon application that calls web APIs](#)

Tutorial: Call the Microsoft Graph API in a Node.js console app

4/12/2022 • 6 minutes to read • [Edit Online](#)

In this tutorial, you build a console app that calls Microsoft Graph API using its own identity. The console app you build uses the [Microsoft Authentication Library \(MSAL\) for Node.js](#).

Follow the steps in this tutorial to:

- Register the application in the Azure portal
- Create a Node.js console app project
- Add authentication logic to your app
- Add app registration details
- Add a method to call a web API
- Test the app

Prerequisites

- [Node.js](#)
- [Visual Studio Code](#) or another code editor

Register the application

First, complete the steps in [Register an application with the Microsoft identity platform](#) to register your app.

Use the following settings for your app registration:

- Name: `NodeConsoleApp` (suggested)
- Supported account types: **Accounts in this organizational directory only**
- API permissions: **Microsoft APIs > Microsoft Graph > Application Permissions >** `User.Read.All`
- Client secret: `*****` (record this value for use in a later step - it's shown only once)

Create the project

1. Start by creating a directory for this Node.js tutorial project. For example, `NodeConsoleApp`.
2. In your terminal, change into the directory you created (the project root), and then run the following commands:

```
npm init -y
npm install --save dotenv yargs axios @azure/msal-node
```

3. Next, edit the `package.json` file in the project root and prefix the value of `main` with `bin/`, like this:

```
"main": "bin/index.js",
```

4. Now create the `bin` directory, and inside `bin`, add the following code to a new file named `index.js`:

```

#!/usr/bin/env node

// read in env settings
require('dotenv').config();

const yargs = require('yargs');

const fetch = require('./fetch');
const auth = require('./auth');

const options = yargs
  .usage('Usage: --op <operation_name>')
  .option('op', { alias: 'operation', describe: 'operation name', type: 'string', demandOption: true })
  .argv;

async function main() {
  console.log(`You have selected: ${options.op}`);

  switch (yargs.argv['op']) {
    case 'getUsers':

      try {
        // here we get an access token
        const authResponse = await auth.getToken(auth.tokenRequest);

        // call the web API with the access token
        const users = await fetch.callApi(auth.apiConfig.uri, authResponse.accessToken);

        // display result
        console.log(users);
      } catch (error) {
        console.log(error);
      }

      break;
    default:
      console.log('Select a Graph operation first');
      break;
  }
}

main();

```

The *index.js* file you just created references two other node modules that you'll create next:

- *auth.js* - Uses MSAL Node for acquiring access tokens from the Microsoft identity platform.
- *fetch.js* - Requests data from the Microsoft Graph API by including access tokens (acquired in *auth.js*) in HTTP requests to the API.

At the end of the tutorial, your project's file and directory structure should look similar to this:

```

NodeConsoleApp/
├── bin
│   ├── auth.js
│   ├── fetch.js
│   └── index.js
└── package.json
└── .env

```

Add authentication logic

Inside the `bin` directory, add the following code to a new file named `auth.js`. The code in `auth.js` acquires an access token from the Microsoft identity platform for including in Microsoft Graph API requests.

```
const msal = require('@azure/msal-node');

/**
 * Configuration object to be passed to MSAL instance on creation.
 * For a full list of MSAL Node configuration parameters, visit:
 * https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-node/docs/configuration.md
 */
const msalConfig = {
    auth: {
        clientId: process.env.CLIENT_ID,
        authority: process.env.AAD_ENDPOINT + '/' + process.env.TENANT_ID,
        clientSecret: process.env.CLIENT_SECRET,
    }
};

/***
 * With client credentials flows permissions need to be granted in the portal by a tenant administrator.
 * The scope is always in the format '<resource>/default'. For more, visit:
 * https://docs.microsoft.com/azure/active-directory/develop/v2-oauth2-client-creds-grant-flow
 */
const tokenRequest = {
    scopes: [process.env.GRAPH_ENDPOINT + '/.default'],
};

const apiConfig = {
    uri: process.env.GRAPH_ENDPOINT + '/v1.0/users',
};

/***
 * Initialize a confidential client application. For more info, visit:
 * https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-node/docs/initialize-confidential-client-application.md
 */
const cca = new msal.ConfidentialClientApplication(msalConfig);

/***
 * Acquires token with client credentials.
 * @param {object} tokenRequest
 */
async function getToken(tokenRequest) {
    return await cca.acquireTokenByClientCredential(tokenRequest);
}

module.exports = {
    apiConfig: apiConfig,
    tokenRequest: tokenRequest,
    getToken: getToken
};
```

In the code snippet above, we first create a configuration object (`msalConfig`) and pass it to initialize an MSAL [ConfidentialClientApplication](#). Then we create a method for acquiring tokens via `client credentials` and finally expose this module to be accessed by `main.js`. The configuration parameters in this module are drawn from an environment file, which we will create in the next step.

Add app registration details

Create an environment file to store the app registration details that will be used when acquiring tokens. To do so, create a file named `.env` inside the root folder of the sample (`NodeConsoleApp`), and add the following code:

```
# Credentials
TENANT_ID=Enter_the_Tenant_Id_Here
CLIENT_ID=Enter_the_Application_Id_Here
CLIENT_SECRET=Enter_the_Client_Secret_Here

# Endpoints
AAD_ENDPOINT=Enter_the_Cloud_Instance_Id_Here
GRAPH_ENDPOINT=Enter_the_Graph_Endpoint_Here
```

Fill in these details with the values you obtain from Azure app registration portal:

- `Enter_the_Tenant_Id_Here` should be one of the following:
 - 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`.
- `Enter_the_Application_Id_Here` : The **Application (client) ID** of the application you registered.
- `Enter_the_Cloud_Instance_Id_Here` : The Azure cloud instance in which your application is registered.
 - For the main (or *global*) Azure cloud, enter `https://login.microsoftonline.com`.
 - For **national** clouds (for example, China), you can find appropriate values in [National clouds](#).
- `Enter_the_Graph_Endpoint_Here` is the instance of the Microsoft Graph API the application should communicate with.
 - For the **global** Microsoft Graph API endpoint, replace both instances of this string with `https://graph.microsoft.com`.
 - For endpoints in **national** cloud deployments, see [National cloud deployments](#) in the Microsoft Graph documentation.

Add a method to call a web API

Inside the `bin` folder, create another file named `fetch.js` and add the following code for making REST calls to the Microsoft Graph API:

```

const axios = require('axios');

/**
 * Calls the endpoint with authorization bearer token.
 * @param {string} endpoint
 * @param {string} accessToken
 */
async function callApi(endpoint, accessToken) {

    const options = {
        headers: {
            Authorization: `Bearer ${accessToken}`
        }
    };

    console.log('request made to web API at: ' + new Date().toString());

    try {
        const response = await axios.default.get(endpoint, options);
        return response.data;
    } catch (error) {
        console.log(error)
        return error;
    }
};

module.exports = {
    callApi: callApi
};

```

Here, the `callApi` method is used to make an HTTP `GET` request against a protected resource that requires an access token. The request then returns the content to the caller. This method adds the acquired token in the *HTTP Authorization header*. The protected resource here is the Microsoft Graph API [users endpoint](#) which displays the users in the tenant where this app is registered.

Test the app

You've completed creation of the application and are now ready to test the app's functionality.

Start the Node.js console app by running the following command from within the root of your project folder:

```
node . --op getUsers
```

This should result in some JSON response from Microsoft Graph API and you should see an array of user objects in the console:

```
You have selected: getUsers
request made to web API at: Fri Jan 22 2021 09:31:52 GMT-0800 (Pacific Standard Time)
{
  '@odata.context': 'https://graph.microsoft.com/v1.0/$metadata#users',
  value: [
    {
      displayName: 'Adele Vance',
      givenName: 'Adele',
      jobTitle: 'Retail Manager',
      mail: 'AdeleV@msaltestingjs.onmicrosoft.com',
      mobilePhone: null,
      officeLocation: '18/2111',
      preferredLanguage: 'en-US',
      surname: 'Vance',
      userPrincipalName: 'AdeleV@msaltestingjs.onmicrosoft.com',
      id: 'a6a218a5-f5ae-462a-acd3-581af4bcc00'
    }
  ]
}
```

```
$ node . --op getUsers
You have selected: getUsers
request made to web API at: Fri Jan 22 2021 09:31:52 GMT-0800 (Pacific Standard Time)
{
  '@odata.context': 'https://graph.microsoft.com/v1.0/$metadata#users',
  value: [
    {
      businessPhones: [Array],
      displayName: 'Adele Vance',
      givenName: 'Adele',
      jobTitle: 'Retail Manager',
      mail: 'AdeleV@msaltestingjs.onmicrosoft.com',
      mobilePhone: null,
      officeLocation: '18/2111',
      preferredLanguage: 'en-US',
      surname: 'Vance',
      userPrincipalName: 'AdeleV@msaltestingjs.onmicrosoft.com',
      id: 'a6a218a5-f5ae-462a-acd3-581af3bcc00'
    },
    {

```

How the application works

This application uses [OAuth 2.0 client credentials grant](#). This type of grant is commonly used for server-to-server interactions that must run in the background, without immediate interaction with a user. The 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. The type of applications supported with this authentication model are usually **daemons or service accounts**.

The scope to request for a client credential flow is the name of the resource followed by `/default`. This notation tells Azure Active Directory (Azure AD) to use the application-level permissions declared statically during application registration. Also, these API permissions must be granted by a **tenant administrator**.

Next steps

If you'd like to dive deeper into Node.js console application development on the Microsoft identity platform, see our multi-part scenario series:

[Scenario: Daemon application](#)

Microsoft identity platform code samples

4/12/2022 • 9 minutes to read • [Edit Online](#)

These code samples are built and maintained by Microsoft to demonstrate usage of our authentication libraries with the Microsoft identity platform. Common authentication and authorization scenarios are implemented in several [application types](#), development languages, and frameworks.

- Sign in users to web applications and provide authorized access to protected web APIs.
- Protect a web API by requiring an access token to perform API operations.

Each code sample includes a *README.md* file describing how to build the project (if applicable) and run the sample application. Comments in the code help you understand how these libraries are used in the application to perform authentication and authorization by using the identity platform.

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Angular	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Call .NET Core web API• Call .NET Core web API (B2C)• Call Microsoft Graph via OBO• Call .NET Core web API using PoP• Use App Roles for access control• Use Security Groups for access control• Deploy to Azure Storage and App Service	MSAL Angular	<ul style="list-style-type: none">• Authorization code with PKCE• On-behalf-of (OBO)• Proof of Possession (PoP)
Blazor WebAssembly	<ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Deploy to Azure App Service	MSAL.js	Implicit Flow

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
JavaScript	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API via OBO and CA • Deploy to Azure Storage and App Service 	MSAL.js	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA)
React	<ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call Node.js web API • Call Node.js web API (B2C) • Call Microsoft Graph via OBO • Call Node.js web API using PoP • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service • Deploy to Azure Static Web Apps 	MSAL React	<ul style="list-style-type: none"> • Authorization code with PKCE • On-behalf-of (OBO) • Conditional Access (CA) • Proof of Possession (PoP)

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.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Customize token cache • Call Graph (multi-tenant) • Call Azure REST APIs • Protect web API • Protect web API (B2C) • Protect multi-tenant web API • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure Storage and App Service 	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	<ul style="list-style-type: none"> • OpenID connect • Authorization code • On-Behalf-Of

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Blazor	Blazor Server Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Call web API • Call web API (B2C) 	MSAL.NET	Authorization code Grant Flow
ASP.NET Core	Advanced Token Cache Scenarios	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	On-Behalf-Of (OBO)
ASP.NET Core	Use the Conditional Access auth context to perform step-up authentication	<ul style="list-style-type: none"> • MSAL.NET • Microsoft.Identity.Web 	Authorization code
ASP.NET Core	Active Directory FS to Azure AD migration	MSAL.NET	<ul style="list-style-type: none"> • SAML • OpenID connect
ASP.NET	<ul style="list-style-type: none"> • Microsoft Graph Training Sample • Sign in users and call Microsoft Graph • Sign in users and call Microsoft Graph with admin restricted scope • Quickstart: Sign in users 	MSAL.NET	<ul style="list-style-type: none"> • OpenID connect • Authorization code
Java Spring	Azure AD Spring Boot Starter Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Groups for access control • Deploy to Azure App Service 	<ul style="list-style-type: none"> • MSAL Java • Azure AD Boot Starter 	Authorization code
Java Servlets	Spring-less Servlet Series <ul style="list-style-type: none"> • Sign in users • Sign in users (B2C) • Call Microsoft Graph • Use App Roles for access control • Use Security Groups for access control • Deploy to Azure App Service 	MSAL Java	Authorization code
Java	Sign in users and call Microsoft Graph	MSAL Java	Authorization code
Java Spring	Sign in users and call Microsoft Graph via OBO • Web API	MSAL Java	<ul style="list-style-type: none"> • Authorization code • On-Behalf-Of (OBO)

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js Express	Express web app series <ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Deploy to Azure App Service• Use App Roles for access control• Use Security Groups for access control• Web app that sign in users	MSAL Node	Authorization code
Python Flask	Flask Series <ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Sign in users and call Microsoft Graph• Call Microsoft Graph• Deploy to Azure App Service	MSAL Python	Authorization code
Python Django	Django Series <ul style="list-style-type: none">• Sign in users• Sign in users (B2C)• Call Microsoft Graph• Deploy to Azure App Service	MSAL Python	Authorization code
Ruby	Graph Training <ul style="list-style-type: none">• Sign in users and call Microsoft Graph	OmniAuth OAuth2	Authorization code

Web API

The following samples show how to protect a web API with the Microsoft identity platform, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET	Call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
ASP.NET Core	Sign in users and call Microsoft Graph	MSAL.NET	On-Behalf-Of (OBO)
Java	Sign in users	MSAL Java	On-Behalf-Of (OBO)
Node.js	<ul style="list-style-type: none">• Protect a Node.js web API• Protect a Node.js Web API with Azure AD B2C	MSAL Node	Authorization bearer

Desktop

The following samples show public client desktop applications that access the Microsoft Graph API, or your own

web API in the name of the user. Apart from the *Desktop (Console) with Web Authentication Manager (WAM)* sample, all these client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> • Call Microsoft Graph • Call Microsoft Graph with token cache • Call Microsoft Graph with custom web UI HTML • Call Microsoft Graph with custom web browser • Sign in users with device code flow 	MSAL.NET	<ul style="list-style-type: none"> • Authorization code with PKCE • Device code
.NET	<ul style="list-style-type: none"> • Call Microsoft Graph with daemon console • Call web API with daemon console 	MSAL.NET	Authorization code with PKCE
.NET	Invoke protected API with integrated Windows authentication	MSAL.NET	Integrated Windows authentication
Java	Call Microsoft Graph	MSAL Java	Integrated Windows authentication
Node.js	Sign in users	MSAL Node	Authorization code with PKCE
PowerShell	Call Microsoft Graph by signing in users using username/password	MSAL.NET	Resource owner password credentials
Python	Sign in users	MSAL Python	Resource owner password credentials
Universal Window Platform (UWP)	Call Microsoft Graph	MSAL.NET	Web account manager
Windows Presentation Foundation (WPF)	Sign in users and call Microsoft Graph	MSAL.NET	Authorization code with PKCE
XAML	<ul style="list-style-type: none"> • Sign in users and call ASP.NET core web API • Sign in users and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Mobile

The following samples show public client mobile applications that access the Microsoft Graph API, or your own web API in the name of the user. These client applications use the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
iOS	<ul style="list-style-type: none"> Call Microsoft Graph native Call Microsoft Graph with Azure AD nxauth 	MSAL iOS	Authorization code with PKCE
Java	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Kotlin	Sign in users and call Microsoft Graph	MSAL Android	Authorization code with PKCE
Xamarin	<ul style="list-style-type: none"> Sign in users and call Microsoft Graph Sign in users with broker and call Microsoft Graph 	MSAL.NET	Authorization code with PKCE

Service / daemon

The following samples show an application that accesses the Microsoft Graph API with its own identity (with no user).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET Core	<ul style="list-style-type: none"> Call Microsoft Graph Call web API Call own web API Using managed identity and Azure key vault 	MSAL.NET	Client credentials grant
ASP.NET	Multi-tenant with Microsoft identity platform endpoint	MSAL.NET	Client credentials grant
Java	Call Microsoft Graph	MSAL Java	Client credentials grant
Node.js	Sign in users and call web API	MSAL Node	Client credentials grant
Python	<ul style="list-style-type: none"> Call Microsoft Graph with secret Call Microsoft Graph with certificate 	MSAL Python	Client credentials grant

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, and how to call a downstream API from the web API.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET	.NET Azure function web API secured by Azure AD	MSAL.NET	Authorization code

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Node.js Azure function web API secured by Azure AD	MSAL Node	Authorization bearer
Node.js	Call Microsoft Graph API on behalf of a user	MSAL Node	On-Behalf-Of (OBO)
Python	Python Azure function web API secured by Azure AD	MSAL Python	Authorization code

Headless

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 the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
.NET core	Invoke protected API from text-only device	MSAL.NET	Device code
Java	Sign in users and invoke protected API	MSAL Java	Device code
Python	Call Microsoft Graph	MSAL Python	Device code

Microsoft Teams applications

The following sample illustrates Microsoft Teams Tab application that signs in users. Additionally it demonstrates how to call Microsoft Graph API with the user's identity using the Microsoft Authentication Library (MSAL).

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
Node.js	Teams Tab app: single sign-on (SSO) and call Microsoft Graph	MSAL Node	On-Behalf-Of (OBO)

Multi-tenant SaaS

The following samples show how to configure your application to accept sign-ins from any Azure Active Directory (Azure AD) tenant. Configuring your application to be *multi-tenant* means that you can offer a **Software as a Service** (SaaS) application to many organizations, allowing their users to be able to sign-in to your application after providing consent.

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW

LANGUAGE/ PLATFORM	CODE SAMPLE(S) ON GITHUB	AUTH LIBRARIES	AUTH FLOW
ASP.NET Core	ASP.NET Core MVC web application calls Microsoft Graph API	MSAL.NET	OpenID connect
ASP.NET Core	ASP.NET Core MVC web application calls ASP.NET Core Web API	MSAL.NET	Authorization code

Next steps

If you'd like to delve deeper into more sample code, see:

- [Sign in users and call the Microsoft Graph API from an Angular](#)
- [Sign in users in a Nodejs and Express web app](#)
- [Call the Microsoft Graph API from a Universal Windows Platform](#)

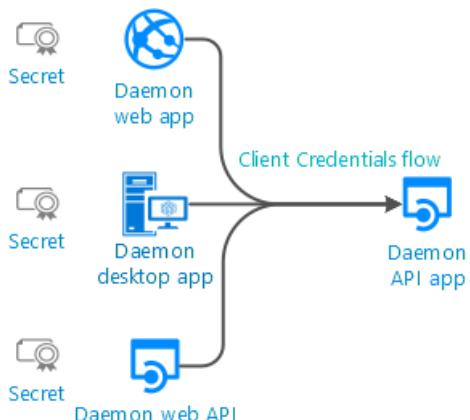
Scenario: Daemon application that calls web APIs

4/12/2022 • 3 minutes to read • [Edit Online](#)

Learn all you need to build a daemon application that calls web APIs.

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 uses 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 (like Windows services on Windows or daemon processes on Linux) that perform batch jobs, or an operating system service that runs 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 under their own identity for technical reasons. An example is access to secrets in Azure Key Vault or Azure SQL Database for a cache.

NOTE

You can't deploy a daemon application to a regular user's device, and a regular user can't access a daemon application. Only a limited set of IT administrators can access devices that have daemon applications running, so a bad actor can't access a client secret or token from device traffic and act on behalf of the daemon application. The daemon application scenario doesn't replace device authentication.

Examples of non-daemon applications:

- A mobile application that accesses a web service on behalf of an application, but not on behalf of a user.
- An IoT device that accesses a web service on behalf of a device, but not on behalf of a user.

Applications that acquire a token for their own identities:

- Are confidential client applications. These apps, given that they access resources independently of users, need to prove their identity. They're also rather sensitive apps. 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

Users can't interact 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. This token is then used to call the web API (and is refreshed as needed).

Because users can't interact with daemon applications, incremental consent isn't possible. All the required API permissions need to be configured at application registration. 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 work only 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 multitenant daemon application. Each tenant admin will need to provide consent.
- During [application registration](#), the reply URI isn't needed. Share secrets or certificates or signed assertions with Azure AD. You also need to request application 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.

Recommended reading

If you're new to identity and access management (IAM) with OAuth 2.0 and OpenID Connect, or even just new to IAM on the Microsoft identity platform, the following set of articles should be high on your reading list.

Although not required reading before completing your first quickstart or tutorial, they cover topics integral to the platform, and familiarity with them will help you on your path as you build more complex scenarios.

Authentication and authorization

- [Authentication basics](#)
- [ID tokens](#)
- [Access tokens](#)

Microsoft identity platform

- [Audiences](#)
- [Applications and service principals](#)
- [Permissions and consent](#)

Next steps

Move on to the next article in this scenario, [App registration](#).

Daemon app that calls web APIs - app registration

4/12/2022 • 2 minutes to read • [Edit Online](#)

For a daemon application, here's what you need to know when you register the app.

Supported account types

Daemon applications make sense only in Azure ActiveDirectory (Azure AD) tenants. So when you create the application, choose one of the following options:

- **Accounts in this organizational directory only.** This choice is the most common one because daemon applications are written by line-of-business (LOB) developers.
- **Accounts in any organizational directory.** You'll make this choice if you're an Independent Software Vendor (ISV) providing a utility tool to your customers. You'll need your customers' tenant 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 for the application configuration or construction. The client credentials flow doesn't use it.

API permissions - app permissions and admin consent

A daemon application can request only application permissions to APIs (not delegated permissions). On the **API permissions** 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 Azure portal interface with the URL https://portal.azure.com/#blade/Microsoft_AAD_RegisteredApps/ApplicationMenuBlade/CallAnAPI/appid/8c76dc8-5e7b-4dcc-a137-3f966a6e1b58/objectId/2539068d-4645-4d06-9353-b8577dc4b205/isMSA. The left sidebar shows 'Create a resource', 'Home', 'Dashboard', and 'All services'. Under 'FAVORITES', there are links for 'All resources', 'Resource groups', 'App Services', 'SQL databases', 'SQL data warehouses', '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 main content area is titled 'dotnetcore-daemon-v2 - API permissions' and shows the 'Request API permissions' blade. It includes sections for 'API permissions', 'Grant consent', and 'Select permissions'. The 'Select permissions' section has a search bar and a list of API permissions. A callout box highlights the 'Application permissions' section, which is described as 'Your application runs as a background service or daemon without a signed-in user.' The 'ADMIN CONSENT REQUIRED' column indicates that admin consent is required for some permissions.

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 that a tenant admin pre-consent to the application calling the web API. Tenant admins provide this consent on the same **API permission** page by selecting **Grant admin consent to *our organization***.

If you're an ISV building a multitenant application, you should read the section [Deployment - case of multitenant daemon apps](#).

Add a client secret or certificate

As with any confidential client application, you need to add a secret or certificate to act as that application's *credentials* so it can authenticate as itself, without user interaction.

You can add credentials to your client app's registration by using the [Azure portal](#) or by using a command-line tool like [PowerShell](#).

Add client credentials by using the Azure portal

To add credentials to your confidential client application's app registration, follow the steps in [Quickstart: Register an application with the Microsoft identity platform](#) for the type of credential you want to add:

- [Add a client secret](#)
- [Add a certificate](#)

Add client credentials by using PowerShell

Alternatively, you can add credentials when you register your application with the Microsoft identity platform by using PowerShell.

The [active-directory-dotnetcore-daemon-v2](#) code sample on GitHub shows how to add an application secret or certificate when registering an application:

- For details on how to add a **client secret** with PowerShell, see [AppCreationScripts/Configure.ps1](#).
- For details on how to add a **certificate** with PowerShell, see [AppCreationScripts-withCert/Configure.ps1](#).

Next steps

Move on to the next article in this scenario, [App code configuration](#).

Daemon app that calls web APIs - code configuration

4/12/2022 • 7 minutes to read • [Edit Online](#)

Learn how to configure the code for your daemon application that calls web APIs.

Microsoft libraries supporting daemon apps

The following Microsoft libraries support daemon apps:

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIS	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
.NET	MSAL.NET	<code>Microsoft.Identity.Client</code>	Quickstart	✖	✓	GA
Java	MSAL4J	<code>msal4j</code>	—	✖	✓	GA
Node	MSAL Node	<code>msal-node</code>	Quickstart	✖	✓	GA
Python	MSAL Python	<code>msal-python</code>	Quickstart	✖	✓	GA

¹ [Supplemental terms of use for Microsoft Azure Previews](#) apply to libraries in *Public preview*.

Configure the authority

Daemon applications use application permissions rather than delegated permissions. So their supported account type can't be an account in any organizational directory or any personal Microsoft account (for example, Skype, Xbox, Outlook.com). There's no tenant admin to grant consent to a daemon application for a Microsoft personal account. You'll need to choose *accounts in my organization* or *accounts in any organization*.

The authority specified in the application configuration should be tenanted (specifying a tenant ID or a domain name associated with your organization).

Even if you want to provide a multitenant tool, you should use a tenant ID or domain name, and not `common` or `organizations` with this flow, because the service cannot reliably infer which tenant should be used.

Configure and instantiate the application

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 that runs as a service, if it's a daemon application, it needs to be a confidential client application.

Configuration file

The configuration file defines:

- The cloud instance and tenant ID, which together make up the *authority*.
 - The client ID that you got from the application registration.
 - Either a client secret or a certificate.
-
- .NET
 - Java
 - Node.js
 - Python

Here's an example of defining the configuration in an `appsettings.json` file. This example is taken from the [.NET Core console daemon](#) code sample on GitHub.

```
{  
  "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]"  
}
```

You provide either a `ClientSecret` or a `CertificateName`. These settings are exclusive.

Instantiate the MSAL application

To instantiate the MSAL application, add, reference, or import the MSAL package (depending on the language).

The construction is different, depending on whether 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
- Java
- Node.js
- Python

Add the `Microsoft.Identity.Client` NuGet package to your application, and then add a `using` directive in your code to reference it.

In MSAL.NET, the confidential client application is represented by the `IConfidentialClientApplication` interface.

```
using Microsoft.Identity.Client;  
IConfidentialClientApplication app;
```

Instantiate the confidential client application with a client secret

Here's the code to instantiate the confidential client application with a client secret:

- .NET
- Java
- Node.js
- Python

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

The `Authority` is a concatenation of the cloud instance and the tenant ID, for example

`https://login.microsoftonline.com/contoso.onmicrosoft.com` or

`https://login.microsoftonline.com/eb1ed152-0000-0000-0000-32401f3f9abd`. In the `appsettings.json` file shown in the [Configuration file](#) section, these are represented by the `Instance` and `Tenant` values, respectively.

In the code sample the previous snippet was taken from, `Authority` is a property on the [AuthenticationConfig](#) class, and is defined as such:

```
/// <summary>
/// URL of the authority
/// </summary>
public string Authority
{
    get
    {
        return String.Format(CultureInfo.InvariantCulture, Instance, Tenant);
    }
}
```

Instantiate the confidential client application with a client certificate

Here's the code to build an application with a certificate:

- [.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

```
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](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

Instead of a client secret or a certificate, the confidential client application can also prove its identity by using client assertions.

MSAL.NET has two methods to provide signed assertions to the confidential client app:

- `.WithClientAssertion()`
- `.WithClientClaims()`

When you use `WithClientAssertion`, 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 produce a signed assertion that contains the claims expected by Azure AD, plus additional client claims that you want to send. This code shows 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](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

Move on to the next article in this scenario, [Acquire a token for the app](#).

Daemon app that calls web APIs - acquire a token

4/12/2022 • 5 minutes to read • [Edit Online](#)

After you've constructed a confidential client application, you can acquire a token for the app by calling `AcquireTokenForClient`, passing the scope, and optionally forcing 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 Active Directory (Azure AD) to use the *application-level permissions* declared statically during application registration. Also, these API permissions must be granted by a tenant administrator.

- [.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

```
ResourceId = "someAppIDURI";
var scopes = new [] { ResourceId + ".default" };
```

Azure AD (v1.0) resources

The scope used for client credentials should always be the resource ID followed by `/.default`.

IMPORTANT

When MSAL requests an access token for a resource that accepts a version 1.0 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. So if, like Azure SQL Database (`https://database.windows.net`), the resource expects an audience that ends with a slash (for Azure SQL Database, `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 its equivalent, depending on the platform.

- [.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

```

using Microsoft.Identity.Client;

// With client credentials flows, the scope is always of the shape "resource/.default" because 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 doesn't have sufficient permissions.
    // - Did you declare enough app permissions during app creation?
    // - Did the tenant admin grant permissions to the application?
}
catch (MsalServiceException ex) when (ex.Message.Contains("AADSTS70011"))
{
    // Invalid scope. The scope has to be in the form "https://resourceurl/.default"
    // Mitigation: Change the scope to be as expected.
}

```

AcquireTokenForClient uses the 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 you call `AcquireTokenForClient`, because `AcquireTokenSilent` uses the *user* token cache. `AcquireTokenForClient` checks the *application* token cache itself and updates it.

Protocol

If you don't yet have a library for your chosen language, you might want to use the protocol directly:

First case: Access the token request by using 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 the token request by using 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=eyJhbGciOiJSUzI1NiIsIng1dCI6Imd40HRHeXN5amNScUtqR1BuZDdSRnd2d1pJMCJ9.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](#).

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 **Insufficient privileges to complete the operation** error when you call the API, 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 that looks like this error:

```
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>"
    }
  }
}
```

Are you calling your own API?

If you call your own web API and couldn't add an app permission to the app registration for your daemon app, did you expose an app role in your web API?

For details, see [Exposing application permissions \(app roles\)](#) and, in particular, [Ensuring that Azure AD issues tokens for your web API to only allowed clients](#).

Next steps

- [.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

Move on to the next article in this scenario, [Calling a web API](#).

Daemon app that calls web APIs - call a web API from the app

4/12/2022 • 4 minutes to read • [Edit Online](#)

.NET daemon apps can call a web API. .NET daemon apps can also call several pre-approved web APIs.

Calling a web API from a daemon application

Here's how to use the token to call an API:

- [.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

AuthenticationResult properties in MSAL.NET

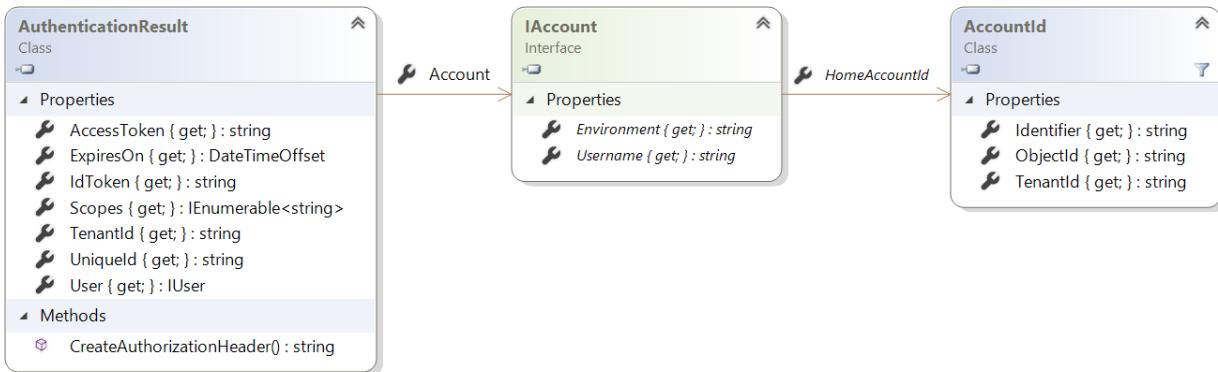
The methods to acquire tokens return `AuthenticationResult`. For async methods, `Task<AuthenticationResult>` returns.

In MSAL.NET, `AuthenticationResult` exposes:

- `AccessToken` for the web API to access resources. This parameter is a string, usually a Base-64-encoded JWT. 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. Writing code that depends on access token content on the client is one of the biggest sources of errors and client logic breaks. For more information, see [Access tokens](#).
- `IdToken` for the user. This parameter is an encoded JWT. For more information, see [ID tokens](#).
- `ExpiresOn` tells the date and time when the token expires.
- `TenantId` contains the tenant in which the user was found. For guest users in Azure Active Directory (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 an account through the `IAccount` interface. This breaking change provides the right semantics. The same user can have several accounts, in different Azure AD directories. Also, MSAL.NET provides better information in the case of guest scenarios because 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 with the properties shown in the following table.

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. They're not Base 64 encoded.

The `IAccount` interface represents information about a single account. The same user can be present in different tenants, which means that a user can have multiple accounts. Its members are shown in the following table.

PROPERTY	DESCRIPTION
<code>Username</code>	A string that contains the displayable value in UserPrincipalName (UPN) format, for example, <code>john.doe@contoso.com</code> . This string can be null, unlike <code>HomeAccountId</code> and <code>HomeAccountId.Identifier</code> , which 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 that contains 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. Here, the value is only the host.
<code>HomeAccountId</code>	The account ID of the home account for the user. This property uniquely identifies the user across Azure AD tenants.

Use the token to call a protected API

After `AuthenticationResult` is returned by MSAL in `result`, add it to the HTTP authorization header before you make the call to access the protected web API.

```
httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer",
result.AccessToken);

// Call the 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's no incremental consent with daemon apps. (There's no user interaction.) The tenant admin needs to provide consent in advance for the application and all the API permissions. If you want to call several APIs, 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](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

Move on to the next article in this scenario, [Move to production](#).

Daemon app that calls web APIs - move to production

4/12/2022 • 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 - multitenant daemon apps

If you're an ISV creating a daemon application that can run in several tenants, make sure that the tenant admin:

- 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](#).

Enable logging

To help in debugging and authentication failure troubleshooting scenarios, the Microsoft Authentication Library provides built-in logging support. Logging is each library is covered in the following articles:

- [Logging in MSAL.NET](#)
- [Logging in MSAL for Android](#)
- [Logging in MSAL.js](#)
- [Logging in MSAL for iOS/macOS](#)
- [Logging in MSAL for Java](#)
- [Logging in MSAL for Python](#)

Here are some suggestions for data collection:

- Users might ask for help when they have problems. A best practice is to capture and temporarily store logs. Provide a location where users can upload the logs. MSAL provides logging extensions to capture detailed information about authentication.
- If telemetry is available, enable it through MSAL to gather data about how users sign in to your app.

Validate your integration

Test your integration by following the [Microsoft identity platform integration checklist](#).

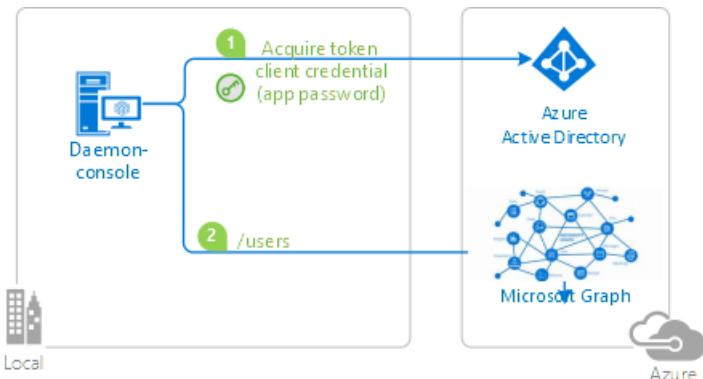
Build for resilience

Learn how to increase resiliency in your app. For details, see [Increase resilience of authentication and authorization applications you develop](#)

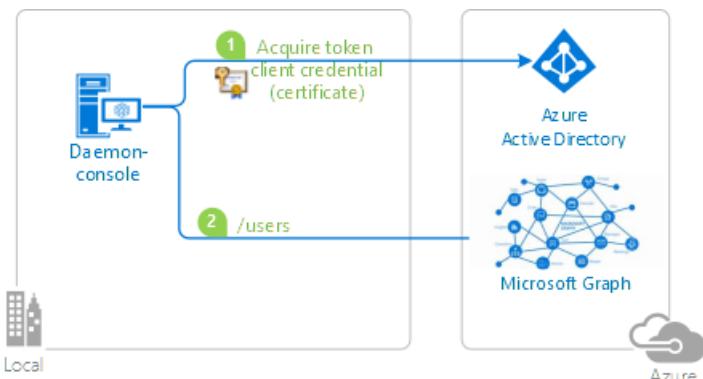
Code samples

- [.NET](#)
- [Java](#)

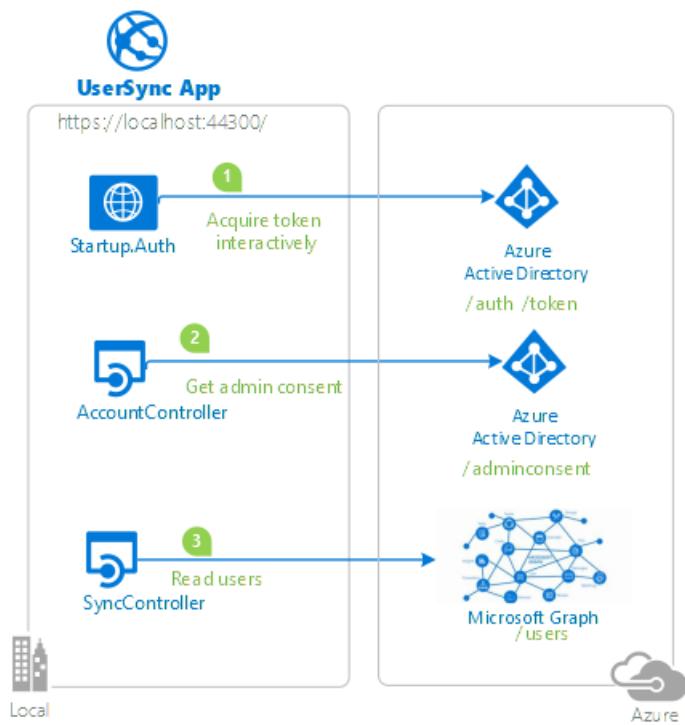
- Node.js
- Python
- Reference documentation for:
 - Instantiating [ConfidentialClientApplication](#).
 - Calling [AcquireTokenForClient](#).
- Other samples/tutorials:
 - [microsoft-identity-platform-console-daemon](#) features a small .NET Core daemon console application that displays the users of a tenant querying Microsoft Graph.



The same sample also illustrates a variation with certificates:



- [microsoft-identity-platform-aspnet-webapp-daemon](#) features an ASP.NET MVC web application that syncs data from Microsoft Graph by using the identity of the application instead of on behalf of a user. This sample also illustrates the admin consent process.



Next steps

Here are a few links to help you learn more:

- [.NET](#)
- [Java](#)
- [Node.js](#)
- [Python](#)

Try the quickstart [Acquire a token and call Microsoft Graph API from a .NET Core console app using app's identity](#).

Authentication flow support in MSAL

4/12/2022 • 11 minutes to read • [Edit Online](#)

The Microsoft Authentication Library (MSAL) supports several authorization grants and associated token flows for use by different application types and scenarios.

AUTHENTICATION FLOW	ENABLES	SUPPORTED APPLICATION TYPES
Authorization code	User sign-in and access to web APIs on behalf of the user.	* Desktop * Mobile * Single-page app (SPA) (requires PKCE) * Web
Client credentials	Access to web APIs by using the identity of the application itself. Typically used for server-to-server communication and automated scripts requiring no user interaction.	Daemon
Device code	User sign-in and access to web APIs on behalf of the user on input-constrained devices like smart TVs and IoT devices. Also used by command line interface (CLI) applications.	Desktop, Mobile
Implicit grant	User sign-in and access to web APIs on behalf of the user. <i>The implicit grant flow is no longer recommended - use authorization code with PKCE instead.</i>	* Single-page app (SPA) * Web
On-behalf-of (OBO)	Access from an "upstream" web API to a "downstream" web API on behalf of the user. The user's identity and delegated permissions are passed through to the downstream API from the upstream API.	Web API
Username/password (ROPC)	Allows an application to sign in the user by directly handling their password. <i>The ROPC flow is NOT recommended.</i>	Desktop, Mobile
Integrated Windows authentication (IWA)	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

Tokens

Your application can use one or more authentication flows. Each flow uses certain token types for authentication, authorization, and token refresh, and some also use an authorization code.

AUTHENTICATION FLOW OR ACTION	REQUIRES	ID TOKEN	ACCESS TOKEN	REFRESH TOKEN	AUTHORIZATION CODE
Authorization code flow		x	x	x	x
Client credentials			x (app-only)		
Device code flow		x	x	x	
Implicit flow		x	x		
On-behalf-of flow	access token	x	x	x	
Username/password (ROPC)	username, password	x	x	x	
Hybrid OIDC flow		x			x
Refresh token redemption	refresh token	x	x	x	

Interactive and non-interactive authentication

Several of these flows support both interactive and non-interactive token acquisition.

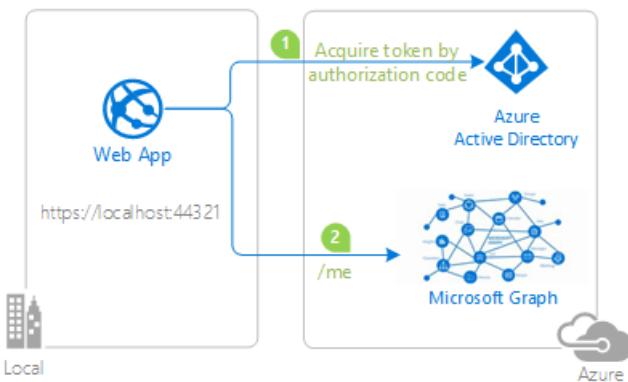
- **Interactive** - The user may be prompted for input by the authorization server. For example, to sign in, perform multi-factor authentication (MFA), or to grant consent to more resource access permissions.
- **Non-interactive** - The user may *not* be prompted for input. Also called "silent" token acquisition, the application tries to get a token by using a method in which the authorization server *may not* prompt the user for input.

Your MSAL-based application should first try to acquire a token silently and fall back to the interactive method only if the non-interactive attempt fails. For more information about this pattern, see [Acquire and cache tokens using the Microsoft Authentication Library \(MSAL\)](#).

Authorization code

The [OAuth 2.0 authorization code grant](#) can be used by web apps, single-page apps (SPA), and native (mobile and desktop) apps to gain access to protected resources like web APIs.

When users sign in to web applications, the application receives an authorization code that it can redeem for an access token to call web APIs.



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, Microsoft Graph.

Constraints for authorization code

- Single-page applications require Proof Key for Code Exchange (PKCE) when using the authorization code grant flow. PKCE is supported by MSAL.
- The OAuth 2.0 specification requires you use an authorization code to redeem an access token only *once*.

If you attempt to acquire access tokens multiple times with the same authorization code, an error similar to the following is returned by the Microsoft identity platform. Keep in mind that some libraries and frameworks request the authorization code for you automatically, and requesting a code manually in such cases will also result in this error.

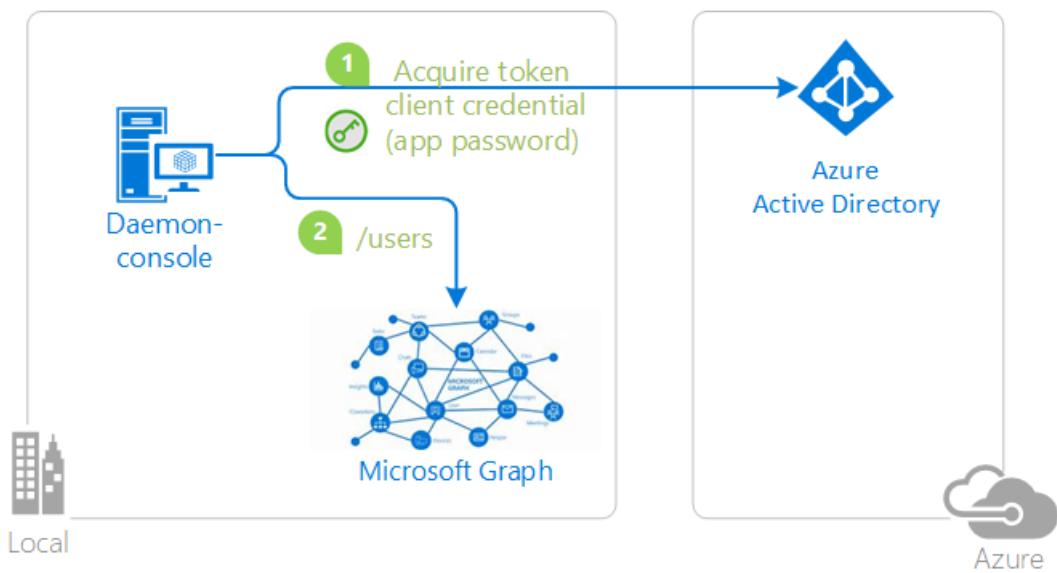
AADSTS70002: Error validating credentials. AADSTS50005: OAuth2 Authorization code was already redeemed, please retry with a new valid code or use an existing refresh token.

Client credentials

The [OAuth 2 client credentials 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.

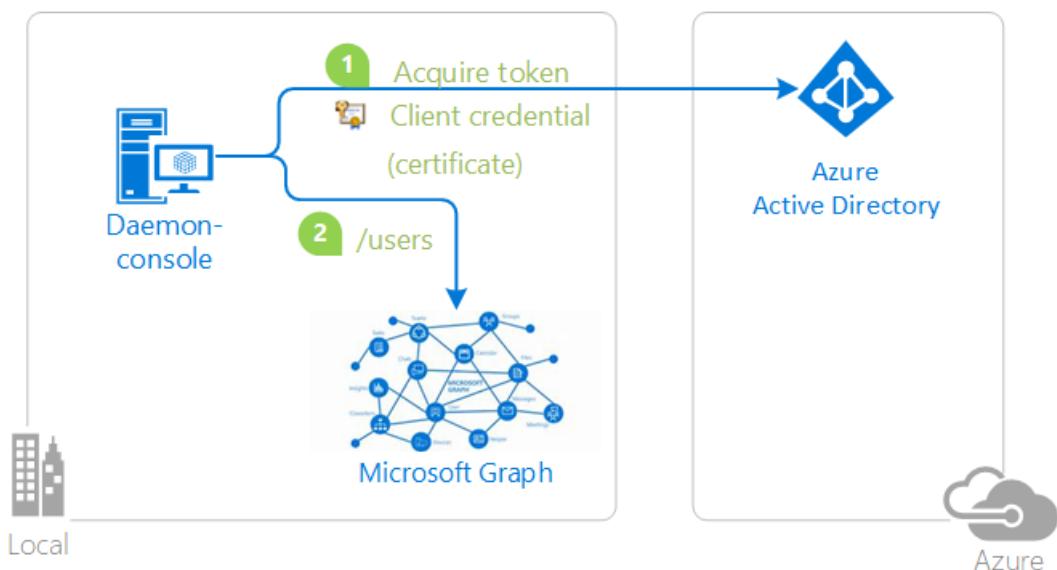
Application secrets



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 when constructing the confidential client application object in your code.

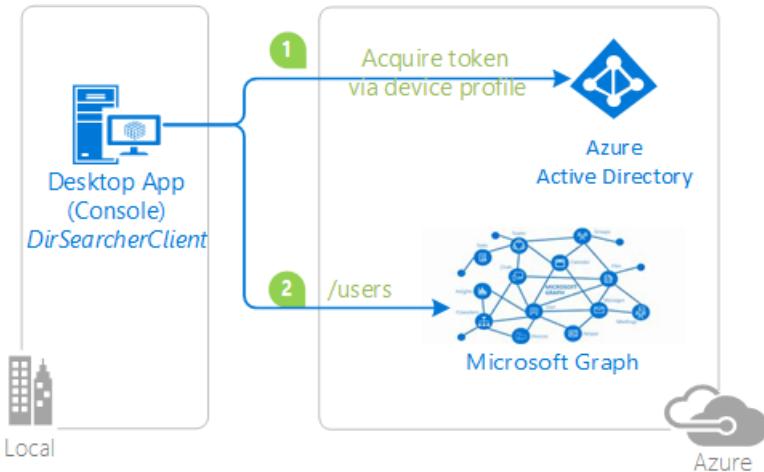
Constraints for client credentials

The confidential client flow is **unsupported** on mobile platforms like Android, iOS, or UWP. Mobile applications are considered public client applications that are incapable of guaranteeing the confidentiality of their credentials.

Device code

The [OAuth 2 device code flow](#) allows users to sign in to input-constrained devices like smart TVs, IoT devices, and printers. Interactive authentication with Azure AD requires a web browser. Where the device or operating system doesn't provide a web browser, the device code flow allows the user use another device like a computer or mobile phone to sign in interactively.

By using the device code flow, the application obtains tokens through a two-step process designed for these devices and operating systems. Examples of such applications include those running on IoT devices and command-line interface (CLI) tools.



In the preceding diagram:

1. Whenever user authentication is required, the app provides a code and asks the user to use another device like an internet-connected smartphone to visit a URL (for example, <https://microsoft.com/deviceLogin>). The user is then prompted to enter the code, and proceeding 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 for device code

- The device code flow is available only for public client applications.
- When you initialize a public client application in MSAL, use one of these authority formats:
 - Tenanted: <https://login.microsoftonline.com/{tenant}/>, where {tenant} is either the GUID representing the tenant ID or a domain name associated with the tenant.
 - Work and school accounts: <https://login.microsoftonline.com/organizations/>.

Implicit grant

The implicit grant has been replaced by the [authorization code flow with PKCE](#) as the preferred and more secure token grant flow for client-side single page-applications (SPAs). If you're building a SPA, use the authorization code flow with PKCE instead.

Single-page web apps written in JavaScript (including frameworks like Angular, Vue.js, or React.js) are downloaded from the server and their code runs directly in the browser. Because their client-side code runs in the browser and not on a web server, they have different security characteristics than traditional server-side web applications. Prior to the availability of Proof Key for Code Exchange (PKCE) for the authorization code flow, the implicit grant flow was used by SPAs for improved responsiveness and efficiency in getting access tokens.

The [OAuth 2 implicit grant flow](#) allows the app to get access tokens from the Microsoft identity platform without performing a back-end server credential exchange. The implicit grant flow allows an app to sign in the user, maintain a session, and get tokens for other web APIs from within the JavaScript code downloaded and run by the user-agent (typically a web browser).



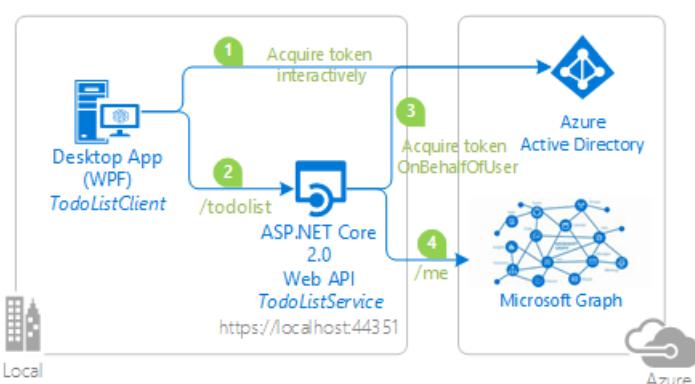
Constraints for implicit grant

The implicit grant flow doesn't include application scenarios that use cross-platform JavaScript frameworks like Electron or React Native. Cross-platform frameworks like these require further capabilities for interaction with the native desktop and mobile platforms on which they run.

Tokens issued via the implicit flow mode have a **length limitation** because they're returned to the browser by URL (where `response_mode` is either `query` or `fragment`). Some browsers limit the length of the URL in the browser bar and fail when it's too long. Thus, these implicit flow tokens don't contain `groups` or `wids` claims.

On-behalf-of (OBO)

The [OAuth 2 on-behalf-of authentication flow](#) flow is used when an application invokes a service or web API that 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).

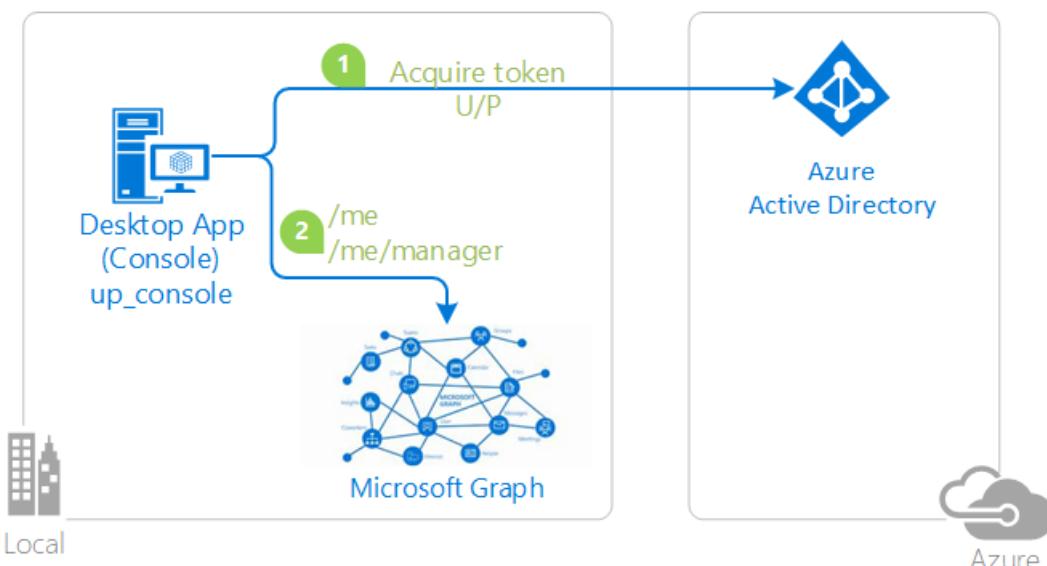
Username/password (ROPC)

WARNING

The resource owner password credentials (ROPC) flow is NOT recommended. ROPC requires a high degree of trust and credential exposure. *Resort to using ROPC only if a more secure flow can't be used.* For more information, see [What's the solution to the growing problem of passwords?](#)

The [OAuth 2 resource owner password credentials](#) (ROPC) grant 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.

Some application scenarios like DevOps might find ROPC useful, but you should avoid it in any application in which you provide an interactive UI for user sign-in.



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.

To acquire a token silently on Windows domain-joined machines, we recommend [integrated Windows authentication \(IWA\)](#) instead of ROPC. For other scenarios, use the [device code flow](#).

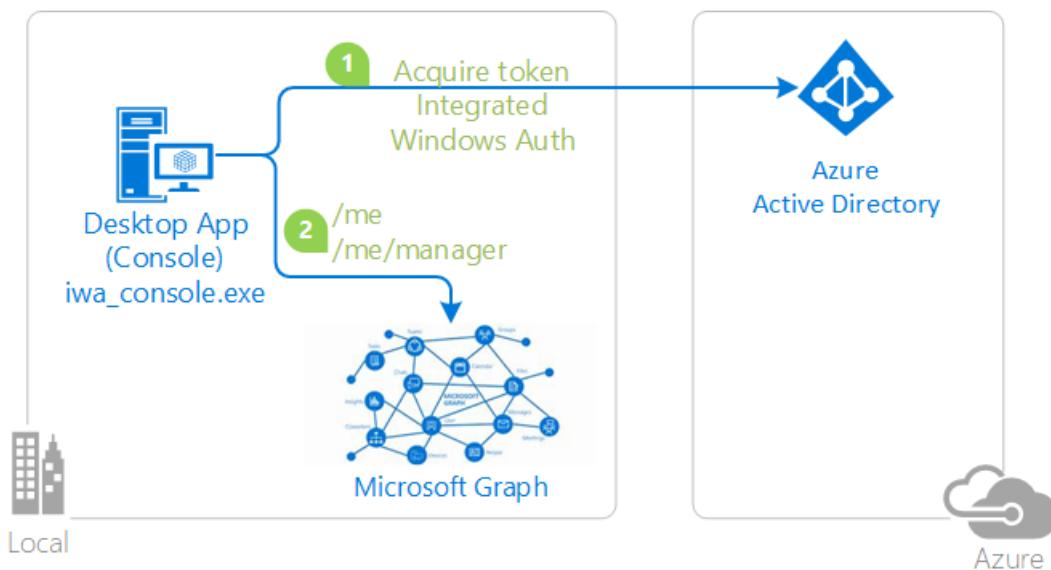
Constraints for ROPC

The following constraints apply to the applications using the ROPC flow:

- Single sign-on is **unsupported**.
- Multi-factor authentication (MFA) is **unsupported**.
 - Check with your tenant admin before using this flow - MFA is a commonly used feature.
- Conditional Access is **unsupported**.
- ROPC works *only* for work and school accounts.
- Personal Microsoft accounts (MSA) are **unsupported** by ROPC.
- ROPC is **supported** in .NET desktop and .NET Core applications.
- ROPC is **unsupported** in Universal Windows Platform (UWP) applications.
- ROPC in Azure AD B2C is supported *only* for local accounts.
 - For information about ROPC in MSAL.NET and Azure AD B2C, see [Using ROPC with Azure AD B2C](#).

Integrated Windows authentication (IWA)

MSAL supports integrated Windows authentication (IWA) for desktop and mobile applications that run on domain-joined or Azure AD-joined Windows computers. By using IWA, these applications acquire a token silently without requiring UI interaction by 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 for IWA

Compatibility

Integrated Windows authentication (IWA) is enabled for .NET desktop, .NET Core, and Windows Universal Platform apps.

IWA supports AD FS-federated users *only* - 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.

Multi-factor authentication (MFA)

IWA's non-interactive (silent) authentication can fail if MFA is enabled in the Azure AD tenant and an MFA challenge is issued by Azure AD. If IWA fails, you should fall back to an [interactive method of authentication](#) as described earlier.

Azure AD uses AI to determine when two-factor authentication is required. Two-factor authentication is typically required when a user signs in from a different country/region, when connected to a corporate network without using a VPN, and sometimes when they *are* connected through a VPN. Because MFA's configuration and challenge frequency may be outside of your control as the developer, your application should gracefully handle a failure of IWA's silent token acquisition.

Authority URI restrictions

The authority passed in when constructing the public client application must be one of:

- `https://login.microsoftonline.com/{tenant}/` - This authority indicates a single-tenant application whose sign-in audience is restricted to the users in the specified Azure AD tenant. The `{tenant}` value can be the tenant ID in GUID form or the domain name associated with the tenant.
- `https://login.microsoftonline.com/organizations/` - This authority indicates a multi-tenant application whose sign-in audience is users in any Azure AD tenant.

Authority values must NOT contain `/common` or `/consumers` because personal Microsoft accounts (MSA) are unsupported by IWA.

Consent requirements

Because IWA 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.

To satisfy either requirement, one of these operations must have been completed:

- You as the application developer have selected **Grant** in the Azure portal for yourself.
- A tenant admin has selected **Grant/revoke admin consent for {tenant domain}** in the **API permissions** tab of the app registration in the Azure portal; see [Add permissions to access your web API](#).
- You've provided a way for users to consent to the application; see [Requesting individual user consent](#).
- You've provided a way for the tenant admin to consent for the application; see [admin consent](#).

For more information on consent, see [Permissions and consent](#).

Next steps

Now that you've reviewed the authentication flows supported by MSAL, learn about acquiring and caching the tokens used in these flows:

[Acquire and cache tokens using the Microsoft Authentication Library \(MSAL\)](#)

Acquire and cache tokens using the Microsoft Authentication Library (MSAL)

4/12/2022 • 8 minutes to read • [Edit Online](#)

Access tokens enable clients to securely call web APIs protected by Azure. There are several ways to acquire a token by using the Microsoft Authentication Library (MSAL). Some require user interaction through a web browser, while others don't require user interaction. In general, the method used for acquiring a token depends on whether the application is a public client application like desktop or mobile app, or a confidential client application like web app, web API, or daemon application.

MSAL caches a token after it's been acquired. Your application code should first try to get a token silently from the cache before attempting to acquire a token by other means.

You can also clear the token cache, which is achieved by removing the accounts from the cache. This doesn't remove the session cookie that's in the browser, however.

Scopes when acquiring tokens

Scopes are the permissions that a web API exposes that client applications can 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 the Microsoft identity platform 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.

Several of MSAL's token acquisition methods require a `scopes` parameter. The `scopes` parameter is a list of strings that declare the desired permissions and the resources requested. Well-known scopes are the [Microsoft Graph permissions](#).

It's also possible in MSAL to access v1.0 resources. For more information, see [Scopes for a v1.0 application](#).

Request scopes for a web API

When your application needs to request an access token with specific permissions for a resource API, pass the scopes containing the app ID URI of the API in the format `<app ID URI>/<scope>`.

Some example scope values for different resources:

- Microsoft Graph API: `https://graph.microsoft.com/User.Read`
- Custom web API: `api://11111111-1111-1111-1111-111111111111/api.read`

The format of the scope value varies depending on the resource (the API) receiving the access token and the `aud` claim values it accepts.

For Microsoft Graph only, the `user.read` scope maps to `https://graph.microsoft.com/User.Read`, and both scope formats can be used interchangeably.

Certain web APIs such as the Azure Resource Manager API (<https://management.core.windows.net/>) expect a trailing forward slash ('/') in the audience claim (`aud`) of the access token. In this case, pass the scope as `https://management.core.windows.net//user_impersonation`, including the double forward slash ('//').

Other APIs might require that *no scheme or host* is included in the scope value, and expect only the app ID (a GUID) and the scope name, for example:

TIP

If the downstream resource is not under your control, you might need to try different scope value formats (for example with/without scheme and host) if you receive `401` or other errors when passing the access token to the resource.

Request dynamic scopes for incremental consent

As the features provided by your application or its requirements change, you can request additional permissions as needed by using the scope parameter. Such *dynamic scopes* allow your users to provide incremental consent to scopes.

For example, you might sign in the user but initially deny them access to any resources. Later, you can give them the ability to view their calendar by requesting the calendar scope in the acquire token method and obtaining the user's consent to do so. For example, by requesting the `https://graph.microsoft.com/User.Read` and `https://graph.microsoft.com/Calendar.Read` scopes.

Acquiring tokens silently (from the cache)

MSAL maintains a token cache (or two caches for confidential client applications) and caches a token after it's 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 first try to get a token silently from the cache. If the method call returns a "UI required" error or exception, try acquiring a token by other means.

There are two flows, however, in 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 the application token cache before sending a request to the security token service (STS).
- [Authorization code flow](#) in web apps, as it redeems a code that the application obtained by signing in the user and having them consent to more scopes. Since a code and not an account is passed as a parameter, the method can't look in the cache before redeeming the code, which invokes 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

In public client applications like desktop and mobile apps, you can:

- Get tokens interactively by having the user sign in through a UI or pop-up window.
- 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.

- Get a token with a [username and password](#) in .NET framework desktop client applications (not recommended). Do not use username/password in confidential client applications.
- Get a token through the [device code flow](#) in applications running on devices that 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 a daemon application like a Windows service), you:

- Acquire tokens **for the application itself** and not for a user, using the [client credentials flow](#). This technique can be used for syncing tools, or tools that process users in general and not a specific user.
- Use the [on-behalf-of \(OBO\) 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 that includes 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 string is 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 most common sources of errors and client logic breakage.
- The [ID token](#) for the user (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.

(Advanced) Accessing the user's cached tokens in background apps and services

You can use MSAL's token cache implementation to allow background apps, APIs, and services to use the access token cache to continue to act on behalf of users in their absence. Doing so is especially useful if the background apps and services need to continue to work on behalf of the user after the user has exited the front-end web app.

Today, most background processes use [application permissions](#) when they need to work with a user's data without them being present to authenticate or reauthenticate. Because application permissions often require admin consent, which requires elevation of privilege, unnecessary friction is encountered as the developer didn't intend to obtain permission beyond that which the user originally consented to for their app.

This code sample on GitHub shows how to avoid this unneeded friction by accessing MSAL's token cache from

background apps:

[Accessing the logged-in user's token cache from background apps, APIs, and services](#)

Next steps

Several of the platforms supported by MSAL have additional token cache-related information in the documentation for that platform's library. For example:

- [Get a token from the token cache using MSAL.NET](#)
- [Single sign-on with MSAL.js](#)
- [Custom token cache serialization in MSAL for Python](#)
- [Custom token cache serialization in MSAL for Java](#)

Public client and confidential client applications

4/12/2022 • 2 minutes to read • [Edit Online](#)

The 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.

- **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 can keep 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:



In MSAL.js, there's 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 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](#).

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 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

For more information about application configuration and instantiating, see:

- [Client application configuration options](#)
- [Instantiating client applications by using MSAL.NET](#)
- [Instantiating client applications by using MSAL.js](#)

Use MSAL in a national cloud environment

4/12/2022 • 5 minutes to read • [Edit Online](#)

National clouds, also known as Sovereign 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 Azure cloud, Azure ActiveDirectory (Azure AD) is deployed in the following national clouds:

- Azure Government
- Azure China 21Vianet
- Azure Germany ([Closing on October 29, 2021](#))

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.

Azure Germany (Microsoft Cloud Deutschland)

WARNING

Azure Germany (Microsoft Cloud Deutschland) will be [closed on October 29, 2021](#). Services and applications you choose *not* to migrate to a region in global Azure before that date will become inaccessible.

If you haven't migrated your application from Azure Germany, follow [Azure Active Directory information for the migration from Azure Germany](#) to get started.

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, 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 Microsoft 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.

For details about using a national cloud with a particular programming language, choose the tab matching your language:

- [.NET](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)
- [Objective-C](#)
- [Swift](#)

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: [Build an ASP.NET Core Web app signing-in users in sovereign clouds with the Microsoft identity platform](#).
- To call the Microsoft Graph API, follow this tutorial: [Using the Microsoft identity platform to call the Microsoft Graph API from an ASP.NET Core 2.x web app, on behalf of a user signing-in using their work and school account in Microsoft National Cloud](#).

Next steps

See [National cloud authentication endpoints](#) for a list of the Azure portal URLs and token endpoints for each cloud.

National cloud documentation:

- [Azure Government](#)
- [Azure China 21Vianet](#)
- [Azure Germany \(closes on October 29, 2021\)](#)

Microsoft Authentication Library for iOS and macOS differences

4/12/2022 • 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.

Using redirect URIs with the Microsoft authentication library for iOS and macOS

4/12/2022 • 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: <code><scheme>://[Your_Bundle_Id]</code>, where <code>scheme</code> is a unique string. This format will continue to work when you use MSAL.

- <code><scheme></code> must be registered in your app's Info.plist under <code>CFBundleURLTypes > CFBundlURLSchemes</code>. 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 <code><scheme></code> must be <code>https</code> and doesn't need to be declared in <code>CFBundleURLSchemes</code>. Instead, configure the app and domain per Apple's instructions at [Universal Links for Developers](#) and call the <code>handleMSALResponse:</code> method of <code>MSALPublicClientApplication</code> when your application is opened through a universal link.

Use a custom redirect URI

To use a custom redirect URI, pass the <code>redirectUri</code> parameter to <code>MSALPublicClientApplicationConfig</code> and pass that object to <code>MSALPublicClientApplication</code> when you initialize the object. If the redirect URI is invalid, the initializer will return <code>nil</code> and set the <code>redirectURIError</code> 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 <code>handleMSALResponse:</code> method of <code>MSALPublicClientApplication</code> 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](#)

Customize browsers and WebViews for iOS/macOS

4/12/2022 • 3 minutes to read • [Edit Online](#)

A web browser is required for interactive authentication. On iOS and macOS 10.15+, 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:

- [SFAuthenticationSession](#)
- [SFSafariViewController](#)

For iOS and macOS:

- [ASWebAuthenticationSession](#)
- [WKWebView](#).

MSAL for macOS only supports `WKWebView` on older OS versions. `ASWebAuthenticationSession` is only supported on macOS 10.15 and above.

System browsers

For iOS, `ASWebAuthenticationSession`, `SFAuthenticationSession`, and `SFSafariViewController` are considered system browsers. For macOS, only `ASWebAuthenticationSession` is available. 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`.

Default configuration for iOS

VERSION	WEB BROWSER
iOS 12+	<code>ASWebAuthenticationSession</code>
iOS 11	<code>SFAuthenticationSession</code>
iOS 10	<code>SFSafariViewController</code>

Default configuration for macOS

VERSION	WEB BROWSER
macOS 10.15+	<code>ASWebAuthenticationSession</code>
other versions	<code>WKWebView</code>

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
<code>ASWebAuthenticationSession</code>	System	iOS12 and up	macOS 10.15 and up	Yes	iOS and macOS 10.15+	w/ Safari instances
<code>SFAuthenticationSession</code>	System	iOS11 and up	N/A	Yes	iOS only	w/ Safari instances
<code>SFSafariViewController</code>	System	iOS11 and up	N/A	No	iOS only	No**
<code>SFSafariViewVieController</code>	System	iOS10	N/A	Yes	iOS only	w/ Safari instances
<code>WKWebView</code>	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]
initWithAuthPresentationViewController: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(authPresentationViewController: 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)
{
    /**
     For iOS 11 and up, uses AuthenticationSession (ASWebAuthenticationSession or SFAuthenticationSession).
     For older versions, with AuthenticationSession not being available, uses SafariViewController.
     For macOS 10.15 and above uses ASWebAuthenticationSession
     For older macOS versions uses WKWebView
    */
    MSALWebviewTypeDefault,

    /**
     Use ASWebAuthenticationSession where available.
     On older iOS versions uses SFAuthenticationSession
     Doesn't allow any other webview type, so if either of these are not present, fails the request*/
    MSALWebviewTypeAuthenticationSession,
};

#if TARGET_OS_IPHONE

    /**
     Use SFSafariViewController for all versions.*/
    MSALWebviewTypeSafariViewController,
#endif

    /**
     Use WKWebView */
    MSALWebviewTypeWKWebView,
};
```

Next steps

Learn more about [Authentication flows and application scenarios](#)

Shared device mode for iOS devices

4/12/2022 • 6 minutes to read • [Edit Online](#)

IMPORTANT

This feature is in public preview. This preview is provided without a service-level agreement and isn't recommended for production workloads. Some features might be unsupported or have constrained capabilities. For more information, see [Supplemental terms of use for Microsoft Azure previews](#).

Frontline workers such as retail associates, flight crew members, and field service workers often use a shared mobile device to perform their work. These shared devices can present security risks if your users share their passwords or PINs, intentionally or not, to access customer and business data on the shared device.

Shared device mode allows you to configure an iOS 13 or higher device to be more easily and securely shared by employees. Employees can sign in and access customer information quickly. When they're finished with their shift or task, they can sign out of the device and it's immediately ready for use by the next employee.

Shared device mode also provides Microsoft identity-backed management of the device.

This feature uses the [Microsoft Authenticator app](#) to manage the users on the device and to distribute the [Microsoft Enterprise SSO plug-in for Apple devices](#).

Create a shared device mode app

To create a shared device mode app, developers and cloud device admins work together:

1. **Application developers** write a single-account app (multiple-account apps are not supported in shared device mode) and write code to handle things like shared device sign-out.
2. **Device administrators** prepare the device to be shared by using a mobile device management (MDM) provider like Microsoft Intune to manage the devices in their organization. The MDM pushes the Microsoft Authenticator app to the devices and turns on "Shared Mode" for each device through a profile update to the device. This Shared Mode setting is what changes the behavior of the supported apps on the device. This configuration from the MDM provider sets the shared device mode for the device and enables the [Microsoft Enterprise SSO plug-in for Apple devices](#) which is required for shared device mode.
3. **[Required during Public Preview only]** A user with [Cloud Device Administrator](#) role must then launch the [Microsoft Authenticator app](#) and join their device to the organization.

To configure the membership of your organizational roles in the Azure portal: **Azure Active Directory** > **Roles and Administrators** > **Cloud Device Administrator**

The following sections help you update your application to support shared device mode.

Use Intune to enable shared device mode & SSO extension

NOTE

The following step is required only during public preview.

Your device needs to be configured to support shared device mode. It must have iOS 13+ installed and be

MDM-enrolled. MDM configuration also needs to enable [Microsoft Enterprise SSO plug-in for Apple devices](#). To learn more about SSO extensions, see the [Apple video](#).

1. In the Intune Configuration Portal, tell the device to enable the [Microsoft Enterprise SSO plug-in for Apple devices](#) with the following configuration:

- **Type:** Redirect
- **Extension ID:** com.microsoft.azureauthenticator.ssoextension
- **Team ID:** (this field is not needed for iOS)
- **URLs:**
 - <https://login.microsoftonline.com>
 - <https://login.microsoft.com>
 - <https://sts.windows.net>
 - <https://login.partner.microsoftonline.cn>
 - <https://login.chinacloudapi.cn>
 - <https://login.microsoftonline.de>
 - <https://login.microsoftonline.us>
 - <https://login.usgovcloudapi.net>
 - <https://login-us.microsoftonline.com>
- **Additional Data to configure:**
 - Key: sharedDeviceMode
 - Type: Boolean
 - Value: true

For more information about configuring with Intune, see the [Intune configuration documentation](#).

2. Next, configure your MDM to push the Microsoft Authenticator app to your device through an MDM profile.

Set the following configuration options to turn on Shared Device mode:

- Configuration 1:
 - Key: sharedDeviceMode
 - Type: Boolean
 - Value: true

Modify your iOS application to support shared device mode

Your users depend on you to ensure their data isn't leaked to another user. The following sections provide helpful signals to indicate to your application that a change has occurred and should be handled.

You are responsible for checking the state of the user on the device every time your app is used, and then clearing the previous user's data. This includes if it is reloaded from the background in multi-tasking.

On a user change, you should ensure both the previous user's data is cleared and that any cached data being displayed in your application is removed. We highly recommend you and your company conduct a security review process after updating your app to support shared device mode.

Detect shared device mode

Detecting shared device mode is important for your application. Many applications will require a change in their user experience (UX) when the application is used on a shared device. For example, your application might have a "Sign-Up" feature, which isn't appropriate for a frontline worker because they likely already have an account. You may also want to add extra security to your application's handling of data if it's in shared device mode.

Use the `getDeviceInformationWithParameters:completionBlock:` API in the `MSALPublicClientApplication` to determine if an app is running on a device in shared device mode.

The following code snippets show examples of using the `getDeviceInformationWithParameters:completionBlock:` API.

Swift

```
application.getDeviceInformation(with: nil, completionBlock: { (deviceInformation, error) in

    guard let deviceInfo = deviceInformation else {
        return
    }

    let isSharedDevice = deviceInfo.deviceMode == .shared
    // Change your app UX if needed
})
```

Objective-C

```
[application getDeviceInformationWithParameters:nil
                                         completionBlock:^(MSALDeviceInformation * _Nullable deviceInformation,
NSError * _Nullable error)
{
    if (!deviceInformation)
    {
        return;
    }

    BOOL isSharedDevice = deviceInformation.deviceMode == MSALDeviceModeShared;
    // Change your app UX if needed
}];
```

Get the signed-in user and determine if a user has changed on the device

Another important part of supporting shared device mode is determining the state of the user on the device and clearing application data if a user has changed or if there is no user at all on the device. You are responsible for ensuring data isn't leaked to another user.

You can use `getCurrentAccountWithParameters:completionBlock:` API to query the currently signed-in account on the device.

Swift

```
let msalParameters = MSALParameters()
msalParameters.completionBlockQueue = DispatchQueue.main

application.getCurrentAccount(with: msalParameters, completionBlock: { (currentAccount, previousAccount,
error) in

    // currentAccount is the currently signed in account
    // previousAccount is the previously signed in account if any
})
```

Objective-C

```
MSALParameters *parameters = [MSALParameters new];
parameters.completionBlockQueue = dispatch_get_main_queue();

[application getCurrentAccountWithParameters:parameters
    completionBlock:^(MSALAccount * _Nullable account, MSALAccount * _Nullable previousAccount, NSError * _Nullable error)
{
    // currentAccount is the currently signed in account
    // previousAccount is the previously signed in account if any
}];
```

Globally sign in a user

When a device is configured as a shared device, your application can call the

`acquireTokenWithParameters:completionBlock:` API to sign in the account. The account will be available globally for all eligible apps on the device after the first app signs in the account.

Objective-C

```
MSALInteractiveTokenParameters *parameters = [[MSALInteractiveTokenParameters alloc]
initWithScopes:@[@"api://myapi/scope"] webViewParameters:[self msalTestWebViewParameters]];

parameters.loginHint = self.loginHintTextField.text;

[application acquireTokenWithParameters:parameters completionBlock:completionBlock];
```

Globally sign out a user

The following code removes the signed-in account and clears cached tokens from not only the app, but also from the device that's in shared device mode. It does not, however, clear the *data* from your application. You must clear the data from your application, as well as clear any cached data your application may be displaying to the user.

Clear browser state

NOTE

The following step is required only during public preview.

In this public preview version, the [Microsoft Enterprise SSO plug-in for Apple devices](#) clears state only for applications. It does not clear state on the Safari browser. We recommend you manually clear browser session to ensure no traces of user state are left behind. You can use the optional `signoutFromBrowser` property shown below to clear any cookies. This will cause the browser to briefly launch on the device.

Swift

```

let account = .... /* account retrieved above */

let signoutParameters = MSALSignoutParameters(webviewParameters: self.webViewParamaters!)
signoutParameters.signoutFromBrowser = true // Only needed for Public Preview.

application.signout(with: account, signoutParameters: signoutParameters, completionBlock: {(success, error)
in

    if let error = error {
        // Signout failed
        return
    }

    // Sign out completed successfully
})

```

Objective-C

```

MSALAccount *account = ... /* account retrieved above */;

MSALSignoutParameters *signoutParameters = [[MSALSignoutParameters alloc]
initWithWebviewParameters:webViewParameters];
signoutParameters.signoutFromBrowser = YES; // Only needed for Public Preview.

[application signoutWithAccount:account signoutParameters:signoutParameters completionBlock:^(BOOL success,
NSError * _Nullable error)
{
    if (!success)
    {
        // Signout failed
        return;
    }

    // Sign out completed successfully
}];

```

Next steps

To see shared device mode in action, the following code sample on GitHub includes an example of running a frontline worker app on an iOS device in shared device mode:

[MSAL iOS Swift Microsoft Graph API Sample](#)

Configure MSAL for iOS and macOS to use different identity providers

4/12/2022 • 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.

Signing 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;
}

```

```

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](#)

Configure keychain

4/12/2022 • 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](#)
- [Swift](#)

```
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
```

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](#)
- [Swift](#)

```
config.cacheConfig.keychainSharingGroup = [[NSBundle mainBundle] 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.

Configure SSO on macOS and iOS

4/12/2022 • 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`
 App2 Redirect URI: `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.

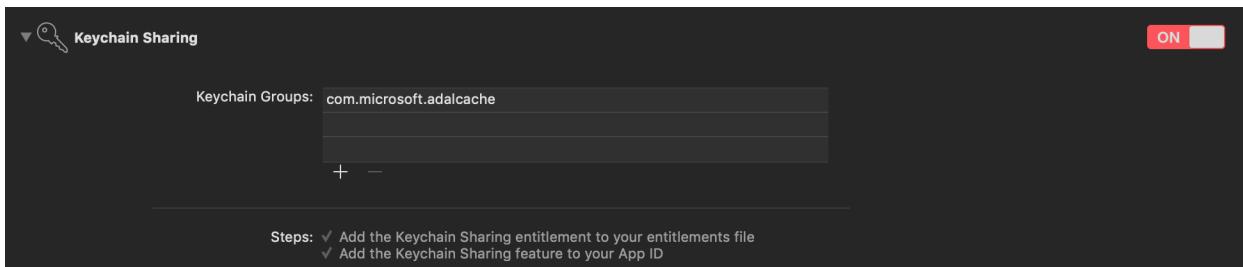
When you have the entitlements set up correctly, you'll see a `entitlements.plist` file in your project directory that contains something like this example:

```
<?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>
```

Add a new keychain group

Add a new keychain group to your project **Capabilities**. The keychain group should be:

- `com.microsoft.adalcache` on iOS
- `com.microsoft.identity.universalstorage` on macOS.



For more information, see [keychain groups](#).

Configure the application object

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

4/12/2022 • 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, 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 accountByUsername:@"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 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 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](#)

Request custom claims using MSAL for iOS and macOS

4/12/2022 • 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\":\"values\":[\"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](#)

Logging in MSAL for iOS/macOS

4/12/2022 • 4 minutes to read • [Edit Online](#)

The 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. Used 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.

The following sections provide more details about MSAL error logging for your application.

- [Objective-C](#)
- [Swift](#)

MSAL for iOS and macOS logging-ObjC

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:

```
[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
    }
}];
```

Personal data

By default, MSAL doesn't capture or log any personal data. 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.

```
// 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;
```

Logging levels

To set the logging level when you log using MSAL for iOS and macOS, use one of the following values:

LEVEL	DESCRIPTION
<code>MSALLogLevelNothing</code>	Disable all logging
<code>MSALLogLevelError</code>	Default level, prints out information only when errors occur
<code>MSALLogLevelWarning</code>	Warnings
<code>MSALLogLevelInfo</code>	Library entry points, with parameters and various keychain operations
<code>MSALLogLevelVerbose</code>	API tracing

For example:

```
MSALGlobalConfig.loggerConfig.logLevel = MSALLogLevelVerbose;
```

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.

Next steps

For more code samples, refer to [Microsoft identity platform code samples](#).

Handle errors and exceptions in MSAL for iOS/macOS

4/12/2022 • 5 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, 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 [Azure AD Authentication and authorization error codes](#).

During the sign-in experience, you may encounter errors about consents, Conditional Access (MFA, Device Management, Location-based restrictions), token issuance and redemption, and user properties.

The following section provides more details about error handling for your app.

Error handling in MSAL for iOS/macOS

The complete list of MSAL for iOS and macOS 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 `NSURLErrorDomain` 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.

```

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(@"Failed with error %@", error);

        break;
    }

    default:
        NSLog(@"Failed with unknown MSAL error %@", error);

        break;
}

return;
}

// Handle no internet connection.
if ([error.domain isEqualToString:NSURLErrorDomain] && error.code ==
NSURLErrorNotConnectedToInternet)
{
    NSLog(@"No internet connection.");
    return;
}

// Other errors may require trying again later,
// or reporting authentication problems to the user.
NSLog(@"Failed with error %@", error);
};

// Acquire token silently
[application acquireTokenSilentWithParameters:silentParameters
    completionBlock:completionBlock];

// or acquire it interactively.
[application acquireTokenWithParameters:interactiveParameters
    completionBlock:completionBlock];

```

```

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 == MSALErrorDomain, 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)\n");

        default:
            print("Failed with unknown MSAL error \(\error)\n")
    }
}

// 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)\n");
}

// Acquire token silently
application.acquireToken(with: interactiveParameters, completionBlock: completionBlock)

// or acquire it interactively.
application.acquireTokenSilent(with: silentParameters, completionBlock: completionBlock)

```

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. This 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.

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 Azure AD service, and occasionally failures can occur. For example the network can go down or the server is overloaded.

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.

Next steps

Consider enabling [Logging in MSAL for iOS/macOS](#) to help you diagnose and debug issues.

Troubleshoot MSAL for iOS and macOS TLS/SSL issues

4/12/2022 • 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 TLS check, refer to `NSURLErrorFailingURLErrorKey` in the `userInfo` dictionary of the error object.

This error is from Apple's networking library. A full list of NSURL error codes is in `NSURLError.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 an SSL validation service such as [SSL Server Test](#). 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 TLS/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.

Migrate applications to MSAL for iOS and macOS

4/12/2022 • 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 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?](#).

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

`ADAAuthenticationContext` is the first object an ADAL app creates. It represents an instantiation of ADAL. Apps create a new instance of `ADAAuthenticationContext` for each Azure Active Directory cloud and tenant (authority) combination. The same `ADAAuthenticationContext` 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 apps 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 {
            assert(true, "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](#)

Custom token cache serialization in MSAL for Java

4/12/2022 • 2 minutes to read • [Edit Online](#)

To persist the token cache between instances of your 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

4/12/2022 • 2 minutes to read • [Edit Online](#)

MSAL for Java 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

`PublicClientApplication.removeAccount()` as shown in the following example:

```
Set<IAccount> accounts = pca.getAccounts().join();

IAccount accountToBeRemoved = accounts.stream().filter(
    x -> x.getUsername().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](#).

ADAL to MSAL migration guide for Java

4/12/2022 • 5 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 the 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, 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 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.

You can learn more about MSAL and get started with an [overview of the Microsoft Authentication Library](#).

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?](#).

Scopes not resources

ADAL4J acquires tokens for resources whereas MSAL for Java acquires tokens for scopes. A number of MSAL for Java 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.

You can add the `/.default` scope suffix to the resource to help migrate your apps from the ADAL to MSAL. For example, for the resource value of `https://graph.microsoft.com`, the equivalent scope value is `https://graph.microsoft.com/.default`. If the resource is not in the URL form, but a resource ID of the form `xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxx`, you can still use the scope value as `xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxx/.default`.

For more details about the different types of scopes, refer [Permissions and consent in the Microsoft identity platform](#) and the [Scopes for a Web API accepting v1.0 tokens](#) articles.

Core classes

In ADAL4J, the `AuthenticationContext` class represents your connection to the Security Token Service (STS), or authorization server, through an Authority. However, MSAL for Java 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 MSAL for Java functions:

ADAL4J METHOD	MSAL4J METHOD
acquireToken(String resource, ClientCredential credential, AuthenticationCallback callback)	acquireToken(ClientCredentialParameters)
acquireToken(String resource, ClientAssertion assertion, AuthenticationCallback callback)	acquireToken(ClientCredentialParameters)
acquireToken(String resource, AsymmetricKeyCredential credential, AuthenticationCallback callback)	acquireToken(ClientCredentialParameters)
acquireToken(String resource, String clientId, String username, String password, AuthenticationCallback callback)	acquireToken(UserNamePasswordParameters)
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.

MSAL for Java 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. MSAL for Java provides better information in guest scenarios because home account information is provided.

Cache persistence

ADAL4J did not have support for token cache. MSAL for Java 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 MSAL for Java, if you want to restrict login to any AAD account, 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.

MSAL for Java does not expose refresh tokens for security reasons. Instead, MSAL handles refreshing tokens for you.

MSAL for Java 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);
```

Active Directory Federation Services support in MSAL for Java

4/12/2022 • 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 your Microsoft Authentication Library for Java (MSAL for Java) app. Once integrated, your app can authenticate users in AD FS, federated through Azure AD. For more information about scenarios, see [AD FS Scenarios for Developers](#).

An app that uses MSAL for Java will talk to Azure Active Directory (Azure AD), which then federates to AD FS.

MSAL for Java connects to Azure AD, which signs in users that are managed in Azure AD (managed users) or users managed by another identity provider such as AD FS (federated users). MSAL for Java doesn't know that a user is federated. It simply talks to Azure AD.

The [authority](#) you use in this case is the usual authority (authority host name + tenant, common, or organizations).

Acquire a token interactively for a federated user

When you call `ConfidentialClientApplication.AcquireToken()` or `PublicClientApplication.AcquireToken()` with `AuthorizationCodeParameters` or `DeviceCodeParameters`, the user experience is typically:

1. The user enters their account ID.
2. Azure AD briefly displays "Taking you to your organization's page", and 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.

The supported AD FS versions in this federated scenario are:

- Active Directory Federation Services FS v2
- Active Directory Federation Services v3 (Windows Server 2012 R2)
- Active Directory Federation Services v4 (AD FS 2016)

Acquire a token via username and password

When you acquire a token using `ConfidentialClientApplication.AcquireToken()` or `PublicClientApplication.AcquireToken()` with `IntegratedWindowsAuthenticationParameters` or `UsernamePasswordParameters`, MSAL for Java gets the identity provider to contact based on the username. MSAL for Java gets a [SAML 1.1 token](#) token from the identity provider, which it then provides to Azure AD which returns the JSON Web Token (JWT).

Next steps

For the federated case, see [Configure Azure Active Directory sign in behavior for an application by using a Home Realm Discovery policy](#)

Handle errors and exceptions in MSAL for Java

4/12/2022 • 4 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, 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 [Azure AD Authentication and authorization error codes](#).

During the sign-in experience, you may encounter errors about consents, Conditional Access (MFA, Device Management, Location-based restrictions), token issuance and redemption, and user properties.

The following section provides more details about error handling for your app.

Error handling in MSAL for Java

In MSAL for Java, there are three types of exceptions: `MsalClientException`, `MsalServiceException`, and `MsalInteractionRequiredException`; all which inherit from `MsalException`.

- `MsalClientException` is thrown when an error occurs that is local to the library or device.
- `MsalServiceException` is thrown when the secure token service (STS) returns an error response or another networking error occurs.
- `MsalInteractionRequiredException` is thrown when UI interaction is required for authentication to succeed.

MsalServiceException

`MsalServiceException` exposes HTTP headers returned in the requests to the STS. Access them via `MsalServiceException.headers()`

MsalInteractionRequiredException

One of common status codes returned from MSAL for Java when calling `AcquireTokenSilently()` is `InvalidGrantError`. This means that additional user interaction is required before an authentication token can be issued. Your application should call the authentication library again, but in interactive mode by sending `AuthorizationCodeParameters` or `DeviceCodeParameters` for public client applications.

Most of the time when `AcquireTokenSilently` fails, it's because the token cache doesn't have a token matching your request. Access tokens expire in one hour, and `AcquireTokenSilently` will try to get 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 such as when a tenant admin configures more stringent login policies.

Some conditions that result in this error are easy for users to resolve. For example, they may need to accept Terms of Use or the request can't be fulfilled with the current configuration because the machine needs to connect to a specific corporate network.

MSAL exposes a `reason` field, which you can use to provide a better user experience. For example, the `reason` field may lead you 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 `InteractionRequiredExceptionReason` enum:

REASON	MEANING	RECOMMENDED HANDLING
BasicAction	Condition can be resolved by user interaction during the interactive authentication flow.	Call <code>acquireToken</code> with interactive parameters.
AdditionalAction	Condition can be resolved by additional remedial interaction with the system outside of the interactive authentication flow.	Call <code>acquireToken</code> with interactive parameters to show a message that explains the remedial action to take. The calling app may choose to hide flows that require additional action if the user is unlikely to complete the remedial action.
MessageOnly	Condition can't be resolved at this time. Launch interactive authentication flow to show a message explaining the condition.	Call <code>acquireToken</code> with interactive parameters to show a message that explains the condition. <code>acquireToken</code> will return the <code>UserCanceled</code> error after the user reads the message and closes the window. The app may choose to hide flows that result in message if the user is unlikely to benefit from the message.
ConsentRequired	User consent is missing, or has been revoked.	Call <code>acquireToken</code> with interactive parameters so that the user can give consent.
UserPasswordExpired	User's password has expired.	Call <code>acquireToken</code> with interactive parameter so the user can reset their password.
None	Further details are provided. The condition may be resolved by user interaction during the interactive authentication flow.	Call <code>acquireToken</code> with interactive parameters.

Code Example

```
IAuthenticationResult result;
try {
    PublicClientApplication application = PublicClientApplication
        .builder("clientId")
        .b2cAuthority("authority")
        .build();

    SilentParameters parameters = SilentParameters
        .builder(Collections.singleton("scope"))
        .build();

    result = application.acquireTokenSilently(parameters).join();
}
catch (Exception ex){
    if(ex instanceof MsalInteractionRequiredException){
        // AcquireToken by either AuthorizationCodeParameters or DeviceCodeParameters
    } else{
        // Log and handle exception accordingly
    }
}
```

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. This 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.

Retrying after errors and exceptions

You're expected to implement your own retry policies when calling MSAL. MSAL makes HTTP calls to the Azure AD service, and occasionally failures can occur. For example the network can go down or the server is overloaded.

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.

Next steps

Consider enabling [Logging in MSAL for Java](#) to help you diagnose and debug issues.

Logging in MSAL for Java

4/12/2022 • 2 minutes to read • [Edit Online](#)

The 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. Used 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.

The following sections provide more details about MSAL error logging for your application.

MSAL for Java logging

MSAL for Java allows you to use the logging library that you are already using with your app, as long as it is compatible with SLF4J. MSAL for Java 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 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();
```

Next steps

For more code samples, refer to [Microsoft identity platform code samples](#).

Microsoft Identity Web authentication library

4/12/2022 • 3 minutes to read • [Edit Online](#)

Microsoft Identity Web is a set of ASP.NET Core libraries that simplifies adding authentication and authorization support to web apps and web APIs integrating with the Microsoft identity platform. It provides a single-surface API convenience layer that ties together ASP.NET Core, its authentication middleware, and the [Microsoft Authentication Library \(MSAL\) for .NET](#).

You can get `Microsoft.Identity.Web` from NuGet or by using a Visual Studio project template to create a new app project.

Supported application scenarios

If you're building ASP.NET Core web apps or web APIs and want to use Azure Active Directory (Azure AD) or Azure AD B2C for identity and access management (IAM), we recommend using Microsoft Identity Web for all of these scenarios:

- [Web app that signs in users](#)
- [Web app that signs in users and calls a web API on their behalf](#)
- [Protected web API that only authenticated users can access](#)
- [Protected web API that calls another \(downstream\) web API on behalf of the signed-in user](#)

Install from NuGet

Microsoft Identity Web is available on NuGet as a set of packages that provide modular functionality based on your app's needs. Use the .NET CLI's `dotnet add` command or Visual Studio's [NuGet Package Manager](#) to install the packages appropriate for your project:

- [Microsoft.Identity.Web](#) - The main package. Required by all apps that use Microsoft Identity Web.
- [Microsoft.Identity.Web.UI](#) - Optional. Adds UI for user sign-in and sign-out and an associated controller for web apps.
- [Microsoft.Identity.Web.MicrosoftGraph](#) - Optional. Provides simplified interaction with the Microsoft Graph API.
- [Microsoft.Identity.Web.MicrosoftGraphBeta](#) - Optional. Provides simplified interaction with the Microsoft Graph API [beta endpoint](#).

Install by using a Visual Studio project template

Several project templates that use Microsoft Identity Web are included in .NET SDK versions 5.0 and above. The project templates aren't included in the ASP.NET Core 3.1 SDK, but you can install them separately.

.NET 5.0+ - Project templates included

The Microsoft Identity Web project templates are included in .NET SDK versions 5.0 and above.

This example .NET CLI command creates a Blazor Server project that includes Microsoft Identity Web.

```
dotnet new blazorserver --auth SingleOrg --calls-graph --client-id "00000000-0000-0000-0000-000000000000" --tenant-id "11111111-1111-1111-1111-111111111111" --output my-blazor-app
```

Don't append a `[2]` to the application type argument (`blazorserver` in the example) if you're using the templates

included in .NET SDK 5.0+. Include the `v2` suffix *only* if you're on ASP.NET Core 3.1 and you installed the templates separately as described in the next section.

ASP.NET Core 3.1 - Install the project templates

If you're using ASP.NET Core 3.1, install the project templates from NuGet.

```
dotnet new --install Microsoft.Identity.Web.ProjectTemplates
```

For ASP.NET Core 3.1 *only*, append a `v2` to the application type argument when you create a new project:

```
dotnet new blazorserver2 --auth SingleOrg --calls-graph --client-id "00000000-0000-0000-0000-000000000000" --tenant-id "11111111-1111-1111-1111-111111111111" --output my-blazor-app
```

The following diagram shows several of the available app type templates and their arguments. Append a `v2` to the app type argument (`blazorserver2` in the example) only if you're using the ASP.NET Core 3.1 SDK and you installed the templates by using `dotnet new --install`.

Scenario	Command	Template		Audience	Call web API (optional)
Web API	dotnet new	webapi2	--auth	SingleOrg	
MVC web app		mvc2			--calls-graph**
Razor web app		webapp2		MultiOrg*	
Blazor Server		blazorserver2			--called-api-scopes "scopes" --called-api-url webApiUrl
Blazor WebAssembly		blazorwasm2 blazorwasm2 --hosted		IndividualB2C	

* `MultiOrg` is not supported with `webapi2`, but can be enabled in `appsettings.json` by setting tenant to `common` or `organizations`

** `--calls-graph` is not supported for Azure AD B2C

Features of the project templates

Microsoft Identity Web includes several features not available in the default ASP.NET Core 3.1 project templates.

FEATURE	ASP.NET CORE 3.1	MICROSOFT IDENTITY WEB
Sign in users in web apps	<ul style="list-style-type: none">Work or school accountsSocial identities (with Azure AD B2C)	<ul style="list-style-type: none">Work or school accountsPersonal Microsoft accountsSocial identities (with Azure AD B2C)
Protect web APIs	<ul style="list-style-type: none">Work or school accountsSocial identities (with Azure AD B2C)	<ul style="list-style-type: none">Work or school accountsPersonal Microsoft accountsSocial identities (with Azure AD B2C)
Issuer validation in multi-tenant apps	No	Yes, for all clouds and Azure AD B2C
Web app/API calls Microsoft graph	No	Yes
Web app/API calls web API	No	Yes
Supports certificate credentials	No	Yes, including Azure Key Vault

FEATURE	ASP.NET CORE 3.1	MICROSOFT IDENTITY WEB
Incremental consent and conditional access support in web apps	No	Yes, in MVC, Razor pages, and Blazor
Token encryption certificates in web APIs	No	Yes
Scopes/app role validation in web APIs	No	Yes
<code>WWW-Authenticate</code> header generation in web APIs	No	Yes

Next steps

To see Microsoft Identity Web in action, try our Blazor Server tutorial:

[Tutorial: Create a Blazor Server app that uses the Microsoft identity platform for authentication](#)

The Microsoft Identity Web wiki on GitHub contains extensive reference documentation for various aspects of the library. For example, certificate usage, incremental consent, and conditional access reference can be found here:

- [Using certificates with Microsoft.Identity.Web](#) (GitHub)
- [Incremental consent and conditional access](#) (GitHub)

Application configuration options

4/12/2022 • 6 minutes to read • [Edit Online](#)

To authenticate and acquire tokens, you initialize a new public or confidential client application in your code. You can set several configuration options when you initialize the client app in the 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:

COMMON AUTHORITY URLs	WHEN TO USE
<code>https://login.microsoftonline.com/<tenant>/</code>	Sign in users of a specific organization only. The < <i>tenant</i> > in the URL is the tenant ID of the Azure Active Directory (Azure AD) tenant (a GUID), or its tenant domain.
<code>https://login.microsoftonline.com/common/</code>	Sign in users with work and school accounts or personal Microsoft accounts.
<code>https://login.microsoftonline.com/organizations/</code>	Sign in users with work and school accounts.
<code>https://login.microsoftonline.com/consumers/</code>	Sign in users with personal Microsoft accounts (MSA) only.

The authority you specify in your code needs to be consistent with the **Supported account types** you specified for the app in **App registrations** in the Azure portal.

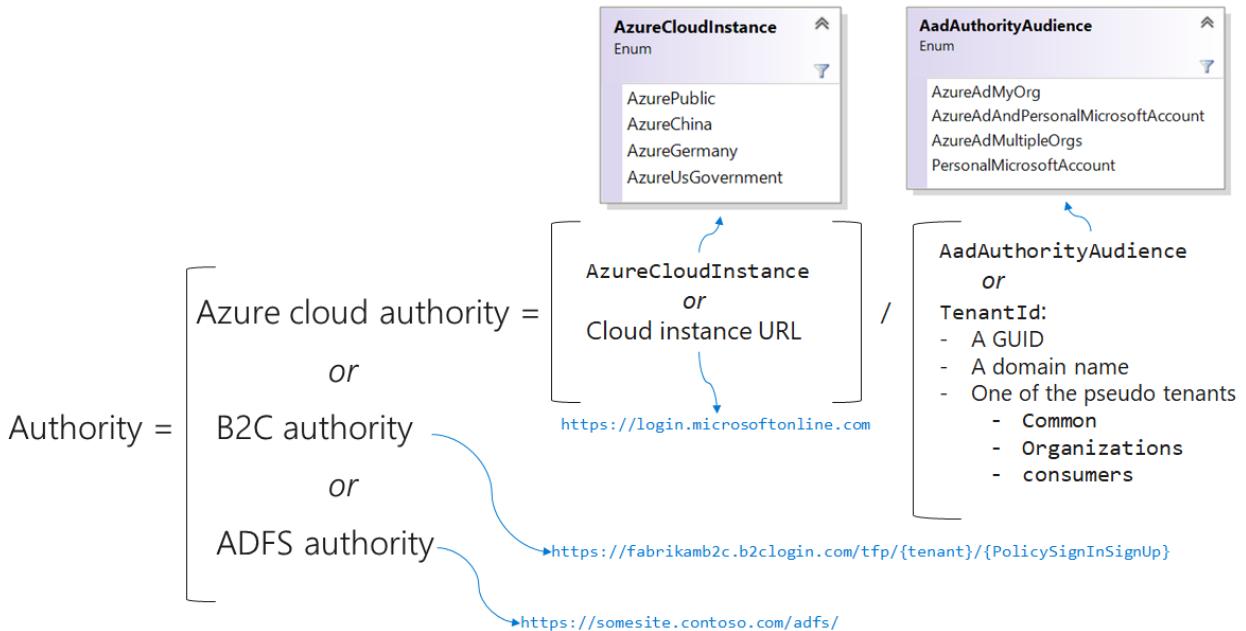
The authority can be:

- 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. 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.onmicrosoftonline.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, specify the organization 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></p> <p>. This enables SSO with the browser by setting the value to the result of <code>WebAuthenticationBroker.GetCurrentApplicationCallbackUri()</code> 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 the broker redirect URI. It is automatically set to `msal{ClientId}://auth` for Xamarin Android and iOS.
- Configure the redirect URI in [App registrations](#):

Configure Desktop + devices

X

All platforms

Quickstart Docs

Redirect URIs

The URLs we will accept as destinations when returning authentication responses (tokens) after successfully authenticating users. Also referred to as reply URLs. [Learn more about Redirect URLs and their restrictions](#)

- <https://login.microsoftonline.com/common/oauth2/nativeclient> 
- https://login.live.com/oauth20_desktop.srf (LiveSDK) 
- <msal649be5cf-abfc-430e-a235-e5029775aded://auth> (MSAL only) 

Custom redirect URIs



You can override the redirect URI by using the `RedirectUri` property (for example, if you use brokers). Here are some examples of redirect URLs 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 URL) is the URL that Azure AD will use to send the token back to the application. This URL 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

To help in debugging and authentication failure troubleshooting scenarios, the Microsoft Authentication Library provides built-in logging support. Logging is each library is covered in the following articles:

- [Logging in MSAL.NET](#)
- [Logging in MSAL for Android](#)
- [Logging in MSAL.js](#)
- [Logging in MSAL for iOS/macOS](#)
- [Logging in MSAL for Java](#)

- Logging in MSAL for Python

Next steps

Learn about [instantiating client applications by using MSAL.NET](#) and [instantiating client applications by using MSAL.js](#).

Initialize client applications using MSAL.NET

4/12/2022 • 5 minutes to read • [Edit Online](#)

This article describes initializing public client and confidential client applications using the Microsoft Authentication Library for .NET (MSAL.NET). To learn more about the client application types, see [Public client and confidential client applications](#).

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

[API reference documentation](#) | [Package on NuGet](#) | [Library source code](#) | [Code samples](#)

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>	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.

MODIFIER	DESCRIPTION
<code>.WithClientId(string)</code>	Overrides the client ID.
<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 Logging for more information.
<code>.WithExtraQueryParameters(IDictionary<string, string> 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 Logging 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>	Xamarin.iOS only: 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)
    .WithAuthority(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)
    .WithAuthority(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();
```

Next steps

After you've initialized the client application, your next task is to add support for user sign-in, authorized API access, or both.

Our application scenario documentation provides guidance for signing in a user and acquiring an access token to access an API on behalf of that user:

- [Web app that signs in users: Sign-in and sign-out](#)
- [Web app that calls web APIs: Acquire a token](#)

Instantiate a confidential client application with configuration options using MSAL.NET

4/12/2022 • 2 minutes to read • [Edit Online](#)

This article describes how to instantiate a [confidential client application](#) using the 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-00a8-4d999re19f292]",  
    "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"  
    }  

```

Starting in MSAL.NET v3.x, you can configure your confidential client application from the config file.

In the class where you want to configure and instantiate your application, 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 as follows:

```
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);
```

Confidential client assertions

4/12/2022 • 5 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()`

NOTE

While it is possible to use the `.WithClientAssertion()` API to acquire tokens for the confidential client, we do not recommend using it by default as it is more advanced and is designed to handle very specific scenarios which are not common. Using the `.WithCertificate()` API will allow MSAL.NET to handle this for you. This API offers you the ability to customize your authentication request if needed but the default assertion created by `.WithCertificate()` will suffice for most authentication scenarios. This API can also be used as a workaround in some scenarios where MSAL.NET fails to perform the signing operation internally.

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();
```

You can also use the delegate form, which enables you to compute the assertion just in time:

```
string signedClientAssertion = ComputeAssertion();
app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
    .WithClientAssertion(() => { return GetSignedClientAssertion(); })
    .Build();

// or in async manner

app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
    .WithClientAssertion(async cancellationToken => { return await
GetClientAssertionAsync(cancellationToken); })
    .Build();
```

The [claims expected by Azure AD](#) in the signed assertion are:

CLAIM TYPE	VALUE	DESCRIPTION
aud	https://login.microsoftonline.com/{tenant}/token	The "aud" (audience) claim identifies the recipients that the JWT is intended for (here Azure AD) See RFC 7519, Section 4.1.3 . In this case, that recipient is the login server (login.microsoftonline.com).
exp	1601519414	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 . This allows the assertion to be used until then, so keep it short - 5-10 minutes after <code>nbf</code> at most. Azure AD does not place restrictions on the <code>exp</code> time currently.
iss	{ClientID}	The "iss" (issuer) claim identifies the principal that issued the JWT, in this case your client application. Use the GUID application ID.
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" value is a case-sensitive string. RFC 7519, Section 4.1.7
nbf	1601519114	The "nbf" (not before) claim identifies the time before which the JWT MUST NOT be accepted for processing. RFC 7519, Section 4.1.5 . Using the current time is appropriate.
sub	{ClientID}	The "sub" (subject) claim identifies the subject of the JWT, in this case also your application. Use the same value as <code>iss</code> .

If you use a certificate as a client secret, the certificate must be deployed safely. We recommend that you store the certificate in a secure spot supported by the platform, such as in the certificate store on Windows or by using Azure Key Vault.

Here's an example of how to craft these claims:

```
using System.Collections.Generic;
private static IDictionary<string, object> GetClaims(string tenantId, string clientId)
{
    //aud = https://login.microsoftonline.com/ + Tenant ID + /v2.0
    string aud = $"https://login.microsoftonline.com/{tenantId}/v2.0";

    string ConfidentialClientID = clientId; //client id 00000000-0000-0000-0000-000000000000
    const uint JwtToAadLifetimeInSeconds = 60 * 10; // Ten minutes
    DateTimeOffset validFrom = DateTimeOffset.UtcNow;
    DateTimeOffset validUntil = validFrom.AddSeconds(JwtToAadLifetimeInSeconds);

    return new Dictionary<string, object>()
    {
        { "aud", aud },
        { "exp", validUntil.ToUnixTimeSeconds() },
        { "iss", ConfidentialClientID },
        { "jti", Guid.NewGuid().ToString() },
        { "nbf", validFrom.ToUnixTimeSeconds() },
        { "sub", ConfidentialClientID }
    };
}
```

Here's how to craft a signed client assertion:

```

using System.Collections.Generic;
using System.Security.Cryptography.X509Certificates;
using System.Security.Cryptography;
using System.Text;
using System.Text.Json;
...
static string Base64UrlEncode(byte[] arg)
{
    char Base64PadCharacter = '=';
    char Base64Character62 = '+';
    char Base64Character63 = '/';
    char Base64UrlCharacter62 = '-';
    char Base64UrlCharacter63 = '_';

    string s = Convert.ToBase64String(arg);
    s = s.Split(Base64PadCharacter)[0]; // Remove any trailing padding
    s = s.Replace(Base64Character62, Base64UrlCharacter62); // 62nd char of encoding
    s = s.Replace(Base64Character63, Base64UrlCharacter63); // 63rd char of encoding

    return s;
}

static string GetSignedClientAssertion(X509Certificate2 certificate, string tenantId, string clientId)
{
    // Get the RSA with the private key, used for signing.
    var rsa = certificate.GetRSAPrivateKey();

    //alg represents the desired signing algorithm, which is SHA-256 in this case
    //x5t represents the certificate thumbprint base64 url encoded
    var header = new Dictionary<string, string>()
    {
        { "alg", "RS256" },
        { "typ", "JWT" },
        { "x5t", Base64UrlEncode(certificate.GetCertHash()) }
    };

    //Please see the previous code snippet on how to craft claims for the GetClaims() method
    var claims = GetClaims(tenantId, clientId);

    var headerBytes = JsonSerializer.SerializeToUtf8Bytes(header);
    var claimsBytes = JsonSerializer.SerializeToUtf8Bytes(claims);
    string token = Base64UrlEncode(headerBytes) + "." + Base64UrlEncode(claimsBytes);

    string signature = Base64UrlEncode(rsa.SignData(Encoding.UTF8.GetBytes(token), HashAlgorithmName.SHA256,
RSASignaturePadding.Pkcs1));
    string signedClientAssertion = string.Concat(token, ".", signature);
    return signedClientAssertion;
}

```

Alternative method

You also have the option of using [Microsoft.IdentityModel.JsonWebTokens](#) to create the assertion for you. The code will be a more elegant as shown in the example below:

```

string GetSignedClientAssertionAlt(X509Certificate2 certificate)
{
    //aud = https://login.microsoftonline.com/ + Tenant ID + /v2.0
    string aud = $"https://login.microsoftonline.com/{tenantID}/v2.0";

    // client_id
    string confidentialClientID = "00000000-0000-0000-0000-000000000000";

    // no need to add exp, nbf as JsonWebTokenHandler will add them by default.
    var claims = new Dictionary<string, object>()
    {
        { "aud", aud },
        { "iss", confidentialClientID },
        { "jti", Guid.NewGuid().ToString() },
        { "sub", confidentialClientID }
    };

    var securityTokenDescriptor = new SecurityTokenDescriptor
    {
        Claims = claims,
        SigningCredentials = new X509SigningCredentials(certificate)
    };

    var handler = new JsonWebTokenHandler();
    var signedClientAssertion = handler.CreateToken(securityTokenDescriptor);
}

```

Once you have your signed client assertion, you can use it with the MSAL apis as shown below.

```

X509Certificate2 certificate = ReadCertificate(config.CertificateName);
string signedClientAssertion = GetSignedClientAssertion(certificate, tenantId,
ConfidentialClientID)
    // OR
    //string signedClientAssertion = GetSignedClientAssertionAlt(certificate);

var confidentialApp = ConfidentialClientApplicationBuilder
    .Create(ConfidentialClientID)
    .WithClientAssertion(signedClientAssertion)
    .Build();

```

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.

Instantiate a public client application with configuration options using MSAL.NET

4/12/2022 • 2 minutes to read • [Edit Online](#)

This article describes how to instantiate a [public client application](#) using the 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.

Default Reply URI

In MSAL.NET 4.1+ the default redirect URI (Reply URI) can now be set with the

`public PublicClientApplicationBuilder WithDefaultRedirectUri()` method. This method will set the redirect URI property of public client application to the recommended default.

This method's behavior is dependent upon the platform that you are using at the time. Here is a table that describes what redirect URI is set on certain platforms:

PLATFORM	REDIRECT URI
Desktop app (.NET FW)	<code>https://login.microsoftonline.com/common/oauth2/nativeclient</code>
UWP	value of <code>WebAuthenticationBroker.GetCurrentApplicationCallbackUri()</code>
.NET Core	<code>http://localhost</code>

For the UWP platform, is enhanced the experience by enabling SSO with the browser by setting the value to the result of `WebAuthenticationBroker.GetCurrentApplicationCallbackUri()`.

For .NET Core, MSAL.Net is setting the value to the local host to enable the user to use the system browser for interactive authentication.

NOTE

For embedded browsers in desktop scenarios the redirect URI used is intercepted by MSAL to detect that a response is returned from the identity provider that an auth code has been returned. This URI can therefore be used in any cloud without seeing an actual redirect to that URI. This means you can and should use

`https://login.microsoftonline.com/common/oauth2/nativeclient` in any cloud. If you prefer you can also use any other URI as long as you configure the redirect URI correctly with MSAL and in the app registration. Specifying the default URI in the application registration means there is the least amount of setup in MSAL.

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();

```

Providing your own HttpClient and proxy using MSAL.NET

4/12/2022 • 2 minutes to read • [Edit Online](#)

When [initializing a 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).

`HttpClient` is intended to be instantiated once and then reused throughout the life of an application. See

[Remarks](#).

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();
```

Example implementation using a proxy

```

public class HttpFactoryWithProxy : IMsalHttpClientFactory
{
    private static HttpClient _httpClient;

    public HttpFactoryWithProxy()
    {
        // Consider using Lazy<T>
        if (_httpClient == null)
        {
            var proxy = new WebProxy
            {
                Address = new Uri($"http://{proxyHost}:{proxyPort}"),
                BypassProxyOnLocal = false,
                UseDefaultCredentials = false,
                Credentials = new NetworkCredential(
                    userName: proxyUserName,
                    password: proxyPassword)
            };

            // Now create a client handler which uses that proxy
            var httpClientHandler = new HttpClientHandler
            {
                Proxy = proxy,
            };

            _httpClient = new HttpClient(handler: httpClientHandler);
        }
    }

    public HttpClient GetHttpClient()
    {
        return _httpClient;
    }
}

```

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`](#).

Using web browsers (MSAL.NET)

4/12/2022 • 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 [multi-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 .NET 5.0+ it uses WebView2, 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. To change this behavior, use

```
WithUseEmbeddedWebView(bool)
```

At a glance

FRAMEWORK	EMBEDDED	SYSTEM	DEFAULT
.NET 5.0+	Yes [†]	Yes [^]	Embedded

FRAMEWORK	EMBEDDED	SYSTEM	DEFAULT
.NET Classic	Yes	Yes^	Embedded
.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 OS-specific target framework moniker (TFM) of at least `net5.0-windows10.0.17763.0`. Do *not* use the `net5.0` or `net5.0-windows` TFMs. For more information about TFMs, see [Target frameworks in SDK-style projects](#).

^ Requires redirect URI `http://localhost`

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

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 or .NET 5 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 macOS

On Linux, MSAL.NET opens the default OS browser with a tool like `xdg-open`. Opening the browser with `sudo` is unsupported by MSAL and will cause MSAL to throw an exception.

On macOS, the browser is opened by invoking `open <url>`.

Customizing the experience

MSAL.NET can respond with an HTTP message or HTTP redirect when a token is received or an error occurs.

```
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.

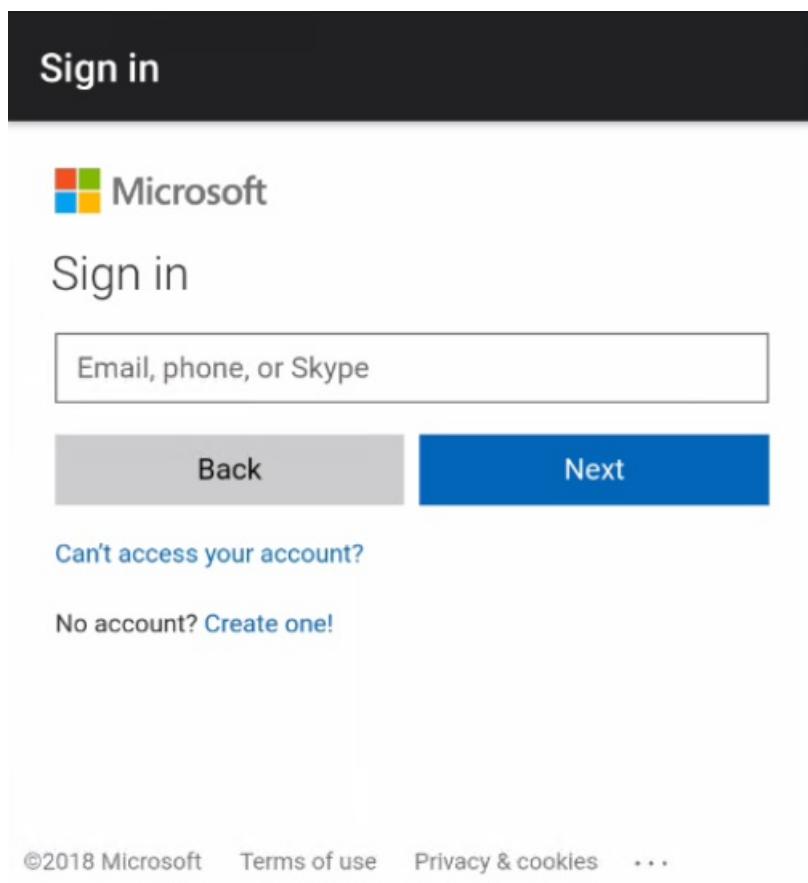
As a developer using MSAL.NET targeting Xamarin, you may choose to use either embedded webviews or system browsers. This is your choice depending on the user experience and security concerns you want to target.

Currently, MSAL.NET doesn't yet support the Android and iOS brokers. Therefore 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.

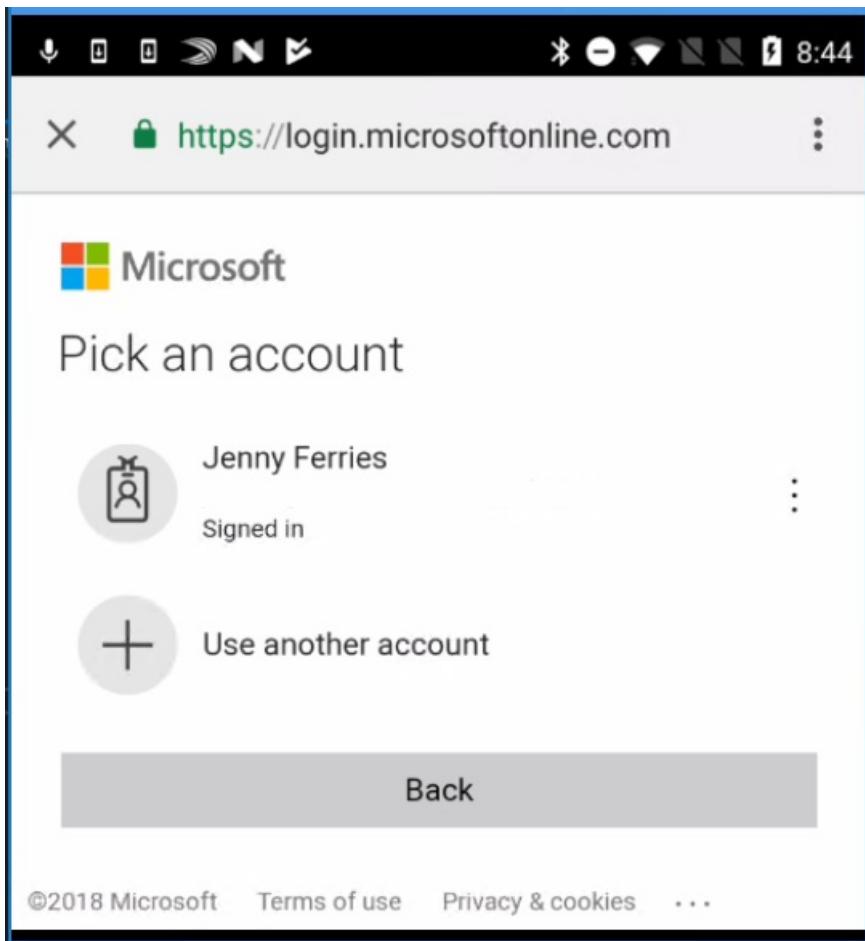
Differences between embedded webview and system browser

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.

Considerations for using Universal Windows Platform with MSAL.NET

4/12/2022 • 2 minutes to read • [Edit Online](#)

Developers of applications that use Universal Windows Platform (UWP) with MSAL.NET should consider the concepts this article presents.

The UseCorporateNetwork property

On the Windows Runtime (WinRT) platform, `PublicClientApplication` has the Boolean property `UseCorporateNetwork`. This property enables Windows 10 applications and UWP applications to benefit from integrated Windows authentication (IWA) if the user is signed in to an account that has a federated Azure Active Directory (Azure AD) tenant. Users who are signed in to the operating system can also use single sign-on (SSO). When you set the `UseCorporateNetwork` property, MSAL.NET uses a web authentication broker (WAB).

IMPORTANT

Setting the `UseCorporateNetwork` property to true assumes that the application developer has enabled IWA in the application. To enable IWA:

- In your UWP application's `Package.appxmanifest`, on the **Capabilities** tab, enable the following capabilities:
 - Enterprise Authentication
 - Private Networks (Client & Server)
 - Shared User Certificate

IWA isn't enabled by default because Microsoft Store requires a high level of verification before it accepts applications that request the capabilities of enterprise authentication or shared user certificates. Not all developers want to do this level of verification.

On the UWP platform, the underlying WAB implementation doesn't work correctly in enterprise scenarios where conditional access is enabled. Users see symptoms of this problem when they try to sign in by using Windows Hello. When the user is asked to choose a certificate:

- The certificate for the PIN isn't found.
- After the user chooses a certificate, they aren't prompted for the PIN.

You can try to avoid this issue by using an alternative method such as username-password and phone authentication, but the experience isn't good.

Troubleshooting

Some customers have reported the following sign-in error in specific enterprise environments in which they know that they have an internet connection and that the connection works with a public network.

We can't connect to the service you need right now. Check your network connection or try this again later.

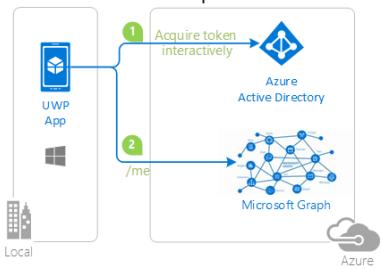
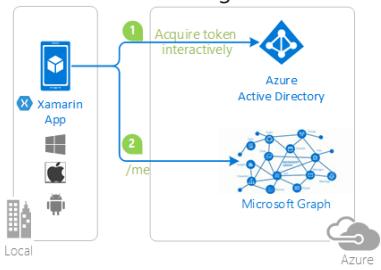
You can avoid this issue by making sure that WAB (the underlying Windows component) allows a private network. You can do that by setting a registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\authhost.exe\EnablePrivateNetwork = 00000001
```

For more information, see [Web authentication broker - Fiddler](#).

Next steps

The following samples provide more information.

SAMPLE	PLATFORM	DESCRIPTION
active-directory-dotnet-native-uwp-v2	UWP	A UWP client application that uses MSAL.NET. It accesses Microsoft Graph for a user who authenticates by using an Azure AD 2.0 endpoint. 
active-directory-xamarin-native-v2	Xamarin iOS, Android, UWP	A Xamarin Forms app that shows how to use MSAL to authenticate Microsoft personal accounts and Azure AD via the Microsoft identity platform. It also shows how to access Microsoft Graph and shows the resulting token. 

Get a token from the token cache using MSAL.NET

4/12/2022 • 2 minutes to read • [Edit Online](#)

When you acquire an access token using the 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).

For authentication flows that require a user interaction, MSAL caches the access, refresh, and ID tokens, as well as the `IAccount` object, which represents information about a single account. Learn more about [IAccount](#). For application flows, such as [client credentials](#), only access tokens are cached, because the `IAccount` object and ID token require a user, and the refresh token is not applicable.

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.

```
var accounts = await app.GetAccountsAsync();

AuthenticationResult result = null;
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
    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
}
```

Clear the token cache using MSAL.NET

4/12/2022 • 2 minutes to read • [Edit Online](#)

When you [acquire an access token](#) using the 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](#).

Token cache serialization in MSAL.NET

4/12/2022 • 19 minutes to read • [Edit Online](#)

After Microsoft Authentication Library (MSAL) [acquires a token](#), it caches that token. Public client applications (desktop and mobile apps) should try to get a token from the cache before acquiring a token by another method. Acquisition methods on confidential client applications manage the cache themselves. This article discusses default and custom serialization of the token cache in MSAL.NET.

Quick summary

The recommendation is:

- When you're writing a desktop application, use the cross-platform token cache as explained in [Desktop apps](#).
- Do nothing for [mobile and UWP apps](#). MSAL.NET provides secure storage for the cache.
- In ASP.NET Core [web apps](#) and [web APIs](#), use [Microsoft.Identity.Web](#) as a higher-level API. You'll get token caches and much more. See [ASP.NET Core web apps and web APIs](#).
- In the other cases of [web apps](#) and [web APIs](#):
 - If you request tokens for users in a production application, use a [distributed token cache](#) (Redis, SQL Server, Azure Cosmos DB, distributed memory). Use token cache serializers available from [Microsoft.Identity.Web.TokenCache](#).
 - Otherwise, if you want to use an in-memory cache:
 - If you're only using `AcquireTokenForClient`, either reuse the confidential client application instance and don't add a serializer, or create a new confidential client application and enable the [shared cache option](#).

A shared cache is faster because it's not serialized. However, the memory will grow as tokens are cached. The number of tokens is equal to the number of tenants times the number of downstream APIs. An app token is about 2 KB in size, whereas tokens for a user are about 7 KB in size. It's great for development, or if you have few users.

- If you want to use an in-memory token cache and control its size and eviction policies, use the [Microsoft.Identity.Web in-memory cache option](#).
- If you build an SDK and want to write your own token cache serializer for confidential client applications, inherit from [Microsoft.Identity.Web.MsalAbstractTokenCacheProvider](#) and override the `WriteCacheBytesAsync` and `ReadCacheBytesAsync` methods.
- [ASP.NET Core web apps and web APIs](#)
- [Non-ASP.NET Core web apps and web APIs](#)
- [Desktop apps](#)
- [Mobile apps](#)
- [Write your own cache](#)

The [Microsoft.Identity.Web.TokenCache](#) NuGet package provides token cache serialization within the [Microsoft.Identity.Web](#) library.

EXTENSION METHOD	DESCRIPTION
------------------	-------------

EXTENSION METHOD	DESCRIPTION
AddInMemoryTokenCaches	Creates a temporary cache in memory for token storage and retrieval. In-memory token caches are faster than the other cache types, but their tokens aren't persisted between application restarts, and you can't control the cache size. In-memory caches are good for applications that don't require tokens to persist between app restarts. Use an in-memory token cache in apps that participate in machine-to-machine auth scenarios like services, daemons, and others that use AcquireTokenForClient (the client credentials grant). In-memory token caches are also good for sample applications and during local app development. Microsoft.Identity.Web versions 1.19.0+ share an in-memory token cache across all application instances.
AddSessionTokenCaches	The token cache is bound to the user session. This option isn't ideal if the ID token contains many claims, because the cookie will become too large.
AddDistributedTokenCaches	The token cache is an adapter against the ASP.NET Core IDistributedCache implementation. It enables you to choose between a distributed memory cache, a Redis cache, a distributed NCache, or a SQL Server cache. For details about the IDistributedCache implementations, see Distributed memory cache .

In-memory token cache

Here's an example of code that uses the in-memory cache in the [ConfigureServices](#) method of the [Startup](#) class in an ASP.NET Core application:

```
#using Microsoft.Identity.Web

using Microsoft.Identity.Web;

public class Startup
{
    const string scopesToRequest = "user.read";

    public void ConfigureServices(IServiceCollection services)
    {
        // code before
        services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
            .AddMicrosoftIdentityWebApp(Configuration)
            .EnableTokenAcquisitionToCallDownstreamApi(new string[] { scopesToRequest })
            .AddInMemoryTokenCaches();
        // code after
    }
    // code after
}
```

[AddInMemoryTokenCaches](#) is suitable in production if you request app-only tokens. If you use user tokens, consider using a distributed token cache.

Token cache configuration code is similar between ASP.NET Core web apps and web APIs.

Distributed token caches

Here are examples of possible distributed caches:

```

// or use a distributed Token Cache by adding
services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
    .AddMicrosoftIdentityWebApp(Configuration)
    .EnableTokenAcquisitionToCallDownstreamApi(new string[] { scopesToRequest })
    .AddDistributedTokenCaches();

// Distributed token caches have a L1/L2 mechanism.
// L1 is in memory, and L2 is the distributed cache
// implementation that you will choose below.
// You can configure them to limit the memory of the
// L1 cache, encrypt, and set eviction policies.
services.Configure<MsalDistributedTokenCacheAdapterOptions>(options =>
{
    // Optional: Disable the L1 cache in apps that don't use session affinity
    //           by setting DisableL1Cache to 'false'.
    options.DisableL1Cache = false;

    // Or limit the memory (by default, this is 500 MB)
    options.L1CacheOptions.SizeLimit = 1024 * 1024 * 1024, // 1 GB

    // You can choose if you encrypt or not encrypt the cache
    options.Encrypt = false;

    // And you can set eviction policies for the distributed
    // cache.
    options.SlidingExpiration = TimeSpan.FromHours(1);
}

// Then, choose your implementation of distributed cache
// -----
// good for prototyping and testing, but this is NOT persisted and it is NOT distributed - do not use in
// production
services.AddDistributedMemoryCache();

// Or a Redis cache
// Requires the Microsoft.Extensions.Caching.StackExchangeRedis NuGet package
services.AddStackExchangeRedisCache(options =>
{
    options.Configuration = "localhost";
    options.InstanceName = "SampleInstance";
});

// You can even decide if you want to repair the connection
// with Redis and retry on Redis failures.
services.Configure<MsalDistributedTokenCacheAdapterOptions>(options =>
{
    options.OnL2CacheFailure = (ex) =>
    {
        if (ex is StackExchange.Redis.RedisConnectionException)
        {
            // action: try to reconnect or something
            return true; //try to do the cache operation again
        }
        return false;
    };
});

// Or even a SQL Server token cache
// Requires the Microsoft.Extensions.Caching.SqlServer NuGet package
services.AddDistributedSqlServerCache(options =>
{
    options.ConnectionString = _config["DistCache_ConnectionString"];
    options.SchemaName = "dbo";
    options.TableName = "TestCache";
});

// Or an Azure Cosmos DB cache

```

```
// Requires the Microsoft.Extensions.Caching.Cosmos NuGet package
services.AddCosmosCache((CosmosCacheOptions cacheOptions) =>
{
    cacheOptions.ContainerName = Configuration["CosmosCacheContainer"];
    cacheOptions.DatabaseName = Configuration["CosmosCacheDatabase"];
    cacheOptions.ClientBuilder = new CosmosClientBuilder(Configuration["CosmosConnectionString"]);
    cacheOptions.CreateIfNotExists = true;
});
```

For more information, see:

- [Distributed cache advanced options](#)
- [Handle L2 cache eviction](#)
- [Set up a Redis cache in Docker](#)
- [Troubleshooting](#)

The usage of distributed cache is featured in the [ASP.NET Core web app tutorial](#) in the phase 2-2 token cache.

Monitor cache hit ratios and cache performance

MSAL exposes important metrics as part of [AuthenticationResult.AuthenticationResultMetadata](#) object. You can log these metrics to assess the health of your application.

METRIC	MEANING	WHEN TO TRIGGER AN ALARM?
<code>DurationTotalInMs</code>	Total time spent in MSAL, including network calls and cache.	Alarm on overall high latency (> 1 second). Value depends on token source. From the cache: one cache access. From Azure Active Directory (Azure AD): two cache accesses plus one HTTP call. First ever call (per-process) will take longer because of one extra HTTP call.
<code>DurationInCacheInMs</code>	Time spent loading or saving the token cache, which is customized by the app developer (for example, save to Redis).	Alarm on spikes.
<code>DurationInHttpInMs</code>	Time spent making HTTP calls to Azure AD.	Alarm on spikes.
<code>TokenSource</code>	Source of the token. Tokens are retrieved from the cache much faster (for example, ~100 ms versus ~700 ms). Can be used to monitor and alarm the cache hit ratio.	Use with <code>DurationTotalInMs</code> .
<code>CacheRefreshReason</code>	Reason for fetching the access token from the identity provider.	Use with <code>TokenSource</code> .

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 that calls the Microsoft Graph API.</p> <pre> graph LR Local[Local] --- Client[Desktop App (WPF) TodoListClient] Client -- "1 Acquire token interactively" --> AD[Azure Active Directory] Client -- "/me" --> Graph[Microsoft Graph] subgraph "Azure" AD Graph end </pre> <p>The diagram illustrates the flow of a desktop application (TodoListClient) interacting with Azure Active Directory and the Microsoft Graph API. Step 1 shows the client acquiring a token interactively from Azure Active Directory. Step 2 shows the client making a call to the Microsoft Graph API endpoint '/me'.</p>
active-directory-dotnet-v1-to-v2	Desktop (console)	<p>Set of Visual Studio solutions that illustrate the migration of Azure AD v1.0 applications (using ADAL.NET) to Microsoft identity platform applications (using MSAL.NET). In particular, see Token cache migration and Confidential client token cache.</p>
ms-identity-aspnet-webapp-openidconnect	ASP.NET (net472)	<p>Example of token cache serialization in an ASP.NET MVC application (using MSAL.NET). In particular, see MsalAppBuilder.</p>

User gets consent for several resources using MSAL.NET

4/12/2022 • 2 minutes to read • [Edit Online](#)

The Microsoft identity platform 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, to access the second web API you can silently acquire the token from the token cache:

```
AcquireTokenSilent(scopesForVendorApi, accounts.FirstOrDefault()).ExecuteAsync();
```

Active Directory Federation Services support in MSAL.NET

4/12/2022 • 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-premises system to AD FS 2019, you'll be able to use MSAL.NET.

Next steps

For the federated case, see [Configure Azure Active Directory sign in behavior for an application by using a Home Realm Discovery policy](#)

Use MSAL.NET to sign in users with social identities

4/12/2022 • 5 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, specify the policy as part of the authority.
- When you want to apply a policy, call an override of `AcquireTokenInteractive` that accepts the `authority` parameter.

This article applies to MSAL.NET 3.x. For MSAL.NET 2.x, see [Azure AD B2C specifics in MSAL 2.x](#) in the MSAL.NET Wiki on GitHub.

Authority for an Azure AD B2C tenant and policy

The authority format for Azure AD B2C is: `https://{azureADB2Hostname}/tfp/{tenant}/{policyName}`

- `azureADB2Hostname` - The name of the Azure AD B2C tenant plus the host. For example, `contosob2c.b2clogin.com`.
- `tenant` - The domain name or the directory (tenant) ID of the Azure AD B2C tenant. For example, `contosob2c.onmicrosoft.com` or a GUID, respectively.
- `policyName` - The name of the user flow or custom policy to apply. For example, a sign-up/sign-in policy like `b2c_1_susi`.

For more information about Azure AD B2C authorities, see [Set redirect URLs to b2clogin.com](#).

Instantiating the application

Provide the authority by calling `WithB2CAuthority()` when you create the application object:

```
// Azure AD B2C Coordinates
public static string Tenant = "fabrikamb2c.onmicrosoft.com";
public static string AzureADB2Hostname = "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://{{AzureADB2Hostname}}/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:

```

AuthenticationResult authResult = null;
IEnumerable<IAccount> accounts = await application.GetAccountsAsync(policy);
IAccount account = accounts.FirstOrDefault();
try
{
    authResult = await application.AcquireTokenSilent(scopes, account)
        .ExecuteAsync();
}
catch (MsalUiRequiredException ex)
{
    authResult = await application.AcquireTokenInteractive(scopes)
        .WithAccount(account)
        .WithParentActivityOrWindow(ParentActivityOrWindow)
        .ExecuteAsync();
}

```

In the preceding code snippet:

- `policy` is a string containing the name of your Azure AD B2C user flow or custom policy (for example, `PolicySignUpSignIn`).
- `ParentActivityOrWindow` is required for Android (the Activity) and is optional for other platforms that support a parent UI like windows on Microsoft Windows and UIViewController in iOS. For more information on the UI dialog, see [WithParentActivityOrWindow](#) on the MSAL Wiki.

Applying a user flow or custom policy (for example, letting the user edit their profile or reset their password) is currently done by calling `AcquireTokenInteractive`. For these two policies, you don't use the returned token/authentication result.

Profile edit policies

To enable your users to sign in with a social identity and then edit their profile, apply the Azure AD B2C edit profile policy.

Do so by calling `AcquireTokenInteractive` with the authority for that policy. Because the user is already signed in and has an active cookie session, use `Prompt.NoPrompt` to prevent the account selection dialog from being displayed.

```

private async void EditProfileButton_Click(object sender, RoutedEventArgs e)
{
    IEnumerable<IAccount> accounts = await application.GetAccountsAsync(PolicyEditProfile);
    IAccount account = accounts.FirstOrDefault();
    try
    {
        var authResult = await application.AcquireTokenInteractive(scopes)
            .WithPrompt(Prompt.NoPrompt),
            .WithAccount(account)
            .WithB2CAuthority(AuthorityEditProfile)
            .ExecuteAsync();
    }
    catch
    {
    }
}

```

Resource owner password credentials (ROPC)

For more information on the ROPC flow, see [Sign in with resource owner password credentials grant](#).

The ROPC flow is **not recommended** because asking a user for their password in your application is not

secure. For more information about this problem, see [What's the solution to the growing problem of passwords?](#).

By using username/password in an ROPC flow, you sacrifice several things:

- Core tenets of modern identity: The password can be fished or replayed because the shared secret can be intercepted. By definition, ROPC is incompatible with passwordless flows.
- Users who need to do MFA won't be able to sign in (as there is no interaction).
- Users won't be able to use single sign-on (SSO).

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** to enable ROPC for the user flow. For more information, see [Configure the resource owner password credentials flow](#).

`IPublicClientApplication` contains the `AcquireTokenByUsernamePassword` method:

```
AcquireTokenByUsernamePassword(  
    IEnumerable<string> scopes,  
    string username,  
    SecureString password)
```

This `AcquireTokenByUsernamePassword` method takes the following parameters:

- The *scopes* for which to obtain an access token.
- A *username*.
- A `SecureString` *password* for the user.

Limitations of the ROPC flow

The ROPC flow **only works for local accounts**, where your users have registered with Azure AD B2C using an email address or username. This flow doesn't work when federating to an external identity provider supported by Azure AD B2C (Facebook, Google, etc.).

Google auth and embedded webview

If you're using Google as an identity provider, we recommend you use the system browser as Google doesn't allow [authentication from embedded webviews](#). Currently, `login.microsoftonline.com` is a trusted authority with Google and will work with embedded webview. However, `b2clogin.com` is not a trusted authority with Google, so users will not be able to authenticate.

We'll provide an update to this [issue](#) if things change.

Token caching 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 (IdP).

Currently, MSAL.NET needs two claims to build a token cache key:

- `tid` (the Azure AD tenant ID)
- `preferred_username`

Both of these claims may be missing in Azure AD B2C scenarios because not all social identity providers (Facebook, Google, and others) return them in the tokens they return to Azure AD B2C.

A symptom of such a scenario is that MSAL.NET returns `Missing from the token response` when you access the

`preferred_username` claim value in tokens issued by Azure AD B2C. MSAL uses the `Missing from the token response` value for `preferred_username` to maintain cache cross-compatibility between libraries.

Workarounds

Mitigation for missing tenant ID

The suggested workaround is to use [caching by policy](#) described earlier.

Alternatively, you can use the `tid` claim if you're using [custom policies](#) in Azure AD B2C. Custom policies can return additional claims to your application by using [claims transformation](#).

Mitigation for "Missing from the token response"

One option is to use the `name` claim instead of `preferred_username`. To include the `name` claim in ID tokens issued by Azure AD B2C, select **Display Name** when you configure your user flow.

For more information about specifying which claims are returned by your user flows, see [Tutorial: Create user flows in Azure AD B2C](#).

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 Xamarin Forms app that uses MSAL.NET to authenticate users via Azure AD B2C and then access a web API with the tokens returned.

Handle errors and exceptions in MSAL.NET

4/12/2022 • 9 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, 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 [Azure AD Authentication and authorization error codes](#).

During the sign-in experience, you may encounter errors about consents, Conditional Access (MFA, Device Management, Location-based restrictions), token issuance and redemption, and user properties.

The following section provides more details about error handling for your app.

Error handling in MSAL.NET

Exception types

[MsalClientException](#) is thrown when the library itself detects an error state, such as a bad configuration.

[MsalServiceException](#) is thrown when the Identity Provider (AAD) returns an error. It is a translation of the server error.

[MsalUIRequiredException](#) is type of [MsalServiceException](#) and indicates that user interaction is required, for example because MFA is required or because the user has changed their password and a token cannot be acquired silently.

Processing exceptions

When processing .NET 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.

If [MsalUIRequiredException](#) is thrown, it is an indication that an interactive flow needs to happen for the user to resolve the issue. In public client apps such as desktop and mobile app, this is resolved by calling `AcquireTokenInteractive` which displays a browser. In confidential client apps, web apps should redirect the user to the authorization page, and web APIs should return an HTTP status code and header indicative of the authentication failure (401 Unauthorized and a WWW-Authenticate header).

Common .NET exceptions

Here are the common exceptions that might be thrown and some possible mitigations:

EXCEPTION	ERROR CODE	MITIGATION
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.	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: <code>https://login.microsoftonline.com/common/oauth2/v2.0/authorize?client_id={clientId}&response_type=code&scope=user.read</code> to call <code>AcquireTokenInteractive</code> : <code>app.AcquireTokenInteractive(scopes).WithAccount(account).WithClaims(e</code>
MsalUIRequiredException	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, fall back to an interactive flow such as <code>AcquireTokenInteractive</code> .
MsalServiceException	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> .

EXCEPTION	ERROR CODE	MITIGATION
MsalServiceException	AADSTS7002: 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> .
MsalClientException	<code>unknown_user</code> Message : 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 Azure AD joined (work-place 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, john@contoso.com) and use the <code>AcquireTokenByIntegratedWindowsAuth</code> form that takes in the username.
MsalClientException	integrated_windows_auth_not_supported_managed_user	This method relies on a protocol exposed by Active Directory (AD). If a user was created in Azure AD without AD backing ("managed" user), this method will fail. Users created in AD and backed by Azure AD ("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, do have 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 <code>AcquireTokenInteractively()</code> for user to give consent.
UserPasswordExpired	User's password has expired.	Call <code>AcquireTokenInteractively()</code> so that user can reset their password.
PromptNeverFailed	Interactive Authentication was called with the parameter <code>prompt=never</code> , forcing MSAL to rely on browser cookies and not to display the browser. This has failed.	Call <code>AcquireTokenInteractively()</code> without <code>Prompt.None</code>
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 <code>AcquireTokenInteractively()</code> .
None	No further details are provided. Condition may be resolved by user interaction during the interactive authentication flow.	Call <code>AcquireTokenInteractively()</code> .

.NET 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;
    }
}

```

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. This 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.

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.

Retrying after errors and exceptions

You're expected to implement your own retry policies when calling MSAL. MSAL makes HTTP calls to the Azure AD service, and occasionally failures can occur. For example the network can go down or the server is overloaded.

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.

HTTP error codes 500-600

MSAL.NET implements a simple retry-once mechanism for errors with HTTP error codes 500-600.

[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 [RetryAfterproperty](#) (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.

```
bool retry = false;
do
{
    TimeSpan? delay;
    try
    {
        result = await publicClientApplication.AcquireTokenForClient(scopes, account).ExecuteAsync();
    }
    catch (MsalServiceException serviceException)
    {
        if (serviceException.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);
```

Next steps

Consider enabling [Logging in MSAL.NET](#) to help you diagnose and debug issues.

Logging in MSAL.NET

4/12/2022 • 2 minutes to read • [Edit Online](#)

The 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. Used 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.

The following sections provide more details about MSAL error logging for your application.

Configure logging in MSAL.NET

In MSAL logging is set at application 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 (PII) 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 messages that can have PII, in which case the `containsPii` flag will be set to true.
- `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;
    }
}
```

TIP

See the [MSAL.NET wiki](#) for samples of MSAL.NET logging and more.

Next steps

For more code samples, refer to [Microsoft identity platform code samples](#).

Migrating applications to MSAL.NET or Microsoft.Identity.Web

4/12/2022 • 2 minutes to read • [Edit Online](#)

Why migrate to MSAL.NET or Microsoft.Identity.Web

Both the 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 requested tokens from Azure AD for developers platform (v1.0) using Azure AD Authentication Library (ADAL). These tokens are used to authenticate Azure AD identities (work and school accounts).

MSAL comes with benefits over ADAL. Some of these benefits are listed below:

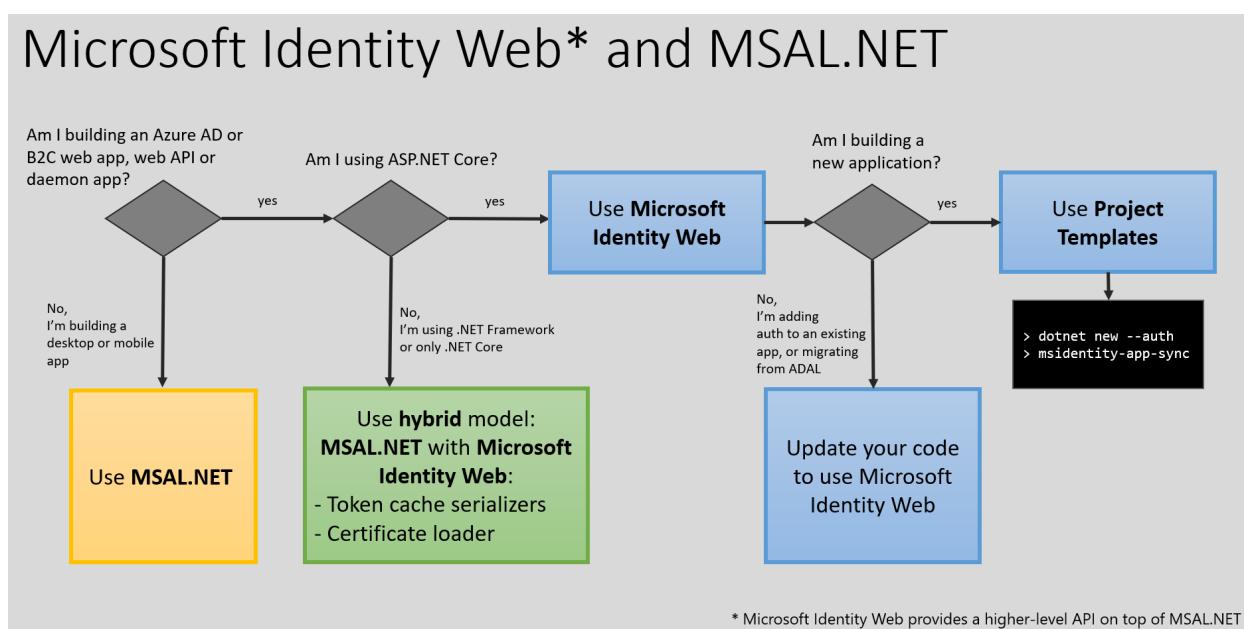
- You can authenticate a broader set of Microsoft identities: work or school accounts, personal Microsoft accounts, and social or local accounts with Azure AD B2C,
- Your users will get the best single-sign-on experience,
- Your application can enable incremental consent, Conditional Access,
- You benefit from continuous innovation in term of security and resilience,
- Your application implements the best practices in term of resilience and security.

MSAL.NET or Microsoft.Identity.Web are now the recommended auth libraries to use with the Microsoft identity platform. No new features will be implemented on ADAL.NET. The efforts are focused on improving MSAL.NET. For details see the announcement: [Update your applications to use Microsoft Authentication Library and Microsoft Graph API](#)

Should you migrate to MSAL.NET or to Microsoft.Identity.Web

Before digging in the details of MSAL.NET vs ADAL.NET, you might want to check if you want to use MSAL.NET or a higher-level abstraction like [Microsoft.Identity.Web](#)

For details about the decision tree below, read [MSAL.NET or Microsoft.Identity.Web?](#)



* Microsoft Identity Web provides a higher-level API on top of MSAL.NET

Next steps

- Learn about [public client and confidential client applications](#).
- Learn how to [migrate confidential client applications built on top of ASP.NET MVC or .NET classic from ADAL.NET to MSAL.NET](#).
- Learn how to [migrate public client applications built on top of .NET or .NET classic from ADAL.NET to MSAL.NET](#).
- Learn more about the [Differences between ADAL.NET and MSAL.NET apps](#).
- Learn how to migrate confidential client applications built on top of ASP.NET Core from ADAL.NET to `Microsoft.Identity.Web`:
 - [Web apps](#)
 - [Web APIs](#)

Migrate confidential client applications from ADAL.NET to MSAL.NET

4/12/2022 • 12 minutes to read • [Edit Online](#)

This article describes how to migrate a confidential client application from Azure Active Directory Authentication Library for .NET (ADAL.NET) to Microsoft Authentication Library for .NET (MSAL.NET). Confidential client applications are web apps, web APIs, and daemon applications that call another service on their own behalf. For more information about confidential applications, see [Authentication flows and application scenarios](#). If your app is based on ASP.NET Core, use [Microsoft.Identity.Web](#).

For app registrations:

- You don't need to create a new app registration. (You keep the same client ID.)
- You don't need to change the preauthorizations (admin-consented API permissions).

Migration steps

1. Find the code by using ADAL.NET in your app.

The code that uses ADAL in a confidential client application instantiates `AuthenticationContext` and calls either `AcquireTokenByAuthorizationCode` or one override of `AcquireTokenAsync` with the following parameters:

- A `resourceId` string. This variable is the app ID URI of the web API that you want to call.
- An instance of `IClientAssertionCertificate` or `ClientAssertion`. This instance provides the client credentials for your app to prove the identity of your app.

2. After you've identified that you have apps that are using ADAL.NET, install the MSAL.NET NuGet package [Microsoft.Identity.Client](#) and update your project library references. For more information, see [Install a NuGet package](#). If you want to use token cache serializers, also install [Microsoft.Identity.Web.TokenCache](#).
3. Update the code according to the confidential client scenario. Some steps are common and apply across all the confidential client scenarios. Other steps are unique to each scenario.

The confidential client scenarios are:

- [Daemon scenarios](#) supported by web apps, web APIs, and daemon console applications.
- [Web API calling downstream web APIs](#) supported by web APIs calling downstream web APIs on behalf of the user.
- [Web app calling web APIs](#) supported by web apps that sign in users and call a downstream web API.

You might have provided a wrapper around ADAL.NET to handle certificates and caching. This article uses the same approach to illustrate the process of migrating from ADAL.NET to MSAL.NET. However, this code is only for demonstration purposes. Don't copy/paste these wrappers or integrate them in your code as they are.

- [Daemon](#)
- [Web API calling downstream web APIs](#)
- [Web app calling web APIs](#)

Migrate daemon apps

Daemon scenarios use the OAuth2.0 [client credential flow](#). They're also called service-to-service calls. Your app

acquires a token on its own behalf, not on behalf of a user.

Find out if your code uses daemon scenarios

The ADAL code for your app uses daemon scenarios if it contains a call to `AuthenticationContext.AcquireTokenAsync` with the following parameters:

- A resource (app ID URI) as a first parameter
- `IClientAssertionCertificate` or `ClientAssertion` as the second parameter

`AuthenticationContext.AcquireTokenAsync` doesn't have a parameter of type `UserAssertion`. If it does, then your app is a web API, and it's using the [web API calling downstream web APIs](#) scenario.

Update the code of daemon scenarios

The following steps for updating code apply across all the confidential client scenarios:

1. Add the MSAL.NET namespace in your source code: `using Microsoft.Identity.Client;`.
2. Instead of instantiating `AuthenticationContext`, use `ConfidentialClientApplicationBuilder.Create` to instantiate `IConfidentialClientApplication`.
3. Instead of the `resourceId` string, MSAL.NET uses scopes. Because applications that use ADAL.NET are preauthorized, you can always use the following scopes: `new string[] { $"{resourceId}/.default" }`.
4. Replace the call to `AuthenticationContext.AcquireTokenAsync` with a call to `IConfidentialClientApplication.AcquireTokenXXX`, where `XXX` depends on your scenario.

In this case, we replace the call to `AuthenticationContext.AcquireTokenAsync` with a call to `IConfidentialClientApplication.AcquireTokenClient`.

Here's a comparison of ADAL.NET and MSAL.NET code for daemon scenarios:

ADAL

MSAL

```
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using System.Security.Cryptography.X509Certificates;
using System.Threading.Tasks;

public partial class AuthWrapper
{
    const string ClientId = "Guid (AppID)";
    const string authority
        = "https://login.microsoftonline.com/{tenant}";
    // App ID URI of web API to call
    const string resourceId = "https://target-api.domain.com";
    X509Certificate2 certificate = LoadCertificate();

    public async Task<AuthenticationResult> GetAuthenticationResult()
    {
        var authContext = new AuthenticationContext(authority);
        var clientAssertionCert = new ClientAssertionCertificate(
            ClientId,
            certificate);

        var authResult = await authContext.AcquireTokenAsync(
            resourceId,
            clientAssertionCert,
            );
        return authResult;
    }
}
```

```

using Microsoft.Identity.Client;
using System.Security.Cryptography.X509Certificates;
using System.Threading.Tasks;

public partial class AuthWrapper
{
    const string ClientId = "Guid (Application ID)";
    const string authority
        = "https://login.microsoftonline.com/{tenant}";
    // App ID URI of web API to call
    const string resourceId = "https://target-api.domain.com";
    X509Certificate2 certificate = LoadCertificate();

    IConfidentialClientApplication app;

    public async Task<AuthenticationResult> GetAuthenticationResult()
    {
        if (app == null)
        {
            app = ConfidentialClientApplicationBuilder.Create(ClientId)
                .WithCertificate(certificate)
                .WithAuthority(authority)
                .Build();
        }

        var authResult = await app.AcquireTokenForClient(
            new [] { $"{resourceId}/.default" })
            // .WithTenantId(specificTenant)
            // See https://aka.ms/msal.net/withTenantId
            .ExecuteAsync()
            .ConfigureAwait(false);

        return authResult;
    }
}

```

Benefit from token caching

To benefit from the in-memory cache, the instance of `IConfidentialClientApplication` needs to be kept in a member variable. If you re-create the confidential client application each time you request a token, you won't benefit from the token cache.

You'll need to serialize `AppTokenCache` if you choose not to use the default in-memory app token cache. Similarly, if you want to implement a distributed token cache, you'll need to serialize `AppTokenCache`. For details, see [Token cache for a web app or web API \(confidential client application\)](#) and the sample [active-directory-dotnet-v1-to-v2/ConfidentialClientTokenCache](#).

[Learn more about the daemon scenario](#) and how it's implemented with MSAL.NET or Microsoft.Identity.Web in new applications.

MSAL benefits

Key benefits of MSAL.NET for your app include:

- **Resilience.** MSAL.NET helps make your app resilient through the following:
 - Azure AD Cached Credential Service (CCS) benefits. CCS operates as an Azure AD backup.
 - Proactive renewal of tokens if the API that you call enables long-lived tokens through [continuous access evaluation](#).
- **Security.** You can acquire Proof of Possession (PoP) tokens if the web API that you want to call requires it. For details, see [Proof Of Possession tokens in MSAL.NET](#)

- **Performance and scalability.** If you don't need to share your cache with ADAL.NET, disable the legacy cache compatibility when you're creating the confidential client application (`.WithLegacyCacheCompatibility(false)`). This increases the performance significantly.

```
app = ConfidentialClientApplicationBuilder.Create(ClientId)
    .WithCertificate(certificate)
    .WithAuthority(authority)
    .WithLegacyCacheCompatibility(false)
    .Build();
```

Troubleshooting

MsalServiceException

The following troubleshooting information makes two assumptions:

- Your ADAL.NET code was working.
- You migrated to MSAL by keeping the same client ID.

If you get an exception with either of the following messages:

`AADSTS700027: Client assertion contains an invalid signature. [Reason - The key was not found.]`

`AADSTS90002: Tenant 'cf61953b-e41a-46b3-b500-663d279ea744' not found. This may happen if there are no active subscriptions for the tenant. Check to make sure you have the correct tenant ID. Check with your administrator.`

You can troubleshoot the exception by using these steps:

1. Confirm that you're using the latest version of [MSAL.NET](#).
2. Confirm that the authority host that you set when building the confidential client application and the authority host that you used with ADAL are similar. In particular, is it the same [cloud](#) (Azure Government, Azure China 21Vianet, or Azure Germany)?

MsalClientException

In multi-tenant applications, you can have scenarios where you specify a common authority when building the application, but then want to target a specific tenant (for instance the tenant of the user) when calling a web API. Since MSAL.NET 4.37.0, when you specify `.WithAzureRegion` at the application creation, you can no longer specify the Authority using `.WithAuthority` during the token requests. If you do, you'll get the following error when updating from previous versions of MSAL.NET:

`MsalClientException - "You configured WithAuthority at the request level, and also WithAzureRegion. This is not supported when the environment changes from application to request. Use WithTenantId at the request level instead."`

To remediate this issue, replace `.WithAuthority` on the `AcquireTokenXXX` expression by `.WithTenantId`. Specify the tenant using either a GUID or a domain name.

Next steps

Learn more about:

- [differences between ADAL.NET and MSAL.NET apps.](#)
- [token cache serialization in MSAL.NET](#)

Migrate public client applications from ADAL.NET to MSAL.NET

4/12/2022 • 13 minutes to read • [Edit Online](#)

This article describes how to migrate a public client application from Azure Active Directory Authentication Library for .NET (ADAL.NET) to Microsoft Authentication Library for .NET (MSAL.NET). Public client applications are desktop apps, including Win32, WPF, and UWP apps, and mobile apps, that call another service on the user's behalf. For more information about public client applications, see [Authentication flows and application scenarios](#).

Migration steps

1. Find the code by using ADAL.NET in your app.

The code that uses ADAL in a public client application instantiates `AuthenticationContext` and calls an override of `AcquireTokenAsync` with the following parameters:

- A `resourceId` string. This variable is the app ID URI of the web API that you want to call.
- A `clientId` which is the identifier for your application, also known as App ID.

2. After you've identified that you have apps that are using ADAL.NET, install the MSAL.NET NuGet package [Microsoft.Identity.Client](#) and update your project library references. For more information, see [Install a NuGet package](#).
3. Update the code according to the public client application scenario. Some steps are common and apply across all the public client scenarios. Other steps are unique to each scenario.

The public client scenarios are:

- [Web Authentication Manager](#) the preferred broker-based authentication on Windows.
 - [Interactive authentication](#) where the user is shown a web-based interface to complete the sign-in process.
 - [Integrated Windows authentication](#) where a user signs using the same identity they used to sign into a Windows domain (for domain-joined or Azure AD-joined machines).
 - [Username/password](#) where the sign-in occurs by providing a username/password credential.
 - [Device code flow](#) where a device of limited UX shows you a device code to complete the authentication flow on an alternate device.
-
- [Interactive](#)
 - [Integrated Windows authentication](#)
 - [Username Password](#)
 - [Device Code](#)

Interactive scenarios are where your public client application shows a login user interface hosted in a browser, and the user is required to interactively sign-in.

Find out if your code uses interactive scenarios

The ADAL code for your app in a public client application that uses interactive authentication instantiates `AuthenticationContext` and includes a call to `AcquireTokenAsync`, with the following parameters.

- A `clientId` which is a GUID representing your application registration
- A `resourceUrl` which indicates the resource you are asking the token for

- A URI that is the reply URL
- A `PlatformParameters` object.

Update the code for interactive scenarios

The following steps for updating code apply across all the confidential client scenarios:

1. Add the MSAL.NET namespace in your source code: `using Microsoft.Identity.Client;`.
2. Instead of instantiating `AuthenticationContext`, use `PublicClientApplicationBuilder.Create` to instantiate `IPublicClientApplication`.
3. Instead of the `resourceId` string, MSAL.NET uses scopes. Because applications that use ADAL.NET are preauthorized, you can always use the following scopes: `new string[] { $"{resourceId}/.default" }`.
4. Replace the call to `AuthenticationContext.AcquireTokenAsync` with a call to `IPublicClientApplication.AcquireTokenXXX`, where `XXX` depends on your scenario.

In this case, we replace the call to `AuthenticationContext.AcquireTokenAsync` with a call to `IPublicClientApplication.AcquireTokenInteractive`.

Here's a comparison of ADAL.NET and MSAL.NET code for interactive scenarios:

ADAL

MSAL

```
var ac = new AuthenticationContext("https://login.microsoftonline.com/<tenantId>");  
AuthenticationResult result;  
result = await ac.AcquireTokenAsync("<clientId>",  
"https://resourceUrl",  
new Uri("https://ClientReplyUrl"),  
new PlatformParameters(PromptBehavior.Auto));
```

```

// 1. Configuration - read below about redirect URI
var pca = PublicClientApplicationBuilder.Create("client_id")
    .WithBroker()
    .Build();

// Add a token cache, see https://docs.microsoft.com/en-us/azure/active-directory/develop/msal-net-token-cache-serialization?tabs=desktop

// 2. GetAccounts
var accounts = await pca.GetAccountsAsync();
var accountToLogin = // choose an account, or null, or use PublicClientApplication.OperatingSystemAccount
for the default OS account

try
{
// 3. AcquireTokenSilent
var authResult = await pca.AcquireTokenSilent(new[] { "User.Read" }, accountToLogin)
    .ExecuteAsync();
}
catch (MsalUiRequiredException) // no change in the pattern
{
// 4. Specific: Switch to the UI thread for next call . Not required for console apps.
await SwitchToUiThreadAsync(); // not actual code, this is different on each platform / tech

// 5. AcquireTokenInteractive
var authResult = await pca.AcquireTokenInteractive(new[] { "User.Read" })
    .WithAccount(accountToLogin) // this already exists in MSAL, but it is more important for WAM
    .WithParentActivityOrWindow(myWindowHandle) // to be able to parent WAM's windows to your app (optional, but
highly recommended; not needed on UWP)
    .ExecuteAsync();
}

```

The MSAL code shown above uses WAM (Web authentication manager) which is the recommended approach. If you wish to use interactive authentication without WAM, see [Interactive Authentication](#).

MSAL benefits

Key benefits of MSAL.NET for your app include:

- **Resilience.** MSAL.NET helps make your app resilient through the following:
 - Azure AD Cached Credential Service (CCS) benefits. CCS operates as an Azure AD backup.
 - Proactive renewal of tokens if the API that you call enables long-lived tokens through [continuous access evaluation](#).

Troubleshooting

The following troubleshooting information makes two assumptions:

- Your ADAL.NET code was working.
- You migrated to MSAL by keeping the same client ID.

If you get an exception with either of the following messages:

AADSTS90002: Tenant 'cf61953b-e41a-46b3-b500-663d279ea744' not found. This may happen if there are no active
subscriptions for the tenant. Check to make sure you have the correct tenant ID. Check with your subscription
administrator.

You can troubleshoot the exception by using these steps:

1. Confirm that you're using the latest version of MSAL.NET.

2. Confirm that the authority host that you set when building the confidential client application and the authority host that you used with ADAL are similar. In particular, is it the same [cloud](#) (Azure Government, Azure China 21Vianet, or Azure Germany)?

Next steps

Learn more about the [differences between ADAL.NET and MSAL.NET apps](#). Learn more about [token cache serialization in MSAL.NET](#)

Differences between ADAL.NET and MSAL.NET apps

4/12/2022 • 11 minutes to read • [Edit Online](#)

Migrating your applications from using ADAL to using MSAL comes with security and resiliency benefits. This article outlines differences between MSAL.NET and ADAL.NET. In most cases you want to use MSAL.NET and the Microsoft identity platform, 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're already familiar with ADAL.NET and the Azure AD for developers (v1.0) endpoint, get to know [what's different about the Microsoft identity platform?](#). 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](#).

	ADAL.NET	MSAL.NET
NuGet packages and Namespaces	ADAL was consumed from the Microsoft.IdentityModel.Clients.ActiveDirectory NuGet package. The namespace was <code>Microsoft.IdentityModel.Clients.ActiveDirectory</code> .	Add the Microsoft.Identity.Client NuGet package, and use the <code>Microsoft.Identity.Client</code> namespace. If you're building a <code>ConfidentialClientApplication</code> , check out Microsoft.Identity.Web .
Scopes and resources	ADAL.NET acquires tokens for <i>resources</i> .	MSAL.NET acquires tokens for <i>scopes</i> . Several MSAL.NET <code>AcquireTokenXXX</code> overrides require a parameter called <code>scopes(IEnumerable<string> scopes)</code> . This parameter is a simple list of strings that declare the permissions and resources that are requested. Well-known scopes are the Microsoft Graph's scopes . You can also access v1.0 resources using MSAL.NET .
Core classes	ADAL.NET used AuthenticationContext as the representation of your connection to the Security Token Service (STS) or authorization server, through an Authority.	MSAL.NET is designed around client applications . It defines <code>IPublicClientApplication</code> interfaces for public client applications and <code>IConfidentialClientApplication</code> for confidential client applications, as well as a base interface <code>IClientApplicationBase</code> for the contract common to both types of applications.

	ADAL.NET	MSAL.NET
Token acquisition	In public clients, ADAL uses <code>AcquireTokenAsync</code> and <code>AcquireTokenSilentAsync</code> for authentication calls.	In public clients, MSAL uses <code>AcquireTokenInteractive</code> and <code>AcquireTokenSilent</code> for the same authentication calls. The parameters are different from the ADAL ones. In Confidential client applications, there are token acquisition methods with an explicit name depending on the scenario. Another difference is that, in MSAL.NET, you no longer have to pass in the <code>ClientID</code> of your application in every <code>AcquireTokenXX</code> call. The <code>ClientID</code> is set only once when building <code>IPublicClientApplication</code> or <code>IConfidentialClientApplication</code> .
IAccount and IUser	ADAL defines the notion of user through the <code>IUser</code> interface. However, a user is a human or a software agent. As such, a user can own one or more accounts in the Microsoft identity platform (several Azure AD accounts, Azure AD B2C, Microsoft personal accounts). The user can also be responsible for one or more Microsoft identity platform accounts.	MSAL.NET defines the concept of account (through the <code>IAccount</code> interface). The <code>IAccount</code> interface represents information about a single account. The user can have several accounts in different tenants. MSAL.NET provides better information in guest scenarios, as home account information is provided. You can read more about the differences between <code>IUser</code> and <code>IAccount</code> .
Cache persistence	ADAL.NET allows you to extend the <code>TokenCache</code> class to implement the desired persistence functionality on platforms without a secure storage (.NET Framework and .NET core) by using the <code>BeforeAccess</code> , and <code>BeforeWrite</code> 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. As such, 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 article. The serialization for a public client application (See token cache for a public client application), is different from that of for a confidential client application (See token cache for a web app or web API).
Common authority	ADAL uses Azure AD v1.0. https://login.microsoftonline.com/common authority in Azure AD v1.0 (which ADAL uses) allows users to sign in using any AAD organization (work or school) account. Azure AD v1.0 doesn't allow sign in with Microsoft personal accounts. For more information, see authority validation in ADAL.NET .	MSAL uses Azure AD v2.0. https://login.microsoftonline.com/organizations authority in Azure AD v2.0 (which MSAL uses) allows users to sign in with any AAD organization (work or school) account or with a Microsoft personal account. To restrict sign in using only organization accounts (work or school account) in MSAL, you'll need to use the <code>https://login.microsoftonline.com/organizations</code> endpoint. For details, see the <code>authority</code> parameter in public client application .

Supported grants

Below is a summary comparing MSAL.NET and ADAL.NET supported grants for both public and confidential client applications.

Public client applications

The following image summarizes some of the differences between ADAL.NET and MSAL.NET for a public client application.

```

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); ADAL.Net
}

using Microsoft.Identity.Client; Different namespace
string[] scopes = { "User.Read" }; Scopes instead of a resource
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); MSAL.Net
}

```

The diagram highlights several differences between the two libraries:

- Different namespace:** MSAL.NET uses `Microsoft.Identity.Client` while ADAL.NET uses `Microsoft.IdentityModel.Clients.ActiveDirectory`.
- Scopes instead of a resource:** MSAL.NET uses `scopes` (array of strings) instead of `resource` (GUID or AppID URI).
- Implementation:** MSAL.NET uses `IPublicClientApplication` and `PublicClientApplicationBuilder`, while ADAL.NET uses `AuthenticationContext`. A note indicates that `PublicClientApplicationBuilder` and `ConfidentialClientApplicationBuilder` implement the fluent API instead of `AuthenticationContext`.
- No need to pass the clientId at every Token acquisition:** MSAL.NET handles the client ID automatically through the fluent API.
- More explicit exceptions:** MSAL.NET provides more specific exception types like `MsalUiRequiredException` and `MsalException` compared to the general `AdalException` in ADAL.NET.

Here are the grants supported in ADAL.NET and MSAL.NET for Desktop and Mobile applications.

GRANT	MSAL.NET	ADAL.NET
Interactive	Acquiring tokens interactively in MSAL.NET	Interactive Auth
Integrated Windows authentication	Integrated Windows authentication	Integrated authentication on Windows (Kerberos)
Username / Password	Username-password authentication	Acquiring tokens with username and password
Device code flow	Device code flow	Device profile for devices without web browsers

Confidential client applications

The following image summarizes some of the differences between ADAL.NET and MSAL.NET for a confidential client application.

```

using Microsoft.IdentityModel.Clients.ActiveDirectory;
// GUID OR App URI
const string resource = "https://api.example.com/";
AuthenticationContext app =
    new AuthenticationContext(authority);

var certCred = new ClientAssertionCertificate(client_id, cert);

AuthenticationResult result =
    await authenticationContext.AcquireTokenAsync(resource,
        certCred);

var result = await cca.AcquireTokenForClient(scopes).ExecuteAsync(); MSAL.Net
}

using Microsoft.Identity.Client; Different namespace
// use static scopes listed in app registration (like ADAL)
string[] scopes = { "https://api.example.com/.default" };
string region =
    Config.Region ?? // Always try to provide the region
    "TryAutoDetect"; // If app does not know the region, let MSAL try

var cca = ConfidentialClientApplicationBuilder(client_id)
    .WithRegion(region)
    .WithCertificate(x509cert)
    .Build();

var result = await cca.AcquireTokenForClient(scopes).ExecuteAsync(); MSAL.Net

```

The diagram highlights several differences between the two libraries:

- Different namespace:** MSAL.NET uses `Microsoft.Identity.Client` while ADAL.NET uses `Microsoft.IdentityModel.Clients.ActiveDirectory`.
- Scopes:** MSAL.NET uses static scopes defined in the app registration (e.g., `https://api.example.com/.default`), while ADAL.NET uses dynamic scopes (`resource`).
- Region:** MSAL.NET allows specifying the region (e.g., `Config.Region ?? "TryAutoDetect"`) to handle regional differences.
- Implementation:** MSAL.NET uses `ConfidentialClientApplicationBuilder` instead of `AuthenticationContext`.

Here are the grants supported in ADAL.NET, MSAL.NET, and Microsoft.Identity.Web for web applications, web APIs, and daemon applications.

Type of App	Grant	MSAL.NET	ADAL.NET
Web app, web API, daemon	Client Credentials	Client credential flows in MSAL.NET	Client credential flows in ADAL.NET
Web API	On behalf of	On behalf of in MSAL.NET	Service to service calls on behalf of the user with ADAL.NET
Web app	Auth Code	Acquiring tokens with authorization codes on web apps with A MSAL.NET	Acquiring tokens with authorization codes on web apps with ADAL.NET

Migrating from ADAL 2.x with refresh tokens

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 for the users when the users are no longer connected / signed-in to the app.
- WebFarm scenarios for enabling the client to bring the refresh token to the web service (caching is done client side, encrypted cookie, and not server side).

MSAL.NET doesn't expose refresh tokens for security reasons. MSAL handles refreshing tokens for you.

Fortunately, MSAL.NET 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 want. The refresh token will be exchanged for a new one and cached into your application.

As this method is intended for scenarios that aren't typical, it isn't readily accessible with the `IConfidentialClientApplication` without first casting it to `IByRefreshToken`.

The code snippet below shows some migration code in a confidential client application.

```

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);

```

`GetCachedRefreshTokenForSignedInUser` retrieves the refresh token that was stored in some storage by a previous version of the application that used to use ADAL 2.x. `GetTokenCacheForSignedInUser` deserializes a cache for the signed-in user (as confidential client applications should have one cache per user).

An access token and an ID token are returned in the `AuthenticationResult` value while the 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.

v1.0 and v2.0 tokens

There are two versions of tokens: v1.0 tokens and v2.0 tokens. The v1.0 endpoint (used by ADAL) emits v1.0 ID tokens while the v2.0 endpoint (used by MSAL) emits v2.0 ID tokens. However, both endpoints emit access tokens of 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 access tokens, see [Azure Active Directory access tokens](#).

Exceptions

Interaction required exceptions

Using MSAL.NET, you catch `MsalUiRequiredException` as described in [AcquireTokenSilent](#).

```

catch(MsalUiRequiredException exception)
{
    try {"try to authenticate interactively"}
}

```

For details, see [Handle errors and exceptions in MSAL.NET](#)

ADAL.NET had less explicit exceptions. For example, when silent authentication failed in ADAL the procedure was 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"}
    }
}

```

For details, see [the recommended pattern to acquire a token in public client applications with ADAL.NET](#).

Prompt behavior

Prompt behavior in MSAL.NET is equivalent to prompt behavior in ADAL.NET:

ADAL.NET	MSAL.NET	DESCRIPTION
<code>PromptBehavior.Auto</code>	<code>NoPrompt</code>	Azure AD chooses the best behavior (signing in users silently if they are signed in with only one account, or displaying the account selector if they are signed in with several accounts).
<code>PromptBehavior.Always</code>	<code>ForceLogin</code>	Resets the sign-in box and forces the user to reenter their credentials.
<code>PromptBehavior.RefreshSession</code>	<code>Consent</code>	Forces the user to consent again to all permissions.
<code>PromptBehavior.Never</code>	<code>Never</code>	Don't use; instead, use the recommended pattern for public client apps .
<code>PromptBehavior.SelectAccount</code>	<code>SelectAccount</code>	Displays the account selector and forces the user to select an account.

Handling claim challenge exceptions

At times when acquiring a token, Azure AD throws an exception in case a resource requires more claims from the user (for instance two-factor authentication).

In MSAL.NET, claim challenge exceptions are handled in the following way:

- The `Claims` are surfaced in the `MsalServiceException`.
- There's a `.WithClaim(claims)` method that can apply to the `AcquireTokenXXX` builders.

For details see [Handling MsalUiRequiredException](#).

In ADAL.NET, claim challenge exceptions were handled in the following way:

- `AdalClaimChallengeException` is an exception (deriving from `AdalServiceException`). The `Claims` member contains some JSON fragment with the claims, which are expected.
- The public client application receiving this exception needed to call the `AcquireTokenInteractive` override having a claims parameter. This override of `AcquireTokenInteractive` doesn't even try to hit the cache as it isn't necessary. The reason is that the token in the cache doesn't have the right claims (otherwise an `AdalClaimChallengeException` wouldn't have been thrown). As such, there's no need to look at the cache. The `ClaimChallengeException` can be received in a WebAPI doing OBO, but the `AcquireTokenInteractive` needs to be called in a public client application calling this web API.

For details, including samples see [handling AdalClaimChallengeException](#).

Scopes

ADAL uses the concept of resources with `resourceId` string, MSAL.NET, however, uses scopes. The logic used by Azure AD is as follows:

- 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, Azure AD parses the desired audience from the requested scope. This is done by taking everything before the last slash and using it as the resource identifier. As such, 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`

(notice the double slash before ./default). This is illustrated by examples 1 and 2 below.

Example 1

If you want to acquire tokens for an application accepting v1.0 tokens (for instance the Microsoft Graph API, which is <https://graph.microsoft.com>), you'd need to create `scopes` by concatenating a desired resource identifier with a desired OAuth2 permission for that resource.

For instance, to access the name of the user via a v1.0 web API whose 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 Microsoft Graph API (<https://graph.microsoft.com/>), you'd create a list of scopes like in the code snippet below:

```
string ResourceId = "https://graph.microsoft.com/";
string[] scopes = { ResourceId + "Directory.Read", ResourceId + "Directory.Write" }
```

Example 2

If `resourceId` ends with a '/', you'll need to have a double '/' when writing the scope value. For example, if you want to write the scope corresponding to the Azure Resource Manager API (<https://management.core.windows.net/>), request the following scope (note the two slashes).

```
var resource = "https://management.core.windows.net/"
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's a slash to separate the API name from the scope.

If you want to acquire a token for all the static scopes of a v1.0 application, you'd create your scopes list as shown in the code snippet below:

```
ResourceId = "someAppIDURI";
var scopes = new [] { ResourceId + ".default" };
```

For a 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."

Next steps

[Migrate your apps from ADAL to MSAL](#) [Migrate your ADAL.NET confidential client apps to use MSAL.NET](#)

How to migrate a Node.js app from ADAL to MSAL

4/12/2022 • 12 minutes to read • [Edit Online](#)

Microsoft Authentication Library for Node (MSAL Node) is now the recommended SDK for enabling authentication and authorization for your applications registered on the Microsoft identity platform. This article covers the important steps you need to go through in order to migrate your apps from Active Directory Authentication Library for Node (ADAL Node) to MSAL Node.

Prerequisites

- Node version 10, 12 or 14. See the [note on version support](#)

Update app registration settings

When working with ADAL Node, you were likely using the [Azure AD v1.0 endpoint](#). Apps migrating from ADAL to MSAL should also consider switching to [Azure AD v2.0 endpoint](#).

1. Review the [differences between v1 and v2 endpoints](#)
2. Update, if necessary, your existing app registrations accordingly.

NOTE

In order to ensure backward compatibility, MSAL Node supports both v1.0 and v2.0 endpoints.

Install and import MSAL

1. install MSAL Node package via NPM:

```
npm install @azure/msal-node
```

1. After that, import MSAL Node in your code:

```
const msal = require('@azure/msal-node');
```

1. Finally, uninstall the ADAL Node package and remove any references in your code:

```
npm uninstall adal-node
```

Initialize MSAL

In ADAL Node, you initialize an `AuthenticationContext` object, which then exposes the methods you can use in different authentication flows (e.g. `acquireTokenWithAuthorizationCode` for web apps). When initializing, the only mandatory parameter is the **authority URI**:

```
var adal = require('adal-node');

var authorityURI = "https://login.microsoftonline.com/common";
var authenticationContext = new adal.AuthenticationContext(authorityURI);
```

In MSAL Node, you have two alternatives instead: If you are building a mobile app or a desktop app, you instantiate a `PublicClientApplication` object. The constructor expects a [configuration object](#) that contains the `clientId` parameter at the very least. MSAL defaults the authority URI to `https://login.microsoftonline.com/common` if you do not specify it.

```
const msal = require('@azure/msal-node');

const pca = new msal.PublicClientApplication({
    auth: {
        clientId: "YOUR_CLIENT_ID"
    }
});
```

NOTE

If you use the `https://login.microsoftonline.com/common` authority in v2.0, you will allow users to sign in with any Azure AD organization or a personal Microsoft account (MSA). In MSAL Node, if you want to restrict login to any Azure AD account (same behavior as with ADAL Node), use `https://login.microsoftonline.com/organizations` instead.

On the other hand, if you are building a web app or a daemon app, you instantiate a `ConfidentialClientApplication` object. With such apps you also need to supply a *client credential*, such as a client secret or a certificate:

```
const msal = require('@azure/msal-node');

const cca = new msal.ConfidentialClientApplication({
    auth: {
        clientId: "YOUR_CLIENT_ID",
        clientSecret: "YOUR_CLIENT_SECRET"
    }
});
```

Both `PublicClientApplication` and `ConfidentialClientApplication`, unlike ADAL's `AuthenticationContext`, is bound to a client ID. This means that if you have different client IDs that you like to use in your application, you need to instantiate a new MSAL instance for each. See for more: [Initialization of MSAL Node](#)

Configure MSAL

When building apps on Microsoft identity platform, your app will contain many parameters related to authentication. In ADAL Node, the `AuthenticationContext` object has a limited number of configuration parameters that you can instantiate it with, while the remaining parameters hang freely in your code (e.g. `clientSecret`):

```

var adal = require('adal-node');

var authority = "https://login.microsoftonline.com/YOUR_TENANT_ID"
var validateAuthority = true,
var cache = null;

var authenticationContext = new adal.AuthenticationContext(authority, validateAuthority, cache);

```

- `authority` : URL that identifies a token authority
- `validateAuthority` : a feature that prevents your code from requesting tokens from a potentially malicious authority
- `cache` : sets the token cache used by this AuthenticationContext instance. If this parameter is not set, then a default, in memory cache is used

MSAL Node on the other hand uses a configuration object of type [Configuration](#). It contains the following properties:

```

const msal = require('@azure/msal-node');

const msalConfig = {
    auth: {
        clientId: "YOUR_CLIENT_ID",
        authority: "https://login.microsoftonline.com/YOUR_TENANT_ID",
        clientSecret: "YOUR_CLIENT_SECRET",
        knownAuthorities: [],
    },
    cache: {
        // your implementation of caching
    },
    system: {
        loggerOptions: { /** logging related options */ }
    }
}

const cca = new msal.ConfidentialClientApplication(msalConfig);

```

As a notable difference, MSAL does not have a flag to disable authority validation and authorities are always validated by default. MSAL compares your requested authority against a list of authorities known to Microsoft or a list of authorities you've specified in your configuration. See for more: [Configuration Options](#)

Switch to MSAL API

Most of the public methods in ADAL Node have equivalents in MSAL Node:

ADAL	MSAL	NOTES
<code>acquireToken</code>	<code>acquireTokenSilent</code>	Renamed and now expects an account object
<code>acquireTokenWithAuthorizationCode</code>	<code>acquireTokenByCode</code>	
<code>acquireTokenWithClientCredentials</code>	<code>acquireTokenByClientCredential</code>	
<code>acquireTokenWithRefreshToken</code>	<code>acquireTokenByRefreshToken</code>	Useful for migrating valid refresh tokens

ADAL	MSAL	NOTES
<code>acquireTokenWithDeviceCode</code>	<code>acquireTokenByDeviceCode</code>	Now abstracts user code acquisition (see below)
<code>acquireTokenByUsernamePassword</code>	<code>acquireTokenByUsernamePassword</code>	

However, some methods in ADAL Node are deprecated, while MSAL Node offers new methods:

ADAL	MSAL	NOTES
<code>acquireUserCode</code>	N/A	Merged with <code>acquireTokenByDeviceCode</code> (see above)
N/A	<code>acquireTokenOnBehalfOf</code>	A new method that abstracts OBO flow
<code>acquireTokenWithClientCertificate</code>	N/A	No longer needed as certificates are assigned during initialization now (see configuration options)
N/A	<code>getAuthCodeUrl</code>	A new method that abstracts authorize endpoint URL construction

Use scopes instead of resources

An important difference between v1.0 vs. v2.0 endpoints is about how the resources are accessed. In ADAL Node, you would first register a permission on app registration portal, and then request an access token for a resource (such as Microsoft Graph) as shown below:

```
authenticationContext.acquireTokenWithAuthorizationCode(
  req.query.code,
  redirectUri,
  resource, // e.g. 'https://graph.microsoft.com'
  clientId,
  clientSecret,
  function (err, response) {
    // do something with the authentication response
  }
);
```

MSAL Node supports both **v1.0** and **v2.0** endpoints. The v2.0 endpoint employs a *scope-centric* model to access resources. Thus, when you request an access token for a resource, you also need to specify the scope for that resource:

```

const tokenRequest = {
  code: req.query.code,
  scopes: ["https://graph.microsoft.com/User.Read"],
  redirectUri: REDIRECT_URI,
};

pca.acquireTokenByCode(tokenRequest).then((response) => {
  // do something with the authentication response
}).catch((error) => {
  console.log(error);
});

```

One advantage of the scope-centric model is the ability to use *dynamic scopes*. When building applications using v1.0, you needed to register the full set of permissions (called *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 (hence, *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. See for more: [Resources and scopes](#)

Use promises instead of callbacks

In ADAL Node, callbacks are used for any operation after the authentication succeeds and a response is obtained:

```

var context = new AuthenticationContext(authorityUrl, validateAuthority);

context.acquireTokenWithClientCredentials(resource, clientId, clientSecret, function(err, response) {
  if (err) {
    console.log(err);
  } else {
    // do something with the authentication response
  }
});

```

In MSAL Node, promises are used instead:

```

const cca = new msal.ConfidentialClientApplication(msalConfig);

cca.acquireTokenByClientCredential(tokenRequest).then((response) => {
  // do something with the authentication response
}).catch((error) => {
  console.log(error);
});

```

You can also use the **async/await** syntax that comes with ES8:

```

try {
  const authResponse = await cca.acquireTokenByCode(tokenRequest);
} catch (error) {
  console.log(error);
}

```

Enable logging

In ADAL Node, you configure logging separately at any place in your code:

```

var adal = require('adal-node');

//PII or OII logging disabled. Default Logger does not capture any PII or OII.
adal.logging.setLoggingOptions({
    log: function (level, message, error) {
        console.log(message);

        if (error) {
            console.log(error);
        }
    },
    level: logging.LOGGING_LEVEL.VERBOSE, // provide the logging level
    loggingWithPII: false // Determine if you want to log personal identification information. The default value is false.
});

```

In MSAL Node, logging is part of the configuration options and is created with the initialization of the MSAL Node instance:

```

const msal = require('@azure/msal-node');

const msalConfig = {
    auth: {
        // authentication related parameters
    },
    cache: {
        // cache related parameters
    },
    system: {
        loggerOptions: {
            loggerCallback(loglevel, message, containsPii) {
                console.log(message);
            },
            piiLoggingEnabled: false,
            logLevel: msal.LogLevel.Verbose,
        }
    }
}

const cca = new msal ConfidentialClientApplication(msalConfig);

```

Enable token caching

In ADAL Node, you had the option of importing an in-memory token cache. The token cache is used as a parameter when initializing an `AuthenticationContext` object:

```

var MemoryCache = require('adal-node/lib/memory-cache');

var cache = new MemoryCache();
var authorityURI = "https://login.microsoftonline.com/common";

var context = new AuthenticationContext(authorityURI, true, cache);

```

MSAL Node uses an in-memory token cache by default. You do not need to explicitly import it; in-memory token cache is exposed as part of the `ConfidentialClientApplication` and `PublicClientApplication` classes.

```
const msalTokenCache = publicClientApplication.getTokenCache();
```

Importantly, your previous token cache with ADAL Node will not be transferable to MSAL Node, since cache

schemas are incompatible. However, you may use the valid refresh tokens your app obtained previously with ADAL Node in MSAL Node. See the section on [refresh tokens](#) for more.

You can also write your cache to disk by providing your own **cache plugin**. The cache plugin must implement the interface [ICachePlugin](#). Like logging, caching is part of the configuration options and is created with the initialization of the MSAL Node instance:

```
const msal = require('@azure/msal-node');

const msalConfig = {
    auth: {
        // authentication related parameters
    },
    cache: {
        cachePlugin // your implementation of cache plugin
    },
    system: {
        // logging related options
    }
}

const msalInstance = new ConfidentialClientApplication(msalConfig);
```

An example cache plugin can be implemented as below:

```
const fs = require('fs');

// Call back APIs which automatically write and read into a .json file - example implementation
const beforeCacheAccess = async (cacheContext) => {
    cacheContext.tokenCache.deserialize(await fs.readFile(cachePath, "utf-8"));
};

const afterCacheAccess = async (cacheContext) => {
    if(cacheContext.cacheHasChanged) {
        await fs.writeFile(cachePath, cacheContext.tokenCache.serialize());
    }
};

// Cache Plugin
const cachePlugin = {
    beforeCacheAccess,
    afterCacheAccess
};
```

If you are developing [public client applications](#) like desktop apps, the [Microsoft Authentication Extensions for Node](#) offers secure mechanisms for client applications to perform cross-platform token cache serialization and persistence. Supported platforms are Windows, Mac and Linux.

NOTE

[Microsoft Authentication Extensions for Node](#) is **not** recommended for web applications, as it may lead to scale and performance issues. Instead, web apps are recommended to persist the cache in session.

Remove logic around refresh tokens

In ADAL Node, the refresh tokens (RT) were exposed allowing you to develop solutions around the use of these tokens by caching them and using the `acquireTokenWithRefreshToken` method. Typical scenarios where RTs are especially relevant:

- Long running services that do actions including refreshing dashboards on behalf of the users where 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 Node, along with other MSALs, does not expose refresh tokens for security reasons. Instead, MSAL handles refreshing tokens for you. As such, you no longer need to build logic for this. However, you **can** make use of your previously acquired (and still valid) refresh tokens from ADAL Node's cache to get a new set of tokens with MSAL Node. To do this, MSAL Node offers `acquireTokenByRefreshToken`, which is equivalent to ADAL Node's `acquireTokenWithRefreshToken` method:

```
var msal = require('@azure/msal-node');

const config = {
    auth: {
        clientId: "ENTER_CLIENT_ID",
        authority: "https://login.microsoftonline.com/ENTER_TENANT_ID",
        clientSecret: "ENTER_CLIENT_SECRET"
    }
};

const cca = new msal.ConfidentialClientApplication(config);

const refreshTokenRequest = {
    refreshToken: "", // your previous refresh token here
    scopes: ["user.read"],
};

cca.acquireTokenByRefreshToken(refreshTokenRequest).then((response) => {
    console.log(JSON.stringify(response));
}).catch((error) => {
    console.log(JSON.stringify(error));
});
```

NOTE

We recommend you to destroy the older ADAL Node token cache once you utilize the still valid refresh tokens to get a new set of tokens using the MSAL Node's `acquireTokenByRefreshToken` method as shown above.

Handle errors and exceptions

When using MSAL Node, the most common type of error you might face is the `interaction_required` error. This error is often resolved by simply initiating an interactive token acquisition prompt. For instance, when using `acquireTokenSilent`, if there are no cached refresh tokens, MSAL Node will not be able to acquire an access token silently. Similarly, the web API you are trying to access might have a `conditional access` policy in place, requiring the user to perform `multi-factor authentication` (MFA). In such cases, handling `interaction_required` error by triggering `acquireTokenByCode` will prompt the user for MFA, allowing them to fulfill it.

Yet another common error you might face is `consent_required`, which occurs when permissions required for obtaining an access token for a protected resource are not consented by the user. As in `interaction_required`, the solution for `consent_required` error is often initiating an interactive token acquisition prompt, using the `acquireTokenByCode` method.

Run the app

Once your changes are done, run the app and test your authentication scenario:

```
npm start
```

Example: Acquiring tokens with ADAL Node vs. MSAL Node

The snippet below demonstrates a confidential client web app in the Express.js framework. It performs a sign-in when a user hits the authentication route `/auth`, acquires an access token for Microsoft Graph via the `/redirect` route and then displays the content of the said token.

Using ADAL Node	Using MSAL Node
<pre>// Import dependencies var express = require('express'); var crypto = require('crypto'); var adal = require('adal-node'); // Authentication parameters var clientId = 'Enter_the_Application_Id_Here'; var clientSecret = 'Enter_the_Client_Secret_Here'; var tenant = 'common'; var authorityUrl = 'https://login.microsoftonline.com/' + tenant; var redirectUri = 'http://localhost:3000/redirect'; var resource = 'https://graph.microsoft.com'; // Configure logging adal.Logging.setLoggingOptions({ log: function (level, message, error) { console.log(message); }, level: adal.Logging.LOGGING_LEVEL.VERBOSE, loggingWithPII: false }); // Auth code request URL template var templateAuthzUrl = 'https://login.microsoftonline.com/' + tenant + '/oauth2/authorize? response_type=code&client_id=' + clientId + '&redirect_uri=' + redirectUri + '&state=<state>&resource=' + resource; // Initialize express var app = express(); // State variable persists throughout the app // lifetime app.locals.state = ""; app.get('/auth', function(req, res) { // Create a random string to use against // XSRF crypto.randomBytes(48, function(ex, buf) { app.locals.state = buf.toString('base64') .replace(/\\/g, '_') .replace(/\+/g, '-'); // Construct auth code request URL var authorizationUrl = templateAuthzUrl .replace('<state>', app.locals.state); res.redirect(authorizationUrl); }); });</pre>	

```

app.get('/redirect', function(req, res) {
    // Compare state parameter against XSRF
    if (app.locals.state !== req.query.state) {
        res.send('error: state does not
match');
    }

    // Initialize an AuthenticationContext
    object
    var authenticationContext =
        new
adal.AuthenticationContext(authorityUrl);

    // Exchange auth code for tokens

    authenticationContext.acquireTokenWithAuthoriza
tionCode(
    req.query.code,
    redirectUri,
    resource,
    clientId,
    clientSecret,
    function(err, response) {
        res.send(response);
    }
);
});

app.listen(3000, function() {
    console.log(`listening on port 3000!`);
});

```

```

// Import dependencies
const express = require("express");
const msal = require('@azure/msal-node');

// Authentication parameters
const config = {
    auth: {
        clientId:
"Enter_the_Application_Id_Here",
        authority:
"https://login.microsoftonline.com/common",
        clientSecret:
"Enter_the_Client_Secret_Here"
    },
    system: {
        loggerOptions: {
            loggerCallback(loglevel, message,
containsPii) {
                console.log(message);
            },
            piiLoggingEnabled: false,
            logLevel: msal.LogLevel.Verbose,
        }
    }
};

const REDIRECT_URI =
"http://localhost:3000/redirect";

// Initialize MSAL Node object using
authentication parameters
const cca = new
msal.ConfidentialClientApplication(config);

// Initialize express
const app = express();

app.get('/auth', (req, res) => {

    // Construct a request object for auth code
    const authCodeUrlParameters = {
        scopes: ["user.read"],
        redirectUri: REDIRECT_URI,
    };

    // Request auth code, then redirect
    cca.getAuthCodeUrl(authCodeUrlParameters)
        .then((response) => {
            res.redirect(response);
        }).catch((error) => res.send(error));
});

app.get('/redirect', (req, res) => {

    // Use the auth code in redirect request to
construct
    // a token request object
    const tokenRequest = {
        code: req.query.code,
        scopes: ["user.read"],
        redirectUri: REDIRECT_URI,
    };

    // Exchange the auth code for tokens
    cca.acquireTokenByCode(tokenRequest)
        .then((response) => {
            res.send(response);
        }).catch((error) =>
res.status(500).send(error));
});

app.listen(3000, () =>
    console.log(`listening on port 3000!`));

```

Next steps

- [MSAL Node API reference](#)
- [MSAL Node Code samples](#)

Microsoft Authentication Extensions for Node

4/12/2022 • 2 minutes to read • [Edit Online](#)

The Microsoft Authentication Extensions for Node enables developers to perform cross-platform token cache serialization and persistence to disk. It gives extra support to the Microsoft Authentication Library (MSAL) for Node.

The [MSAL for Node](#) supports an in-memory cache by default and provides the `ICachePlugin` interface to perform cache serialization, but doesn't provide a default way of storing the token cache to disk. The Microsoft Authentication Extensions for Node is the default implementation for persisting cache to disk across different platforms.

The Microsoft Authentication Extensions for Node support the following platforms:

- Windows - Data protection API (DPAPI) is used for protection.
- Mac - The Mac Keychain is used.
- Linux - LibSecret is used for storing to "Secret Service".

Installation

The `msal-node-extensions` package is available on Node Package Manager (NPM).

```
npm i @azure/msal-node-extensions --save
```

Configure the token cache

Here's an example of code that uses Microsoft Authentication Extensions for Node to configure the token cache.

```

const {
  DataProtectionScope,
  Environment,
  PersistenceCreator,
  PersistenceCachePlugin,
} = require("@azure/msal-node-extensions");

// You can use the helper functions provided through the Environment class to construct your cache path
// The helper functions provide consistent implementations across Windows, Mac and Linux.
const cachePath = path.join(Environment.getUserRootDirectory(), "./cache.json");

const persistenceConfiguration = {
  cachePath,
  dataProtectionScope: DataProtectionScope.CurrentUser,
  serviceName: "<SERVICE-NAME>",
  accountName: "<ACCOUNT-NAME>",
  usePlaintextFileOnLinux: false,
};

// The PersistenceCreator obfuscates a lot of the complexity by doing the following actions for you :-
// 1. Detects the environment the application is running on and initializes the right persistence instance
// for the environment.
// 2. Performs persistence validation for you.
// 3. Performs any fallbacks if necessary.
PersistenceCreator.createPersistence(persistenceConfiguration).then(
  async (persistence) => {
    const publicClientConfig = {
      auth: {
        clientId: "<CLIENT-ID>",
        authority: "<AUTHORITY>",
      },
      // This hooks up the cross-platform cache into MSAL
      cache: {
        cachePlugin: new PersistenceCachePlugin(persistence),
      },
    };

    const pca = new msal.PublicClientApplication(publicClientConfig);

    // Use the public client application as required...
  }
);

```

The following table provides an explanation for all the arguments for the persistence configuration.

FIELD NAME	DESCRIPTION	REQUIRED FOR
cachePath	The path to the lock file the library uses to synchronize the reads and the writes	Windows, Mac, and Linux
dataProtectionScope	Specifies the scope of the data protection on Windows either the current user or the local machine.	Windows
serviceName	Specifies the service name to be used on Mac and/or Linux	Mac and Linux
accountName	Specifies the account name to be used on Mac and/or Linux	Mac and Linux

FIELD NAME	DESCRIPTION	REQUIRED FOR
usePlaintextFileOnLinux	The flag to default to plain text on linux if LibSecret fails. Defaults to <code>false</code>	Linux

Next steps

For more information about Microsoft Authentication Extensions for Node and MSAL Node, see:

- [Microsoft Authentication Extensions for Node](#)
- [Microsoft Authentication Library for Node](#)

Custom token cache serialization in MSAL for Python

4/12/2022 • 2 minutes to read • [Edit Online](#)

In MSAL Python, an in-memory token cache that persists for the duration of the app session, is provided by default when you create an instance of [ClientApplication](#).

Serialization of the token cache, so that different sessions of your app can access it, is not provided "out of the box." That's because MSAL Python can be used in app types that don't have access to the file system--such as Web apps. To have a persistent token cache in a MSAL Python app, you must provide custom token cache serialization.

The strategies for serializing the token cache differ depending on whether you are writing a public client application (Desktop), or a confidential client application (web app, web API, or daemon app).

Token cache for a public client application

Public client applications run on a user's device and manage tokens for a single user. In this case, you could serialize the entire cache into a file. Remember to provide file locking if your app, or another app, can access the cache concurrently. For a simple example of how to serialize a token cache to a file without locking, see the example in the [SerializableTokenCache](#) class reference documentation.

Token cache for a Web app (confidential client application)

For web apps or web APIs, you might use the session, or a Redis cache, or a database to store the token cache. There should be one token cache per user (per account) so ensure that you serialize the token cache per account.

Next steps

See [ms-identity-python-webapp](#) for an example of how to use the token cache for a Windows or Linux Web app or web API. The example is for a web app that calls the Microsoft Graph API.

ADAL to MSAL migration guide for Python

4/12/2022 • 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).

You can learn more about MSAL and get started with an [overview of the Microsoft Authentication Library](#).

Difference highlights

ADAL works with the Azure Active Directory (Azure AD) 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 AD v1.0 in that it:

Supports:

- 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
- Is standards compatible with:
 - OAuth v2.0
 - OpenID Connect (OIDC)

See [What's different about the Microsoft identity platform?](#) for more details.

Scopes not resources

ADAL Python acquires tokens for resources, but MSAL Python acquires tokens for scopes. The API surface in MSAL Python does not have resource parameter anymore. You would need to provide scopes as a list of strings that declare the desired permissions and resources that are requested. To see some example of scopes, see [Microsoft Graph's scopes](#).

You can add the `/.default` scope suffix to the resource to help migrate your apps from the v1.0 endpoint (ADAL) to the Microsoft identity platform (MSAL). For example, for the resource value of `https://graph.microsoft.com`, the equivalent scope value is `https://graph.microsoft.com/.default`. If the resource is not in the URL form, but a resource ID of the form `xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxx`, you can still use the scope value as `xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxx/.default`.

For more details about the different types of scopes, refer [Permissions and consent in the Microsoft identity platform](#) and the [Scopes for a Web API accepting v1.0 tokens](#) articles.

Error handling

Azure Active Directory Authentication Library (ADAL) for Python uses the exception `AdalError` to indicate that there's been a problem. MSAL for Python typically uses error codes, instead. For more information, see [MSAL for Python error handling](#).

API changes

The following table lists an API in ADAL for Python, and the one to use in its place in MSAL for Python:

ADAL FOR PYTHON API	MSAL FOR PYTHON API
AuthenticationContext	PublicClientApplication or ConfidentialClientApplication
N/A	PublicClientApplication.acquire_token_interactive()
N/A	ConfidentialClientApplication.initiate_auth_code_flow()
acquire_token_with_authorization_code()	ConfidentialClientApplication.acquire_token_by_auth_code_flow()
acquire_token()	PublicClientApplication.acquire_token_silent() or ConfidentialClientApplication.acquire_token_silent()
acquire_token_with_refresh_token()	These two helpers are intended to be used during migration only: PublicClientApplication.acquire_token_by_refresh_token() or ConfidentialClientApplication.acquire_token_by_refresh_token()
acquire_user_code()	initiate_device_flow()
acquire_token_with_device_code() and cancel_request_to_get_token_with_device_code()	acquire_token_by_device_flow()
acquire_token_with_username_password()	acquire_token_by_username_password()
acquire_token_with_client_credentials() and acquire_token_with_client_certificate()	acquire_token_for_client()
N/A	acquire_token_on_behalf_of()
TokenCache()	SerializableTokenCache()
N/A	Cache with persistence, available from MSAL Extensions

Migrate existing refresh tokens for MSAL Python

The Microsoft Authentication Library (MSAL) abstracts the concept of refresh tokens. MSAL Python provides an in-memory token cache by default so that you don't need to store, lookup, or update refresh tokens. Users will also see fewer sign-in prompts because refresh tokens can usually be updated without user intervention. For more information about the token cache, see [Custom token cache serialization in MSAL for Python](#).

The following code will help you migrate your refresh tokens managed by another OAuth2 library (including but not limited to ADAL Python) to be managed by MSAL for Python. One reason for migrating those refresh tokens is to prevent existing users from needing to sign in again when you migrate your app to MSAL for Python.

The method for migrating a refresh token is to use MSAL for Python to acquire a new access token using the previous refresh token. When the new refresh token is returned, MSAL for Python will store it in the cache. Since MSAL Python 1.3.0, we provide an API inside MSAL for this purpose. Please refer to the following code snippet, quoted from [a completed sample of migrating refresh tokens with MSAL Python](#)

```

import msal
def get_preexisting_rt_and_their_scopes_from_elsewhere():
    # Maybe you have an ADAL-powered app like this
    #   https://github.com/AzureAD/azure-active-directory-library-for-
python/blob/1.2.3/sample/device_code_sample.py#L72
    # which uses a resource rather than a scope,
    # you need to convert your v1 resource into v2 scopes
    # See https://docs.microsoft.com/azure/active-directory/azuread-dev/azure-ad-endpoint-comparison#scopes-
not-resources
    # You may be able to append "./default" to your v1 resource to form a scope
    # See https://docs.microsoft.com/azure/active-directory/develop/v2-permissions-and-consent#the-default-
scope

    # Or maybe you have an app already talking to the Microsoft identity platform,
    # powered by some 3rd-party auth library, and persist its tokens somehow.

    # Either way, you need to extract RTs from there, and return them like this.
    return [
        ("old_rt_1", ["scope1", "scope2"]),
        ("old_rt_2", ["scope3", "scope4"]),
    ]


# We will migrate all the old RTs into a new app powered by MSAL
app = msal.PublicClientApplication(
    "client_id", authority="...",
    # token_cache=... # Default cache is in memory only.
    # You can learn how to use SerializableTokenCache from
    # https://msal-python.readthedocs.io/en/latest/#msal.SerializableTokenCache
)

# We choose a migration strategy of migrating all RTs in one loop
for old_rt, scopes in get_preexisting_rt_and_their_scopes_from_elsewhere():
    result = app.acquire_token_by_refresh_token(old_rt, scopes)
    if "error" in result:
        print("Discarding unsuccessful RT. Error: ", json.dumps(result, indent=2))

print("Migration completed")

```

Next steps

For more information, refer to [v1.0 and v2.0 comparison](#).

Active Directory Federation Services support in MSAL for Python

4/12/2022 • 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 your apps by using the Microsoft Authentication Library (MSAL) for Python. Using the MSAL for Python library, your app can authenticate users directly against AD FS. For more information about scenarios, see [AD FS Scenarios for Developers](#).

There are usually two ways of authenticating against AD FS:

- MSAL Python talks to Azure Active Directory, which itself is federated with other identity providers. The federation happens through AD FS. MSAL Python connects to Azure AD, which signs in users that are managed in Azure AD (managed users) or users managed by another identity provider such as AD FS (federated users). MSAL Python doesn't know that a user is federated. It simply talks to Azure AD. The [authority](#) you use in this case is the usual authority (authority host name + tenant, common, or organizations).
- MSAL Python talks directly to an AD FS authority. This is only supported by AD FS 2019 and later.

Connect to Active Directory federated with AD FS

Acquire a token interactively for a federated user

The following applies whether you connect directly to Active Directory Federation Services (AD FS) or through Active Directory.

When you call `acquire_token_by_authorization_code` or `acquire_token_by_device_flow`, the user experience is typically as follows:

1. The user enters their account ID.
2. Azure AD displays briefly the message "Taking you to your organization's page" and 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.

The supported AD FS versions in this federated scenario are:

- Active Directory Federation Services FS v2
- Active Directory Federation Services v3 (Windows Server 2012 R2)
- Active Directory Federation Services v4 (AD FS 2016)

Acquire a token via username and password

The following applies whether you connect directly to Active Directory Federation Services (AD FS) or through Active Directory.

When you acquire a token using `acquire_token_by_username_password`, MSAL Python gets the identity provider to contact based on the username. MSAL Python gets a [SAML 1.1 token](#) from the identity provider, which it then provides to Azure AD which returns the JSON Web Token (JWT).

Connecting directly to AD FS

When you connect directory to AD FS, the authority you'll want to use to build your application will be

something like `https://somesite.contoso.com/adfs/`

MSAL Python supports ADFS 2019.

It does not support a direct connection to ADFS 2016 or ADFS v2. To support scenarios requiring a direct connection to ADFS 2016, use the latest version of ADAL Python. Once you have upgraded your on-premises system to ADFS 2019, you can use MSAL Python.

Next steps

- For the federated case, see [Configure Azure Active Directory sign in behavior for an application by using a Home Realm Discovery policy](#)

Handle errors and exceptions in MSAL for Python

4/12/2022 • 2 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, 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 [Azure AD Authentication and authorization error codes](#).

During the sign-in experience, you may encounter errors about consents, Conditional Access (MFA, Device Management, Location-based restrictions), token issuance and redemption, and user properties.

The following section provides more details about error handling for your app.

Error handling in MSAL for Python

In MSAL for Python, most errors are conveyed as a return value from the API call. The error is represented as a dictionary containing the JSON response from the Microsoft identity platform.

- A successful response contains the `"access_token"` key. The format of the response is defined by the OAuth2 protocol. For more information, see [5.1 Successful Response](#)
- An error response contains `"error"` and usually `"error_description"`. The format of the response is defined by the OAuth2 protocol. For more information, see [5.2 Error Response](#)

When an error is returned, the `"error_description"` key contains a human-readable message; which in turn typically contains a Microsoft identity platform error code. For details about the various error codes, see [Authentication and authorization error codes](#).

In MSAL for Python, exceptions are rare because most errors are handled by returning an error value. The `ValueError` exception is only thrown when there is an issue with how you are attempting to use the library, such as when API parameter(s) are malformed.

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. This 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.

Retrying after errors and exceptions

You're expected to implement your own retry policies when calling MSAL. MSAL makes HTTP calls to the Azure AD service, and occasionally failures can occur. For example the network can go down or the server is overloaded.

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.

Next steps

Consider enabling [Logging in MSAL for Python](#) to help you diagnose and debug issues.

Logging in MSAL for Python

4/12/2022 • 2 minutes to read • [Edit Online](#)

The 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. Used 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.

The following sections provide more details about MSAL error logging for your application.

MSAL for Python logging

Logging in MSAL for Python leverages the [logging module in the Python standard library](#). You can configure MSAL logging as follows (and see it in action in the [username_password_sample](#)):

Enable debug logging for all modules

By default, the logging in any Python script is turned off. If you want to enable verbose logging for **all** Python modules in your script, use `logging.basicConfig` with a level of `logging.DEBUG`:

```
import logging

logging.basicConfig(level=logging.DEBUG)
```

This will print all log messages given to the logging module to the standard output.

Configure MSAL logging level

You can configure the logging level of the MSAL for Python log provider by using the `logging.getLogger()` method with the logger name `"msal"`:

```
import logging

logging.getLogger("msal").setLevel(logging.WARN)
```

Configure MSAL logging with Azure App Insights

Python logs are given to a log handler, which by default is the `StreamHandler`. To send MSAL logs to an Application Insights with an Instrumentation Key, use the `AzureLogHandler` provided by the `opencensus-ext-azure` library.

To install, `opencensus-ext-azure` add the `opencensus-ext-azure` package from PyPI to your dependencies or pip install:

```
pip install opencensus-ext-azure
```

Then change the default handler of the `"msal"` log provider to an instance of `AzureLogHandler` with an instrumentation key set in the `APP_INSIGHTS_KEY` environment variable:

```
import logging
import os

from opencensus.ext.azure.log_exporter import AzureLogHandler

APP_INSIGHTS_KEY = os.getenv('APP_INSIGHTS_KEY')

logging.getLogger("msal").addHandler(AzureLogHandler(connection_string='InstrumentationKey={0}'.format(APP_INSIGHTS_KEY)))
```

Personal and organizational data in Python

MSAL for Python does not log personal data or organizational data. There is no property to turn personal or organization data logging on or off.

You can use standard Python logging to log whatever you want, but you are responsible for safely handling sensitive data and following regulatory requirements.

For more information about logging in Python, please refer to Python's [Logging: how-to](#).

Next steps

For more code samples, refer to [Microsoft identity platform code samples](#).

Microsoft identity platform and OAuth 2.0 authorization code flow

4/12/2022 • 25 minutes to read • [Edit Online](#)

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 and Open ID Connect (OIDC), you can add sign in and API access to your mobile and desktop apps.

This article describes how to program directly against the protocol in your application using any language. When possible, we recommend you use the supported Microsoft Authentication Libraries (MSAL) to [acquire tokens and call secured web APIs](#). For more information, look at [sample apps that use MSAL](#).

The OAuth 2.0 authorization code flow is described in [section 4.1 of the OAuth 2.0 specification](#). With OIDC, this flow does authentication and authorization for most app types. These types include [single page apps](#), [web apps](#), and [natively installed apps](#). The flow enables apps to securely acquire an `access_token` that can be used to access resources secured by the Microsoft identity platform. Apps can refresh tokens to get other access tokens and ID tokens for the signed in user.

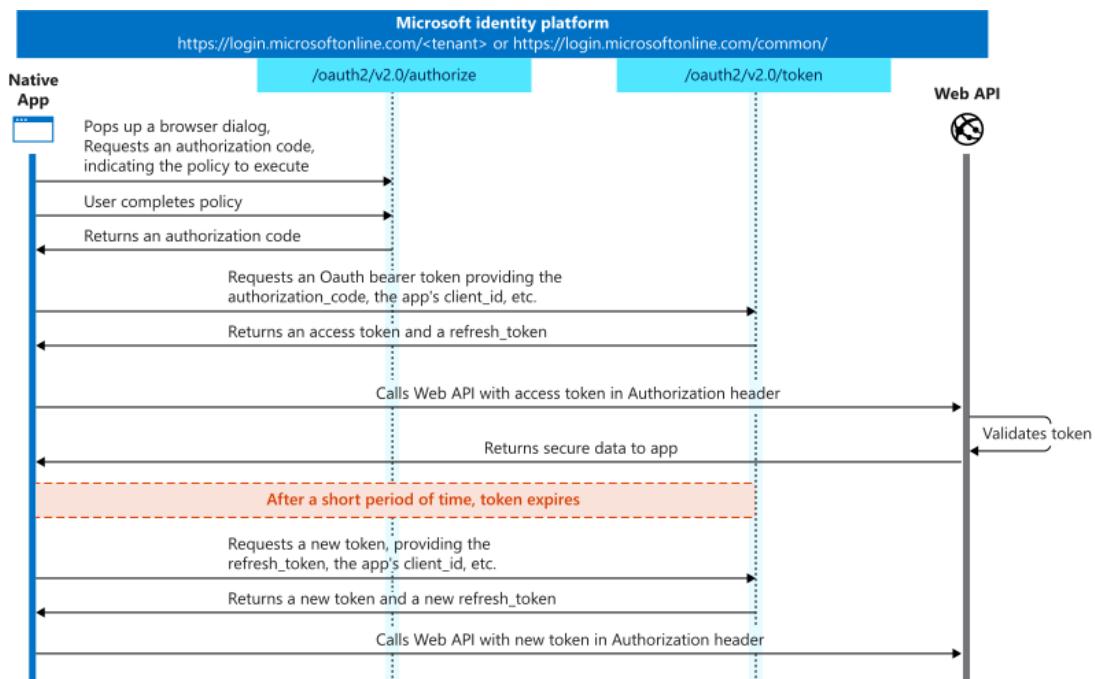
TIP

[▶ Run in Postman](#)

Try executing this request and more in Postman -- don't forget to replace tokens and IDs!

Protocol diagram

This diagram provides a high-level overview of the authentication flow for an application:



Redirect URI setup required for single-page apps

The authorization code flow for single page applications requires additional setup. Follow the instructions for [creating your single-page application](#) to correctly mark your redirect URI as enabled for Cross-Origin Resource Sharing (CORS). To update an existing redirect URI to enable CORS, open the manifest editor and set the `type` field for your redirect URI to `spa` in the `replyUrlsWithType` section. Or, you can select the redirect URI in [Authentication > Web](#) and select URIs to migrate to using the authorization code flow.

The `spa` redirect type is backwards compatible with the implicit flow. Apps currently using the implicit flow to

get tokens can move to the `spa` redirect URI type without issues and continue using the implicit flow.

If you attempt to use the authorization code flow without setting up CORS for your redirect URI, you will see this error in the console:

```
access to XMLHttpRequest at 'https://login.microsoftonline.com/common/v2.0/oauth2/token' from origin
'yourApp.com' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the
requested resource.
```

If so, visit your app registration and update the redirect URI for your app to use the `spa` type.

Applications can't use a `spa` redirect URI with non-SPA flows, for example, native applications or client credential flows. To ensure security and best practices, the Microsoft identity platform returns an error if you attempt to use a `spa` redirect URL without an `Origin` header. Similarly, the Microsoft identity platform also prevents the use of client credentials in all flows in the presence of an `Origin` header, to ensure that secrets aren't used from within the browser.

Request an authorization code

The authorization code flow begins with the client directing the user to the `/authorize` endpoint. In this request, the client requests the `openid`, `offline_access`, and `https://graph.microsoft.com/mail.read` permissions from the user.

Some permissions are admin-restricted, for example, writing data to an organization's directory by using `Directory.ReadWrite.All`. 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. To request access to admin-restricted scopes, you should request them directly from a Global Administrator. For more information, see [Admin-restricted permissions](#).

Unless specified otherwise, there are no default values for optional parameters. There is, however, default behavior for a request omitting optional parameters. The default behavior is to either sign in the sole current user, show the account picker if there are multiple users, or show the login page if there are no users signed in.

```
// 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%2Flocalhost%2Fmyapp%2F
&response_mode=query
&scope=https%3A%2F%2Fgraph.microsoft.com%2Fmail.read%20api%3A%2F%
&state=12345
&code_challenge=YTFjNjI1OWYzMzA3MTI4ZDY2Njg5M2RkNmVjNDE5YmEyZGRhOGYyM2IzNjdmZWfhMTQ1ODg3NDcxY2N1
&code_challenge_method=S256
```

TIP

Select the link below to execute this request! After signing in, your browser should be redirected to

<http://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. Valid values are <code>common</code> , <code>organizations</code> , <code>consumers</code> , and tenant identifiers. For guest scenarios where you sign a user from one tenant into another tenant, you <i>must</i> provide the tenant identifier to sign them into the resource tenant. For more information, see Endpoints .

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. Can also include <code>id_token</code> or <code>token</code> if using the hybrid flow .
<code>redirect_uri</code>	required	The <code>redirect_uri</code> 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 it must be URL-encoded. For native and mobile apps, use one of the recommended values: <code>https://login.microsoftonline.com/common/oauth2/native_app</code> for apps using embedded browsers or <code>http://localhost</code> for apps that use system browsers.
<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 parameter can cover multiple resources. This value allows your app to get consent for multiple web APIs you want to call.
<code>response_mode</code>	recommended	Specifies how the identity platform should return the requested token to your app. Supported values: <ul style="list-style-type: none"> - <code>query</code> : Default when requesting an access token. Provides the code as a query string parameter on your redirect URI. The <code>query</code> parameter is not supported when requesting an ID token by using the implicit flow. - <code>fragment</code> : Default when requesting an ID token by using the implicit flow. Also supported if requesting <i>only</i> a code. - <code>form_post</code> : Executes a POST containing the code to your redirect URI. Supported when requesting a code.
<code>state</code>	recommended	A value included in the request that is also 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. For instance, it could encode the page or view they were on.

PARAMETER	REQUIRED/OPTIONAL	DESCRIPTION
<code>prompt</code>	optional	<p>Indicates the type of user interaction that is required. Valid values are <code>login</code>, <code>none</code>, <code>consent</code>, and <code>select_account</code>.</p> <ul style="list-style-type: none"> - <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 isn't presented with any interactive prompt. If the request can't be completed silently by using single-sign on, the Microsoft identity platform returns an <code>interaction_required</code> error. - <code>prompt=consent</code> triggers the OAuth consent dialog after the user signs in, asking the user to grant permissions to the app. - <code>prompt=select_account</code> interrupts single sign-on providing account selection experience listing all the accounts either in session or any remembered account or an option to choose to use a different account altogether.
<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. Apps can use this parameter during reauthentication, after already extracting the <code>login_hint</code> optional claim from an earlier sign-in.
<code>domain_hint</code>	optional	If included, the app skips the email-based discovery process that user goes through on the sign-in page, leading to a slightly more streamlined user experience. For example, sending them to their federated identity provider. Apps can use this parameter during reauthentication, by extracting the <code>tid</code> from a previous sign-in.
<code>code_challenge</code>	recommended / required	<p>Used to secure authorization code grants by using Proof Key for Code Exchange (PKCE). Required if <code>code_challenge_method</code> is included. For more information, see the PKCE RFC. This parameter is now recommended for all application types, both public and confidential clients, and required by the Microsoft identity platform for single page apps using the authorization code flow.</p>

PARAMETER	REQUIRED/OPTIONAL	DESCRIPTION
<code>code_challenge_method</code>	recommended / required	<p>The method used to encode the <code>code_verifier</code> for the <code>code_challenge</code> parameter. This <i>SHOULD</i> be <code>s256</code>, but the spec allows the use of <code>plain</code> if the client can't support SHA256.</p> <p>If excluded, <code>code_challenge</code> is assumed to be plaintext if <code>code_challenge</code> is included. The Microsoft identity platform supports both <code>plain</code> and <code>s256</code>. For more information, see the PKCE RFC. This parameter is required for single page apps using the authorization code flow.</p>

At this point, the user is asked to enter their credentials and complete the authentication. The Microsoft identity platform also ensures 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, it asks the user to consent to the required permissions. For more information, see [Permissions and consent in the Microsoft identity platform](#).

Once the user authenticates and grants consent, the Microsoft identity platform returns a response to your app at the indicated `redirect_uri`, using the method specified in the `response_mode` parameter.

Successful response

This example shows a successful response using `response_mode=query`:

```
GET http://localhost?
code=AwABAAAAvPM1KaPlrEqdFSBzjqfTGBCmLdgfSTLEMPGYuNHSUYBrq...
&state=12345
```

PARAMETER	DESCRIPTION
<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. Authorization codes are short lived. Typically, they 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 state values in the request and response are identical.

You can also receive an ID token if you request one and have the implicit grant enabled in your application registration. This behavior is sometimes referred to as the [hybrid flow](#). It's used by frameworks like ASP.NET.

Error response

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

```
GET http://localhost?
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, and to react to errors. This part of the error is provided so that the app can react appropriately to the error, but does not explain in depth why an error occurred.

PARAMETER	DESCRIPTION
<code>error_description</code>	A specific error message that can help a developer identify the cause of an authentication error. This part of the error contains most of the useful information about <i>why</i> the error occurred.

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 error 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 continue unless the user consents.
<code>unsupported_response_type</code>	The authorization server doesn't support the response type in the request.	Fix and resubmit the request. This error is a development error typically caught during initial testing. In the hybrid flow , this error signals that you must enable the ID token implicit grant setting on the client app registration.
<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, hasn't been configured in the tenant. The application can prompt the user with instruction for installing the application and adding it to Azure AD.
<code>login_required</code>	Too many or no users found.	The client requested silent authentication (<code>prompt=none</code>), but a single user couldn't be found. This error may mean there are multiple users active in the session, or no users. This error 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 works.

ERROR CODE	DESCRIPTION	CLIENT ACTION
<code>interaction_required</code>	The request requires user interaction.	Another authentication step or consent is required. Retry the request without <code>prompt=none</code> .

Request an ID token as well or hybrid flow

To learn who the user is before redeeming an authorization code, it's common for applications to also request an ID token when they request the authorization code. This approach is called the *hybrid flow* because it mixes the implicit grant with the authorization code flow.

The hybrid flow is commonly used in web apps to render a page for a user without blocking on code redemption, notably in [ASP.NET](#). Both single-page apps and traditional web apps benefit from reduced latency in this model.

The hybrid flow is the same as the authorization code flow described earlier but with three additions. All of these additions are required to request an ID token: new scopes, a new `response_type`, and a new `nonce` query parameter.

```
// Line breaks for legibility only

https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&response_type=code%20id_token
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
&response_mode=fragment
&scope=openid%20offline_access%20https%3A%2F%2Fgraph.microsoft.com%2Fuser.read
&state=12345
&nonce=abcde
&code_challenge=YTFjNjI10WYzMzA3MTI4ZDY2Njg5M2RkNmVjNDE5YmEyZGRhOGYyM2IzNjdmZWfhMTQ1ODg3NDcxY2Nl
&code_challenge_method=S256
```

UPDATED PARAMETER	REQUIRED/OPTIONAL	DESCRIPTION
<code>response_type</code>	required	The addition of <code>id_token</code> indicates to the server that the application would like an ID token in the response from the <code>/authorize</code> endpoint.
<code>scope</code>	required	For ID tokens, this parameter must be updated to include the ID token scopes: <code>openid</code> and optionally <code>profile</code> and <code>email</code> .
<code>nonce</code>	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 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 token back to your app. Default value is <code>query</code> for just an authorization code, but <code>fragment</code> if the request includes an <code>id_token</code> <code>response_type</code> as specified in the OpenID spec . We recommend apps use <code>form_post</code> , especially when using <code>http://localhost</code> as a redirect URI.

The use of `fragment` as a response mode causes issues for web apps that read the code from the redirect.

Browsers don't pass the fragment to the web server. In these situations, apps should use the `form_post`

response mode to ensure that all data is sent to the server.

Successful response

This example shows a successful response using `response_mode=fragment`:

```
GET https://login.microsoftonline.com/common/oauth2/nativeclient#
code=AwABAAAAvPM1KaPlrEqdFSBzjqfTGBCmLdgfSTLEMPGYuNHSUYBrq...
&id_token=eYj...
&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 expiring after about 10 minutes.
<code>id_token</code>	An ID token for the user, issued by using the <i>implicit grant</i> . Contains a special <code>c_hash</code> claim that is the hash of the <code>code</code> in the same request.
<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.

Redeem a code for an access token

All confidential clients have a choice of using client secrets or certificate credentials. Symmetric shared secrets are generated by the Microsoft identity platform. Certificate credentials are asymmetric keys uploaded by the developer. For more information, see [Microsoft identity platform application authentication certificate credentials](#).

For best security, we recommend using certificate credentials. Public clients, which include native applications and single page apps, must not use secrets or certificates when redeeming an authorization code. Always ensure that your redirect URLs include the type of application and [are unique](#).

Request an access token with a client_secret

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 resource. Redeem the `code` 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-6eba6914391
&scope=https%3A%2F%2Fgraph.microsoft.com%2Fmail.read
&code=0AAABAAAiL9Kn2Z7UubvWPbm0gLWQJVzCTE9UKP3pSx1aXxUjq3n8b2JRLk40xVXr...
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
&grant_type=authorization_code
&code_verifier=ThisIsntRandomButItNeedsToBe43CharactersLong
&client_secret=JqQX2PNo9bpM0uEihUPzyrh    // NOTE: Only required for web apps. This secret needs to be URL-Encoded.
```

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. Valid values are <code>common</code> , <code>organizations</code> , <code>consumers</code> , and tenant identifiers. For more information, see Endpoints .

PARAMETER	REQUIRED/OPTIONAL	DESCRIPTION
<code>client_id</code>	required	The Application (client) ID that the Azure portal – App registrations page assigned to your app.
<code>scope</code>	optional	A space-separated list of scopes. The scopes must all be from a single resource, along with OIDC scopes (<code>profile</code> , <code>openid</code> , <code>email</code>). For more information, see Permissions and consent in the Microsoft identity platform . This parameter is a Microsoft extension to the authorization code flow, intended to allow apps to declare the resource they want the token for during token redemption.
<code>code</code>	required	The <code>authorization_code</code> that you acquired in the first leg of the flow.
<code>redirect_uri</code>	required	The same <code>redirect_uri</code> value that was used to acquire the <code>authorization_code</code> .
<code>grant_type</code>	required	Must be <code>authorization_code</code> for the authorization code flow.
<code>code_verifier</code>	recommended	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 .
<code>client_secret</code>	required for confidential web apps	The application secret that you created in the app registration portal for your app. Don't use the application secret in a native app or single page app because a <code>client_secret</code> can't be reliably stored on devices or web pages. It's required for web apps and web APIs, which can store the <code>client_secret</code> securely on the server side. Like all parameters here, the client secret must be URL-encoded before being sent. This step is usually done by the SDK. For more information on URI encoding, see the URI Generic Syntax specification . The Basic auth pattern of instead providing credentials in the Authorization header, per RFC 6749 is also supported.

Request an access token with a certificate credential

```
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=6731de76-14a6-49ae-97bc-6eba6914391e
&scope=https%3A%2F%2Fgraph.microsoft.com%2Fmail.read
&code=0AAABAAAAiL9Kn2Z27UubvWFpbm0gLWQJVzCTE9UkP3pSx1aXxUjq3n8b2JRLk40xVXr...
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
&grant_type=authorization_code
&code_verifier=ThisIsntRandomButItNeedsToBe43CharactersLong
&client_assertion_type=urn%3AiETF%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJSUzI1NiIisInglcI6Imd4OHRHeXN5amNScUtqRlBuZDdSRnd2d1pJMCJ9.eyJ{a lot of
characters here}M8U3bSUKKJDEg
```

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. Valid values are <code>common</code> , <code>organizations</code> , <code>consumers</code> , and tenant identifiers. For more detail, see Endpoints .
<code>client_id</code>	required	The Application (client) ID that the Azure portal – App registrations page assigned to your app.
<code>scope</code>	optional	A space-separated list of scopes. The scopes must all be from a single resource, along with OIDC scopes (<code>profile</code> , <code>openid</code> , <code>email</code>). For more information, see permissions, consent, and scopes . This parameter is a Microsoft extension to the authorization code flow. This extension allows apps to declare the resource they want the token for during token redemption.
<code>code</code>	required	The <code>authorization_code</code> that you acquired in the first leg of the flow.
<code>redirect_uri</code>	required	The same <code>redirect_uri</code> value that was used to acquire the <code>authorization_code</code> .
<code>grant_type</code>	required	Must be <code>authorization_code</code> for the authorization code flow.
<code>code_verifier</code>	recommended	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 .
<code>client_assertion_type</code>	required for confidential web apps	The value must be set to <code>urn:ietf:params:oauth:client-assertion-type:jwt-bearer</code> to use a certificate credential.
<code>client_assertion</code>	required for confidential web apps	An assertion, which is a JSON web token (JWT), 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.

The parameters are same as 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

This example shows a successful token response:

```
{
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZn10aEV1Q...",
    "token_type": "Bearer",
    "expires_in": 3599,
    "scope": "https%3A%2F%2Fgraph.microsoft.com%2Fmail.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 <code>Bearer</code> .
<code>expires_in</code>	How long the access token is valid, in seconds.
<code>scope</code>	The scopes that the <code>access_token</code> is valid for. Optional. This parameter is non-standard and, if omitted, the token is for the scopes requested on the initial leg of the flow.
<code>refresh_token</code>	An OAuth 2.0 refresh token. The app can use this token to acquire other access tokens after the current access token expires. Refresh tokens are long-lived. They can maintain access to resources for extended periods. For more detail on refreshing an access token, refer to Refresh the access token later in this article. Note: Only provided if <code>offline_access</code> scope was requested.
<code>id_token</code>	A JSON Web Token. 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, and confidential clients can use this token for authorization. For more information about id_tokens, see the id_token reference . Note: Only provided if <code>openid</code> scope was requested.

Error response

This example is an 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, and to react to errors.
<code>error_description</code>	A specific error message that can help a developer identify the cause of an authentication error.
<code>error_codes</code>	A list of STS-specific error codes that can help in diagnostics.

PARAMETER	DESCRIPTION
<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 the request or app registration 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 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>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 doesn't 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 doesn't exist, Azure AD can't find it, or it's not correctly configured.	This code indicates the resource, if it exists, hasn't 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>	Non-standard, as the OIDC specification calls for this code only on the <code>/authorize</code> endpoint. The request requires user interaction. For example, another authentication step is required.	Retry the <code>/authorize</code> request with the same scopes.
<code>temporarily_unavailable</code>	The server is temporarily too busy to handle the request.	Retry the request after a small delay. The client application might explain to the user that its response is delayed because of a temporary condition.
<code>consent_required</code>	The request requires user consent. This error is non-standard. It's usually only returned on the <code>/authorize</code> endpoint per OIDC specifications. Returned when a <code>scope</code> parameter was used on the code redemption flow that the client app doesn't have permission to request.	The client should send the user back to the <code>/authorize</code> endpoint with the correct scope to trigger consent.

ERROR CODE	DESCRIPTION	CLIENT ACTION
<code>invalid_scope</code>	The scope requested by the app is invalid.	Update the value of the <code>scope</code> parameter in the authentication request to a valid value.

NOTE

Single page apps may receive an `invalid_request` error indicating that cross-origin token redemption is permitted only for the 'Single-Page Application' client-type. This indicates that the redirect URI used to request the token has not been marked as a `spa` redirect URI. Review the [application registration steps](#) on how to enable this flow.

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:

```
GET /v1.0/me/messages
Host: https://graph.microsoft.com
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZn10aEV1Q...
```

Refresh the access token

Access tokens are short lived. Refresh them after they expire to continue accessing resources. You can do so by submitting another `POST` request to the `/token` endpoint. Provide the `refresh_token` instead of the `code`. Refresh tokens are valid for all permissions that your client has already received consent for. For example, 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 for web apps and native apps don't 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 action. Your application needs to expect and handle [errors returned by the token issuance endpoint](#). Single page apps get a token with a 24-hour lifetime, requiring a new authentication every day. This action can be done silently in an iframe when third-party cookies are enabled. It must be done in a top-level frame, either full page navigation or a pop-up window, in browsers without third-party cookies, such as Safari.

Refresh tokens aren't revoked when used to acquire new access tokens. You're 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."

IMPORTANT

For refresh tokens sent to a redirect URI registered as `spa`, the refresh token expires after 24 hours. Additional refresh tokens acquired using the initial refresh token carries over that expiration time, so apps must be prepared to re-run the authorization code flow using an interactive authentication to get a new refresh token every 24 hours. Users do not have to enter their credentials, and usually don't even see any user experience, just a reload of your application. The browser must visit the login page in a top level frame in order to see the login session. This is due to [privacy features in browsers that block third party cookies](#).

```
// 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=535fb089-9ff3-47b6-9bfb-4f1264799865
&scope=https%3A%2F%2Fgraph.microsoft.com%2Fmail.read
&refresh_token=AAAAAAAI9Kn2Z7UubvWFPbm0gLWQJVzCTE9UkP3pSx1aXxUjq...
&grant_type=refresh_token
&client_secret=sampleCredential1s    // NOTE: Only required for web apps. This secret needs to be URL-Encoded
```

PARAMETER	TYPE	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. Valid values are <code>common</code> , <code>organizations</code> , <code>consumers</code> , and tenant identifiers. For more information, see Endpoints .
<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>	optional	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 <code>authorization_code</code> request leg. If the scopes specified in this request span multiple resource server, then the Microsoft identity platform returns a token for the resource specified in the first scope. For more information, see Permissions and consent in the Microsoft identity platform .
<code>refresh_token</code>	required	The <code>refresh_token</code> 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 shouldn't be used in a native app, because a <code>client_secret</code> can't be reliably stored on devices. It's required for web apps and web APIs, which can store the <code>client_secret</code> securely on the server side. This secret needs to be URL-Encoded. For more information, see the URI Generic Syntax specification .

Successful response

This example shows a successful token response:

```
{
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZnl0aEV1Q...",
  "token_type": "Bearer",
  "expires_in": 3599,
  "scope": "https%3A%2F%2Fgraph.microsoft.com%2Fmail.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.

PARAMETER	DESCRIPTION
<code>scope</code>	The scopes that the <code>access_token</code> is valid for.
<code>refresh_token</code>	A new OAuth 2.0 refresh token. 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. 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 <code>id_token</code> , see the Microsoft identity platform ID tokens . Note: Only provided if <code>openid</code> scope was requested.

WARNING

Don't attempt to validate or read tokens for any API you don't own, including the tokens in this example, in your code. Tokens for Microsoft services can use a special format that will not validate as a JWT, and may also be encrypted for consumer (Microsoft account) users. While reading tokens is a useful debugging and learning tool, do not take dependencies on this in your code or assume specifics about tokens that aren't for an API you control.

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, and 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](#).

Next steps

- Go over the [MSAL JS samples](#) to get started coding.
- Learn about [token exchange scenarios](#).

Microsoft identity platform and the OAuth 2.0 client credentials flow

4/12/2022 • 13 minutes to read • [Edit Online](#)

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*.

This article describes how to program directly against the protocol in your application. When possible, we recommend you use the supported Microsoft Authentication Libraries (MSAL) instead to [acquire tokens and call secured web APIs](#). Also take a look at the [sample apps that use MSAL](#).

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. For a higher level of assurance, the Microsoft identity platform also allows the calling service to authenticate using a [certificate](#) or federated credential instead of a shared secret. Because the application's own credentials are being used, these credentials must be kept safe - *never* publish that credential in your source code, embed it in web pages, or use it in a widely distributed native application.

In the client credentials flow, permissions are granted directly to the application itself by an administrator. When the app presents a token to a resource, the resource enforces that the app itself has authorization to perform an action since there is no user involved in the authentication. This article covers both the steps needed to [authorize an application to call an API](#), as well as [how to get the tokens needed to call that API](#).

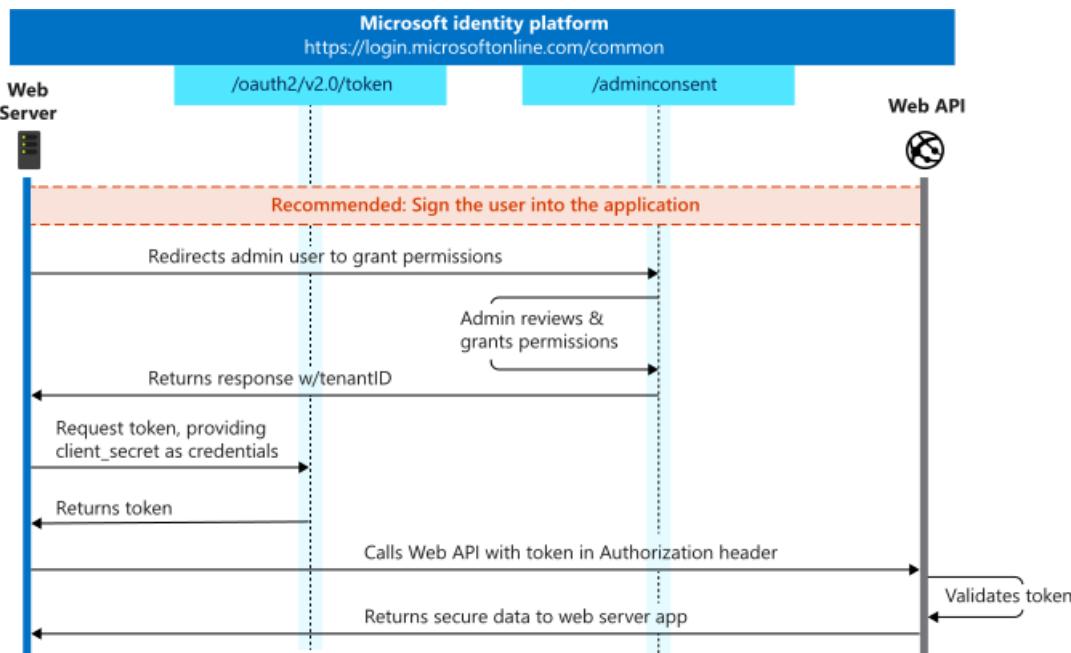
TIP

[▶ Run in Postman](#)

Try executing this request and more in Postman -- don't forget to replace tokens and IDs!

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, 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 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.

Controlling tokens without the `roles` claim

In order to enable this ACL-based authorization pattern, Azure AD doesn't require that applications be authorized to get tokens for another application. Thus, app-only tokens can be issued without a `roles` claim. Applications that expose APIs must implement permission checks in order to accept tokens.

If you'd like to prevent applications from getting role-less app-only access tokens for your application, [ensure that assignment requirements are enabled for your app](#). This will block users and applications without assigned roles from being able to get a token for this application.

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

To use app roles (application permissions) with your own API (as opposed to Microsoft Graph), you must first [expose the app roles](#) in the API's app registration in the Azure portal. Then, [configure the required app roles](#) by selecting those permissions in your client application's app registration. If you haven't exposed any app roles in your API's app registration, you won't be able to specify application permissions to that API in your client application's app registration in the Azure portal.

When authenticating as an application (as opposed to with a user), you can't use *delegated permissions* because there is no user for your app to act on behalf of. You must use application permissions, also known as app roles, that are granted by an admin or by the API's owner.

For more information about application permissions, see [Permissions and consent](#).

Recommended: Sign the admin into your app to have app roles assigned

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 the [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*.

```
// 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
-----------	-----------	-------------

PARAMETER	CONDITION	DESCRIPTION
<code>tenant</code>	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 <code>common</code> .
<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 portal, except that it must be URL-encoded, and it can have additional path segments.
<code>state</code>	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
<code>tenant</code>	The directory tenant that granted your application the permissions that it requested, in GUID format.
<code>state</code>	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.
<code>admin_consent</code>	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
<code>error</code>	An error code string that you can use to classify types of errors, and which you can use to react to errors.
<code>error_description</code>	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:

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=sampleCredentials
&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.

PARAMETER	CONDITION	DESCRIPTION
<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 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	<p>The client secret that you generated for your app in the app registration portal. The client secret must be URL-encoded before being sent. The Basic auth pattern of instead providing credentials in the Authorization header, per RFC 6749 is also supported.</p>
<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          // 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=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.

PARAMETER	CONDITION	DESCRIPTION
<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 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	<p>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.</p>
<code>grant_type</code>	Required	Must be set to <code>client_credentials</code> .

The parameters for the certificate-based request differ in only one way from the shared secret-based request: the `client_secret` parameter is replaced by the `client_assertion_type` and `client_assertion` parameters.

Third case: Access token request with a federated credential

```

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

```

PARAMETER	CONDITION	DESCRIPTION
-----------	-----------	-------------

PARAMETER	CONDITION	DESCRIPTION
<code>client_assertion</code>	Required	An assertion (a JWT, or JSON web token) that your application gets from another identity provider outside of Microsoft identity platform, like Kubernetes. The specifics of this JWT must be registered on your application as a federated identity credential . Read about workload identity federation to learn how to setup and use assertions generated from other identity providers.

Everything in the request is the same as the certificate-based flow above, with one crucial exception - the source of the `client_assertion`. In this flow, your application does not create the JWT assertion itself. Instead, your app uses a JWT created by another identity provider. This is called "[workload identity federation](#)", where your app's identity in another identity platform is used to acquire tokens inside the Microsoft identity platform. This is best suited for cross-cloud scenarios, such as hosting your compute outside Azure but accessing APIs protected by Microsoft identity platform. For information about the required format of JWTs created by other identity providers, read about the [assertion format](#).

Successful response

A successful response from any method 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 the Microsoft identity platform supports is <code>bearer</code> .
<code>expires_in</code>	The amount of time that an access token is valid (in seconds).

WARNING

Don't attempt to validate or read tokens for any API you don't own, including the tokens in this example, in your code. Tokens for Microsoft services can use a special format that will not validate as a JWT, and may also be encrypted for consumer (Microsoft account) users. While reading tokens is a useful debugging and learning tool, do not take dependencies on this in your code or assume specifics about tokens that aren't for an API you control.

Error response

An error response (400 Bad Request) 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 eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZnl0aEV1Q...
```

```
# 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
--------	----------	-------------

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

4/12/2022 • 4 minutes to read • [Edit Online](#)

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.

This article describes how to program directly against the protocol in your application. When possible, we recommend you use the supported Microsoft Authentication Libraries (MSAL) instead to [acquire tokens and call secured web APIs](#). Also take a look at the [sample apps that use MSAL](#).

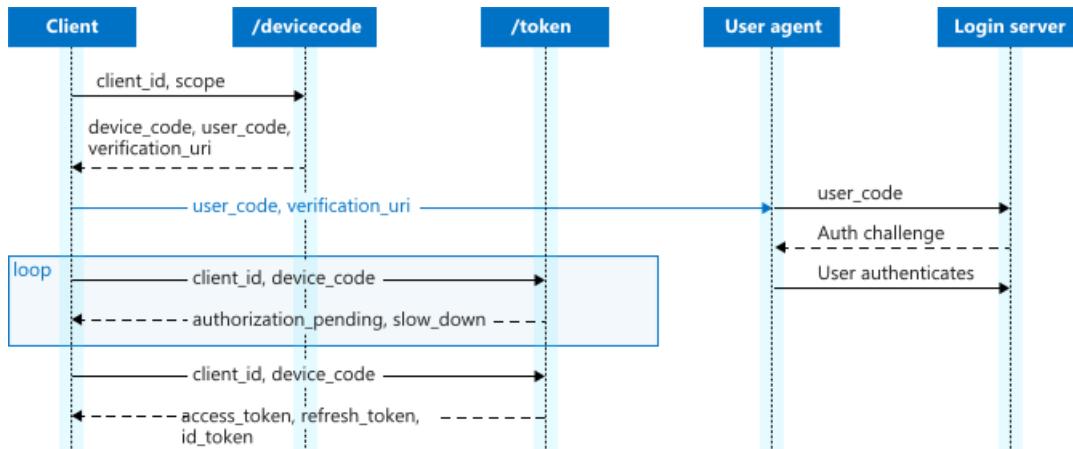
TIP

▶ [Run in Postman](#)

Try executing this request and more in Postman -- don't forget to replace tokens and IDs!

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.

```
// 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>	Required	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.

ERROR	DESCRIPTION	CLIENT ACTION
<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 <code>Bearer</code> .
<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 <code>scopes</code> 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](#).

WARNING

Don't attempt to validate or read tokens for any API you don't own, including the tokens in this example, in your code. Tokens for Microsoft services can use a special format that will not validate as a JWT, and may also be encrypted for consumer (Microsoft account) users. While reading tokens is a useful debugging and learning tool, do not take dependencies on this in your code or assume specifics about tokens that aren't for an API you control.

Microsoft identity platform and OAuth 2.0 On-Behalf-Of flow

4/12/2022 • 15 minutes to read • [Edit Online](#)

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.

The OBO flow only works for user principals at this time. A service principal cannot request an app-only token, send it to an API, and have that API exchange that for another token that represents that original service principal. Additionally, the OBO flow is focused on acting on another party's behalf, known as a delegated scenario - this means that it uses only delegated *scopes*, and not application *roles*, for reasoning about permissions. *Roles* remain attached to the principal (the user) in the flow, never the application operating on the user's behalf.

This article describes how to program directly against the protocol in your application. When possible, we recommend you use the supported Microsoft Authentication Libraries (MSAL) instead to [acquire tokens and call secured web APIs](#). Also take a look at the [sample apps that use MSAL](#).

TIP

[▶ Run in Postman](#)

Try executing this request and more in Postman -- don't forget to replace tokens and IDs!

Client limitations

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.

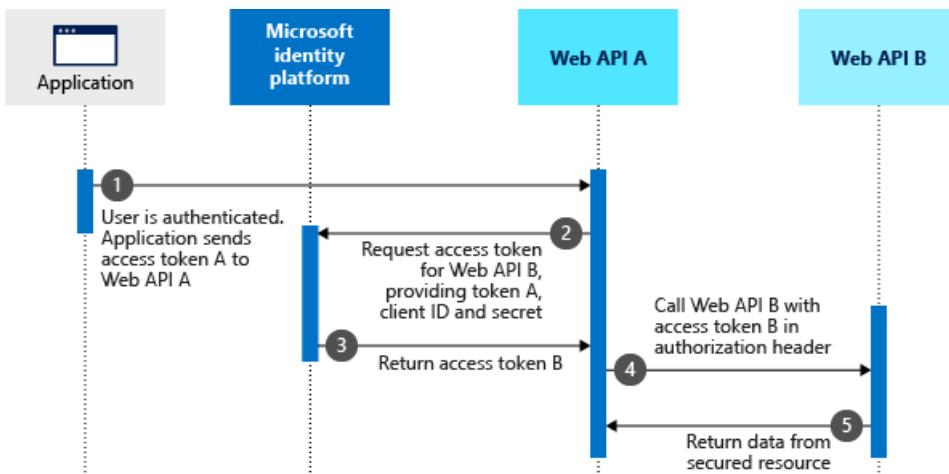
If a client uses the implicit flow to get an `id_token`, and that client also has wildcards in a reply URL, the `id_token` can't be used for an OBO flow. However, access tokens acquired through the implicit grant flow can still be redeemed by a confidential client even if the initiating client has a wildcard reply URL registered.

Additionally, applications with custom signing keys cannot be used as middle-tier API's in the OBO flow (this includes enterprise applications configured for single sign-on). This will result in an error because tokens signed with a key controlled by the client cannot be safely accepted.

Protocol diagram

Assume that the user has been authenticated on an application using the [OAuth 2.0 authorization code grant flow](#) or another login 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 along with token A and issues the access token for API B (token B) to API A.
4. Token B is set by API A in the authorization header of the request to API B.
5. Data from the secured resource is returned by API B to API A, then to the client.

In this scenario, the middle-tier service has no user interaction to get 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](#).

Middle-tier 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
```

WARNING

DO NOT send access tokens that were issued to the middle tier to any other party. Access tokens issued to the middle tier are intended for use *only* by that middle tier.

Security risks of relaying access tokens from a middle-tier resource to a client (instead of the client getting the access tokens themselves) include:

- Increased risk of token interception over compromised SSL/TLS channels.
- Inability to satisfy token binding and Conditional Access scenarios requiring claim step-up (for example, MFA, Sign-in Frequency).
- Incompatibility with admin-configured device-based policies (for example, MDM, location-based policies).

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	TYPE	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. The Basic auth pattern of instead providing credentials in the Authorization header, per RFC 6749 is also supported.
<code>assertion</code>	Required	The access token that was sent to the middle-tier API. This token must have an audience (<code>aud</code>) claim of the app making this OBO request (the app denoted by the <code>client-id</code> field). Applications cannot redeem a token for a different app (so e.g. if a client sends an API a token meant for MS Graph, the API cannot redeem it using OBO. It should instead reject the token).
<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/<tenant>
Content-Type: application/x-www-form-urlencoded

grant_type=urn:ietf:params:oauth:grant-type:jwt-bearer
client_id=535fb089-9ff3-47b6-9bfb-4f1264799865
&client_secret=sampleCredentials
&assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6InowMzI6ZHNGdWl6cEJmQ1ZLMVRuMjVRSF1PMCJ9.eyJhdWQiOiIy
O{a lot of characters here}
&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	TYPE	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 access token that was sent to the middle-tier API. This token must have an audience (<code>aud</code>) claim of the app making this OBO request (the app denoted by the <code>client-id</code> field). Applications cannot redeem a token for a different app (so e.g. if a client sends an API a token meant for MS Graph, the API cannot redeem it using OBO. It should instead reject the token).
<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/<tenant>
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=eyJhbGciOiJSUzI1NiIsIng1dCI6Imd40HRHeXN5amNScUtqR1BuZDdSRnd2d1pJMCJ9.eyJ{a lot of
characters here}
&assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6InowMz16ZHNGdWl6cEJmQ1ZLMVRuMjVRSF1PMCIisImtpZCI6InowM
z16ZHNGdWl6cEJmQ1ZLMVRuMjVRSF1PMCI9.eyJhdWQiO{a lot of characters here}
&requested_token_use=on_behalf_of
&scope=https://graph.microsoft.com/user.read+offline_access
```

Middle-tier 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 the 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.
<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": "eyJ0eXAiOiJKV1QiLCJub25jZSI6IkFRQUJBQUFBQUFCbmZpRy1tQTZOVGFlN0NkV1c3UWZkQ0NDYy0tY0hGa18wZE50MVEtc2loVzRMd2RwQVZISGpnTVdQZ0tQeVJ1aG1DbUN2NkdyMEpmYmRfY1RmMUFXu21TcFJkVXVydVJqX3Nqd0J0n211eH1BQSIsImFsZyI6I1JTmjU2IiwieDV0IjoiejAzOXpkc0Z1aXpwQmZCVksxVG4yNVFIWU8wIiwiia2lkIjoiejAzOXpkc0Z1aXpwQmZCVksxVG4yNVFIWU8wIn0.eyJhdWQiOjodHRwczovL2dyYXB0Lm1pY3Jvc29mdC5jb20iLCJpc3MiOjodHRwczovL3N0cy53aw5kb3dzLm51dc83MmY50DhiZi04NmYxLTQxYWYtOTFhYi0yZDdjZDAxMWRINDcvIiwiawF0IjoxNDkzOTMwMzA1LCJuYmYi0jE0OTM5MzAzMDUsImV4cCI6MTQ5MzkzMzg3NSwiYWNyIjoiMCIsImFpbbyI6IkFTUUEyLzhEQUFBU9KYNFFW1RNTnEyZFcXYPkN1RZMD1YeDd0t29EMkJEU1RWMXJ3b2Zrc1k9IiwiYwIjpbInB3ZCJdLCJhcHBfZG1zcGxheW5hbWUiOjUb2RvRG90bmV0T2JvIiwiYXBwaWQioiIyODQ2ZjcxYi1hN2E0LTQ50DctYmFiMy03NjAwMzViMmYzODkiLCJhcHBpZGFjci6IjEiLCJmYW1pbH1fbmFtZSI6IkNhbnVtYWxsYSIsImdpdmVuX25hbWUiOjJOYXZ5YSIsImlwYWRkciI6IjE2Ny4yMjAuMC4xOTkicJuyW1lIjoiTmF2eWEgQ2FudW1hbGxhIiwiib2lkIjoiZDV10Tc5YzctM2QyZC00MmFmLThmMzAtNzI3ZGQ0YzJkMzgziIiwiib25wcmVtX3NpZCI6IlMtMS01LTIxLTIxMjc1MjExODQtMTYwNDAxMjkyMC0x0Dg30TI3NTI3LTi2MTE4NDg0IiwiGxhdGYi0iIxNCIsInB1aWQioiIxMDAzM0ZGRkEwNkQxN0M5Iiwc2NwIjoiVxNlcis5ZWfkIiwiic3ViIjoiwTMMHBIxlpMXQ1ckRGd2JZ1JvTwrxZE52b3UzSjNWNm84UFE3alVCRSI sInRpZCI6IjcyZjk40GJmLtg2ZjEtNDfhZi05MWFiltJkN2NkMDExZGI0NyIsInVuaXF1ZV9uYW1lIjoiwmfjYW51bWFAbw1jcm9zb2Z0LmNvbSIsInVwb1I6Im5hY2FudW1hQG1pY3Jvc29mdC5jb20iLCJ1dGkiOjJWR1ItdmteZ1BFQ2M1dWFdaENRSkFBIiwidmVyiJoiMS4wIn0.cubh1L2VtruiiwF8ut1m9uNBmnUJeYx4x0G30F7CqSpzHj1Sv5DCgNZXyUz3pEiz77G8If0F0_U5A_02k-xzwdYvtJUYGH3bfISzdqymiEGmdfcIRK19KMeeo211Gv0ScCniIhr2U1yxTIkIpp092xcdadT-2_2q_q1Ha_HtjvTV1f9XR3t7_Id9bR5BqwVx5zP07JMYDVhUZRx08eqZcC-F3wi0xd_5ND_mavMuxe2wrpF-EZvi03yg0QVRr59tE3AoW181SGpVc97vvRCnp4WVRk26jJhYXFPsdk4yWqOKZqr3IFGyD08WizD_vPSrXcCPbZP3XWaoTUKZSNJg", "refresh_token": "OAQABAAAAAAABnfig-mA6NTae7CdW7QfdAALzDWjw6qSn4GUDfxWzJDZ6lk9qRw4An{a lot of characters here}" }
}
```

The above access token is a v1.0-formatted token for Microsoft Graph. This is because the token format is based on the **resource** being accessed and unrelated to the endpoints used to request it. The Microsoft Graph is setup to accept v1.0 tokens, so the Microsoft identity platform produces v1.0 access tokens when a client requests tokens for Microsoft Graph. Other apps may indicate that they want v2.0-format tokens, v1.0-format tokens, or even proprietary or encrypted token formats. Both the v1.0 and v2.0 endpoints can emit either format of token - this way the resource can always get the right format of token regardless of how or where the token was requested by the client.

WARNING

Don't attempt to validate or read tokens for any API you don't own, including the tokens in this example, in your code. Tokens for Microsoft services can use a special format that will not validate as a JWT, and may also be encrypted for consumer (Microsoft account) users. While reading tokens is a useful debugging and learning tool, do not take dependencies on this in your code or assume specifics about tokens that aren't for an API you control.

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 [multifactor 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 multifactor 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 eyJ0eXAiO ... 0X2tnSQLLEANnSPHY0gKcgw
```

SAML assertions obtained with an OAuth2.0 OBO flow

Some OAuth-based web services need to access other web service APIs that accept SAML assertions in non-interactive flows. Azure Active Directory can provide a SAML assertion in response to an On-Behalf-Of flow that uses a SAML-based web service as a target resource.

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	TYPE	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	TYPE	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. The Basic auth pattern of instead providing credentials in the Authorization header, per RFC 6749 is also supported.
scope	required	A space-separated list of scopes for the token request. For more information, see scopes . SAML itself doesn't have a concept of scopes, but here it is used to identify the target SAML application for which you want to receive a token. For this OBO flow, the scope value must always be the SAML Entity ID with <code>/default</code> appended. For example, in case the SAML application's Entity ID is <code>https://testapp.contoso.com</code> , then the requested scope should be <code>https://testapp.contoso.com/.default</code> . In case the Entity ID doesn't start with a URI scheme such as <code>https:</code> , Azure AD prefixes the Entity ID with <code>spn:</code> . In that case you must request the scope <code>spn:<EntityID>/default</code> , for example <code>spn:testapp/.default</code> in case the Entity ID is <code>testapp</code> . Note that the scope value you request here determines the resulting <code>Audience</code> element in the SAML token, which may be important to the SAML application receiving the token.
requested_token_use	required	Specifies how the request should be processed. In the On-Behalf-Of flow, the value must be <code>on_behalf_of</code> .
requested_token_type	required	Specifies the type of token requested. The value can be <code>urn:ietf:params:oauth:token-type:saml2</code> or <code>urn:ietf:params:oauth:token-type:saml1</code> 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.
- **API permissions:** You have to [add the necessary API permissions](#) on the middle-tier application to allow access to the SAML application, so that it can request a token for the `/.default` scope of the SAML application.
- **Consent:** Consent must have been granted to receive a SAML token containing user data on an OAuth flow. For information, see [Gaining consent for the middle-tier application](#) below.

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.
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>

Gaining consent for the middle-tier application

Depending on the architecture or usage of 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 so that the client app can call the middle-tier app, and the middle tier app has permission to call the back-end resource.

NOTE

Previously the Microsoft account system (personal accounts) did not support the "known client applications" field, nor could it show combined consent. This has been added and all apps in the Microsoft identity platform can use the known client application approach for getting consent for OBO calls.

.default and combined consent

The middle tier application adds the client to the [known client applications list](#) (`knownClientApplications`) in its manifest. If a consent prompt is triggered by the client, the consent flow will be both for itself and the middle tier application. On the Microsoft identity platform, 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.

The resource service (API) identified in the request should be the API for which the client application is requesting an access token as a result of the user's sign-in. For example,

```
scope=openid https://middle-tier-api.example.com/.default (to request an access token for the middle tier API),  
or scope=openid offline_access .default (when a resource is not identified, it defaults to Microsoft Graph).
```

Regardless of which API is identified in the authorization request, the consent prompt will be a combined consent prompt including all required permissions configured for the client app, as well as all required permissions configured for each middle tier API listed in the client's required permissions list, and which have identified the client as a known client application.

Pre-authorized applications

Resources 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 (`preAuthorizedApplications`) in its manifest - 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](#).

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.

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 implicit grant flow

4/12/2022 • 13 minutes to read • [Edit Online](#)

The Microsoft identity platform supports the OAuth 2.0 Implicit Grant flow as described in the [OAuth 2.0 Specification](#). The defining characteristic of the implicit grant is that tokens (ID tokens or access tokens) are returned directly from the /authorize endpoint instead of the /token endpoint. This is often used as part of the [authorization code flow](#), in what is called the "hybrid flow" - retrieving the ID token on the /authorize request along with an authorization code.

This article describes how to program directly against the protocol in your application to request tokens from Azure AD. When possible, we recommend you use the supported Microsoft Authentication Libraries (MSAL) instead to [acquire tokens and call secured web APIs](#). Also take a look at the [sample apps that use MSAL](#).

TIP

[Run in Postman](#)

Try executing this request and more in Postman -- don't forget to replace tokens and IDs!

Prefer the auth code flow

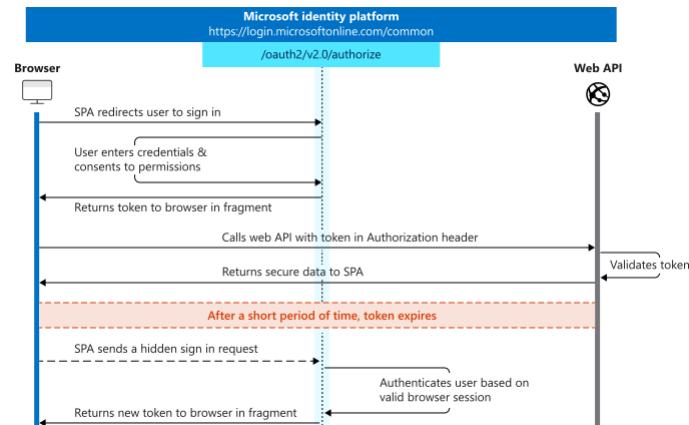
With the plans for [third party cookies to be removed from browsers](#), the [implicit grant flow is no longer a suitable authentication method](#). The [silent SSO features](#) of the implicit flow do not work without third party cookies, causing applications to break when they attempt to get a new token. We strongly recommend that all new applications use the [authorization code flow](#) that now supports single page apps in place of the implicit flow, and that [existing single page apps begin migrating to the authorization code flow](#) as well.

Suitable scenarios for the OAuth2 implicit grant

The implicit grant is only reliable for the initial, interactive portion of your sign in flow, where the lack of [third party cookies](#) cannot impact your application. This limitation means you should use it exclusively as part of the hybrid flow, where your application requests a code as well as a token from the authorization endpoint. This ensures that your application receives a code that can be redeemed for a refresh token, thus ensuring your app's login session remains valid over time.

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.

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 [access tokens](#) in the [Implicit grant and hybrid flows](#) section. If it's not enabled, an `unsupported_response_type` 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%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	TYPE	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 . Critically, for guest scenarios where you sign a user from one tenant into another tenant, you <i>must</i> provide the tenant identifier to correctly sign them into the resource tenant.
<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</code> <code>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> <code>response_type</code> , 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). It can also contain <code>code</code> in place of <code>token</code> to provide an authorization code, for use in the authorization code flow . This <code>id_token+code</code> response is sometimes called the hybrid flow.
<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 (<code>id_tokens</code>), 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 <code>id_token</code> .
<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 <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 that can be used to identify the origin of the request. Only required when an <code>id_token</code> is requested.

PARAMETER	TYPE	DESCRIPTION
<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 will return an 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	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 <code>login_hint</code> optional claim from an earlier sign-in.
<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 parameter 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, forwarding the user to the federation provider for that tenant. Note that this hint prevents guests from signing into this application, and limits the use of cloud credentials like FIDO.

At this point, the user will be asked to enter their credentials and complete the authentication. The Microsoft identity platform 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 authenticates and grants consent, the Microsoft identity platform 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+code` looks like the following (with line breaks for legibility):

```
GET https://localhost/myapp/#  
code=0.AgAAktYV-sfpYE5nQny1W_UKZmH-C9y_G1A  
&id_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVZ2ZEstZnl0aEV1Q...  
&state=12345
```

PARAMETER	DESCRIPTION
<code>code</code>	Included if <code>response_type</code> includes <code>code</code> . This is an authorization code suitable for use in the authorization code flow .
<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.
<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).

PARAMETER	DESCRIPTION
<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 <code>id_tokens</code> , see the id_token reference . Note: Only provided if <code>openid</code> scope was requested and <code>response_type</code> included <code>id_tokens</code> .
<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.

WARNING

Don't attempt to validate or read tokens for any API you don't own, including the tokens in this example, in your code. Tokens for Microsoft services can use a special format that will not validate as a JWT, and may also be encrypted for consumer (Microsoft account) users. While reading tokens is a useful debugging and learning tool, do not take dependencies on this in your code or assume specifics about tokens that aren't for an API you control.

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

IMPORTANT

This part of the implicit flow is unlikely to work for your application as it's used across different browsers due to the [removal of third party cookies by default](#). While this still currently works in Chromium-based browsers that are not in Incognito, developers should reconsider using this part of the flow. In browsers that do not support third party cookies, you will receive an error indicating that no users are signed in, as the login page's session cookies were removed by the browser.

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. You can make the request 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=none&login_hint=myuser@mycompany.com
```

Note that this will work even in browsers without third party cookie support, since you're entering this directly into a browser bar as opposed to opening it within an iframe.

Thanks to the `prompt=none` parameter, this request will either succeed or fail immediately and return to your application. The 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=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2EstZnl0aEV1Q...
&state=12345
&token_type=Bearer
&expires_in=3599
&scope=https%3A%2F%2Fgraph.microsoft.com%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 <code>access_token</code> 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). 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.

In browsers that do not support third party cookies, this will result in an error indicating that no user is signed in.

Send a sign out request

The OpenID Connect `end_session_endpoint` allows your app to send a request to the Microsoft identity platform to end a user's session and clear cookies set by the Microsoft identity platform. 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	TYPE	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	TYPE	DESCRIPTION
<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.

Next steps

- Go over the [MSAL JS samples](#) to get started coding.
- Review the [authorization code flow](#) as a newer, better alternative to the implicit grant.

Microsoft identity platform and OAuth 2.0 Resource Owner Password Credentials

4/12/2022 • 4 minutes to read • [Edit Online](#)

The Microsoft identity platform supports the [OAuth 2.0 Resource Owner Password Credentials \(ROPC\) grant](#), which allows an application to sign in the user by directly handling their password. This article describes how to program directly against the protocol in your application. When possible, we recommend you use the supported Microsoft Authentication Libraries (MSAL) instead to [acquire tokens and call secured web APIs](#). Also take a look at the [sample apps that use MSAL](#).

WARNING

Microsoft recommends you do *not* use the ROPC flow. In most scenarios, more secure alternatives are available and recommended. This flow requires a very high degree of trust in the application, and carries risks which are not present in other flows. You should only use this flow when other more secure flows can't be used.

IMPORTANT

- The Microsoft identity platform only supports ROPC within 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 with ROPC, which means features like SMS sign-in, FIDO, and the Authenticator app won't work with that flow. Use a flow other than ROPC if your app or users require these features.
- 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-premises accounts). If users are full-page redirected to an on-premises identity provider, Azure AD is not able to test the username and password against that identity provider. [Pass-through authentication](#) is supported with ROPC, however.
- An exception to a hybrid identity federation scenario would be the following: Home Realm Discovery policy with AllowCloudPasswordValidation set to TRUE will enable ROPC flow to work for federated users when on-premises password is synced to cloud. For more information, see [Enable direct ROPC authentication of federated users for legacy applications](#).

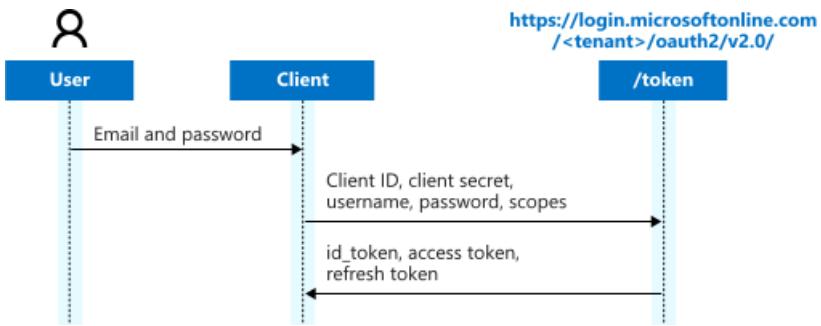
TIP

▶ [Run in Postman](#)

Try executing this request and more in Postman -- don't forget to replace tokens and IDs!

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.

```

// 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 <code>scopes</code> , or permissions, that the app requires. In an interactive flow, the admin or the user must consent to these scopes ahead of time.

PARAMETER	CONDITION	DESCRIPTION
<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.

WARNING

As part of not recommending this flow for use, the official SDKs do not support this flow for confidential clients, those that use a secret or assertion. You may find that the SDK you wish to use does not allow you to add a secret while using ROPC.

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": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZZEstZnl0aEV1Q...",
  "refresh_token": "AwABAAAAAvPM1KaPlrEqdFSBzjqfTGAMxZGUTdM0t4B4...",
  "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 <code>scopes</code> 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](#).

WARNING

Don't attempt to validate or read tokens for any API you don't own, including the tokens in this example, in your code. Tokens for Microsoft services can use a special format that will not validate as a JWT, and may also be encrypted for consumer (Microsoft account) users. While reading tokens is a useful debugging and learning tool, do not take dependencies on this in your code or assume specifics about tokens that aren't for an API you control.

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

For an example of using ROPC, see the [.NET Core console application](#) code sample on GitHub.

Microsoft identity platform and OpenID Connect protocol

4/12/2022 • 16 minutes to read • [Edit Online](#)

OpenID Connect (OIDC) is an authentication protocol built on OAuth 2.0 that you can use to securely sign in a user to an application. When you use the Microsoft identity platform's implementation of OpenID Connect, you can add sign-in and API access to your 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](#).

OpenID Connect extends the OAuth 2.0 *authorization* protocol for 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. It also introduces the [UserInfo endpoint](#), an API that returns information about the user.

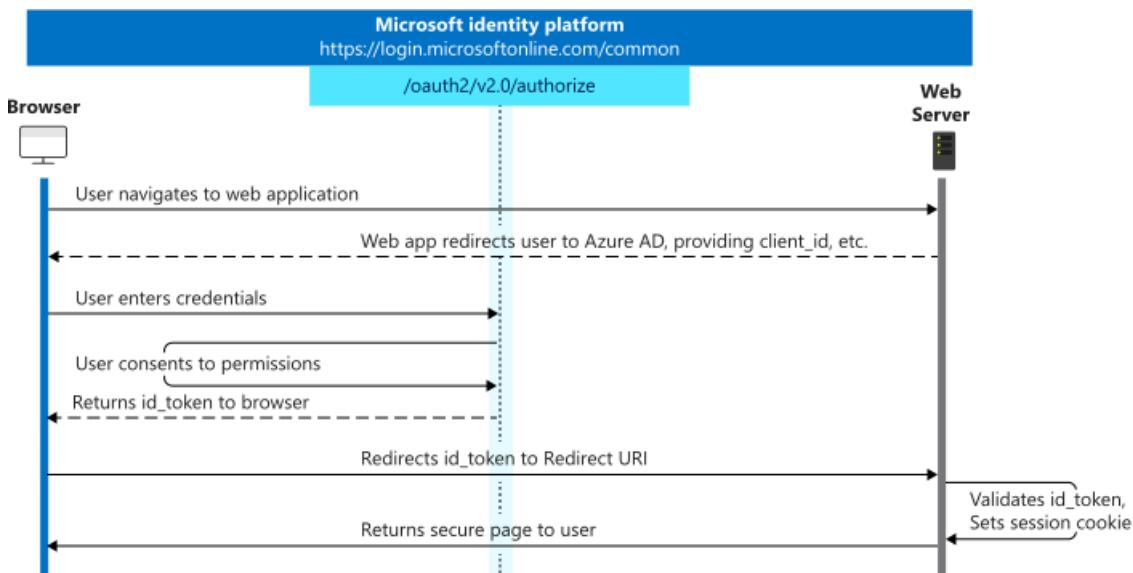
TIP

[▶ Run in Postman](#)

Try executing this request and more in Postman -- don't forget to replace tokens and IDs!

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 ([RFC](#)) 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. You can find this document by appending the discovery document path to the authority URL:

Discovery document path: `/.well-known/openid-configuration`

Authority: `https://login.microsoftonline.com/{tenant}/v2.0`

The `{tenant}` can take one of four values:

VALUE	DESCRIPTION
<code>common</code>	Users with both a personal Microsoft account and a work or school account from Azure AD can sign in to the application.
<code>organizations</code>	Only users with work or school accounts from Azure AD can sign in to the application.
<code>consumers</code>	Only users with a personal Microsoft account can sign in to the application.
<code>8eae023-2b34-4da1-9baa-8bc8c9d6a490</code> or <code>contoso.onmicrosoft.com</code>	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, <code>9188040d-6c67-4c5b-b112-36a304b66dad</code> , in place of the <code>consumers</code> tenant.

The authority differs across national clouds - e.g. <https://login.microsoftonline.de> for the Azure AD Germany instance. If you do not use the public cloud, please review the [national cloud endpoints](#) to find the appropriate one for you. Ensure that the tenant and `/v2.0/` are present in your request so you can use the v2.0 version of the endpoint.

TIP

Try it! Click <https://login.microsoftonline.com/common/v2.0/.well-known/openid-configuration> to see the `common` configuration.

Sample request

To call the userinfo endpoint for the common authority on the public cloud, use the following:

```
GET /common/v2.0/.well-known/openid-configuration
Host: login.microsoftonline.com
```

Sample response

The metadata is a simple JavaScript Object Notation (JSON) document. See the following snippet for an example. The 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",
  "userinfo_endpoint": "https://graph.microsoft.com/oidc/userinfo",
  "subject_types_supported": [
    "pairwise"
  ],
  ...
}
```

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}/v2.0/.well-known/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.

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%2F%2Flocalhost%2Fmyapp%2F
&response_mode=form_post
&scope=openid
&state=12345
&nonce=678910
```

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 . Critically, for guest scenarios where you sign a user from one tenant into another tenant, you <i>must</i> provide the tenant identifier to correctly sign them into the resource tenant.
<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 URIs you registered in the portal, except that it must be URL-encoded. If not present, the endpoint will pick one registered <code>redirect_uri</code> at random to send the user back to.
<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.

PARAMETER	CONDITION	DESCRIPTION
<code>nonce</code>	Required	A value included in the request, generated by the app, that will be included in the resulting id_token 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> , <code>consent</code> , and <code>select_account</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> parameter is the opposite, and should be paired with a <code>login_hint</code> to indicate which user must be signed in. These parameters ensure that the user isn't presented with any interactive prompt at all. If the request can't be completed silently via single sign-on (because no user is signed in, the hinted user isn't signed in, or there are multiple users signed in and no hint is provided), the Microsoft identity platform 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. Finally, <code>select_account</code> shows the user an account selector, negating silent SSO but allowing the user to pick which account they intend to sign in with, without requiring credential entry. You cannot use <code>login_hint</code> and <code>select_account</code> together.
<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 <code>login_hint</code> optional claim from an earlier sign-in.
<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 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 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 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=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik1uQ19WWmNB...&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.

Error code	Description	Client action
<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.
<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 because of 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 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 always sufficient to authenticate the user; you may also need to validate the `id_token`'s signature and verify the claims in the token per your app's requirements. Like all OIDC platforms, the Microsoft identity platform uses [JSON Web Tokens \(JWTs\)](#) and public key cryptography to sign ID tokens and verify that they're valid.

Not all apps benefit from verifying the ID token - native apps and single page apps, for instance, rarely benefit from validating the ID token. Someone with physical access to the device (or browser) can bypass the validation in many ways - from editing the web traffic to the device to provide fake tokens and keys to simply debugging the application to skip the validation logic. On the other hand, web apps and APIs using an ID token to authorization must validate the ID token carefully since they are gating access to data.

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:

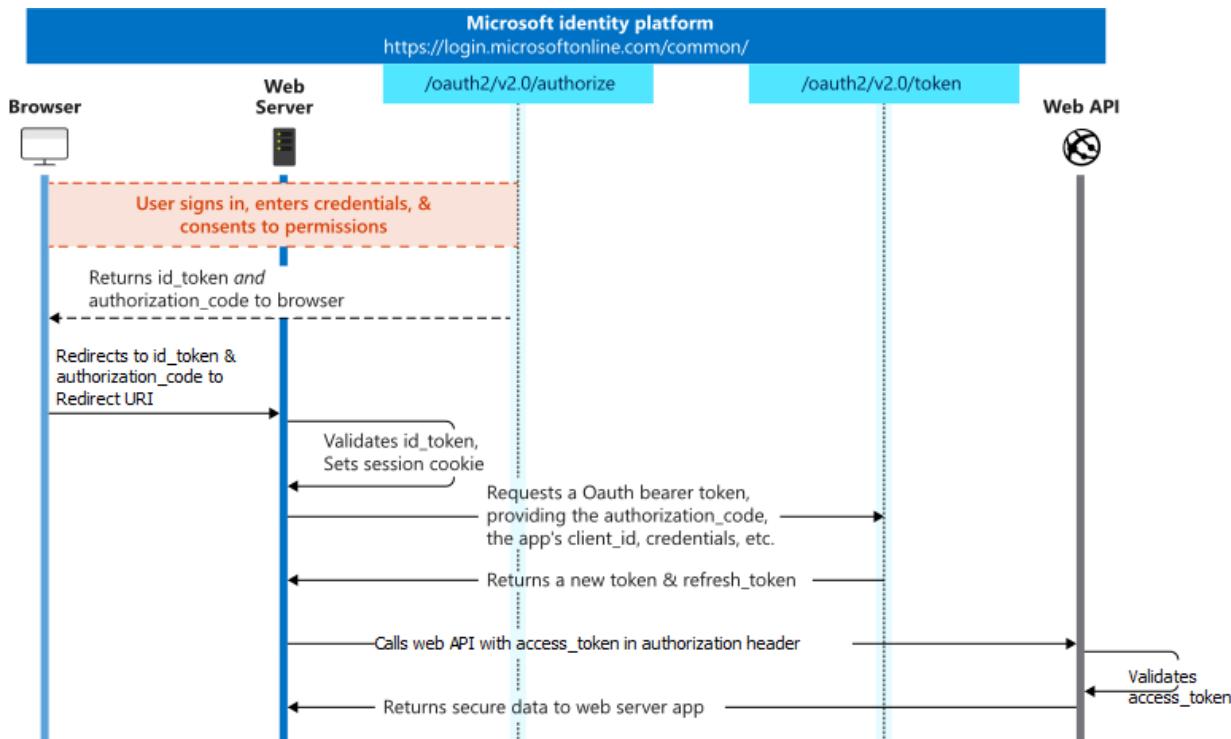
- Ensuring the user/organization has signed up for the app.
- Ensuring the user has proper authorization/privileges
- Ensuring a certain strength of authentication has occurred, such as [multi-factor authentication](#).

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.

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 an access token to call UserInfo

To acquire a token for the OIDC UserInfo endpoint, modify the sign-in request:

```
// Line breaks are for legibility only.

GET https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e          // Your registered Application ID
&response_type=id_token%20token                         // this will return both an id_token and an access
token
&redirect_uri=http%3A%2F%localhost%2Fmyapp%2F           // Your registered redirect URI, URL encoded
&response_mode=form_post                                  // 'form_post' or 'fragment'
&scope=openid+profile+email                            // `openid` is required. `profile` and `email`
provide additional information in the UserInfo endpoint the same way they do in an ID token.
&state=12345                                            // Any value, provided by your app
&nonce=678910                                           // Any value, provided by your app
```

You can also use the [authorization code flow](#), the [device code flow](#), or a [refresh token](#) in place of `response_type=token` to get a token for your app.

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 token in the address bar. Note that this request uses `response_mode=fragment` for demonstration purposes only - for a webapp we recommend using `form_post` for additional security where possible. <https://login.microsoftonline.com/common/oauth2/v2.0/authorize...>

Successful token 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
access_token=eyJ0eXAiOiJKV1QiLCJub25jZSI6I.....
&token_type=Bearer
&expires_in=3598
&scope=email+openid+profile
&id_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI.....
&state=12345
```

Response parameters mean the same thing regardless of the flow used to acquire them.

PARAMETER	DESCRIPTION
<code>access_token</code>	The token that will be used to call the UserInfo endpoint.
<code>token_type</code>	Always "Bearer"
<code>expires_in</code>	How long until the access token expires, in seconds.
<code>scope</code>	The permissions granted on the access token. Note that since the UserInfo endpoint is hosted on MS Graph, there may be additional Graph scopes listed here (e.g. user.read) if they were previously granted to the app. That's because a token for a given resource always includes every permission currently granted to the client.
<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>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.

WARNING

Don't attempt to validate or read tokens for any API you don't own, including the tokens in this example, in your code. Tokens for Microsoft services can use a special format that will not validate as a JWT, and may also be encrypted for consumer (Microsoft account) users. While reading tokens is a useful debugging and learning tool, do not take dependencies on this in your code or assume specifics about tokens that aren't for an API you control.

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](#).

Calling the UserInfo endpoint

Review the [UserInfo documentation](#) to look over how to call the UserInfo endpoint with this token.

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 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.

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%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. This URL must match one of the redirect URIs registered for your application in the app registration portal.

PARAMETER	CONDITION	DESCRIPTION
<code>logout_hint</code>	Optional	Enables sign-out to occur without prompting the user to select an account. To use <code>logout_hint</code> , enable the <code>login_hint optional claim</code> in your client application and use the value of the <code>login_hint</code> optional claim as the <code>logout_hint</code> parameter. Do not use UPNs or phone numbers as the value of the <code>logout_hint</code> parameter.

Single sign-out

When you redirect the user to the `end_session_endpoint`, the Microsoft identity platform 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 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.

Next steps

- Review the [UserInfo documentation](#)
- Learn how to [customize the values in a token](#) with data from your on-premises systems.
- Learn how to [include additional standard claims in tokens](#).

Microsoft identity platform access tokens

4/12/2022 • 26 minutes to read • [Edit Online](#)

Access tokens enable clients to securely call protected web APIs, and are used by web APIs to perform authentication and authorization. Per the OAuth specification, access tokens are opaque strings without a set format - some identity providers (IDPs) use GUIDs, others use encrypted blobs. The Microsoft identity platform uses various access token formats depending on the configuration of the API that accepts the token. [Custom APIs registered by developers](#) on the Microsoft identity platform can choose from two different formats of JSON Web Tokens (JWTs), called "v1" and "v2", and Microsoft-developed APIs like Microsoft Graph or APIs in Azure have other proprietary token formats. These proprietary formats might be encrypted tokens, JWTs, or special JWT-like tokens that will not validate.

Clients must treat access tokens as opaque strings because the contents of the token are intended for the resource (the API) only. For validation and debugging purposes *only*, developers can decode JWTs using a site like [jwt.ms](#). Be aware, however, that the tokens you receive for a Microsoft API might not always be a JWT, and that you can't always decode them.

For details on what's inside the access token, clients should use the token response data that's returned with the access token to your client. When your client requests an access token, the Microsoft identity platform 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.

See the following sections to learn how your API can validate and use the claims inside an access token.

NOTE

All documentation on this page, except where noted, applies only to tokens issued for APIs you've registered. It does not apply to tokens issued for Microsoft-owned APIs, nor can those tokens be used to validate how the Microsoft identity platform will issue tokens for an API you create.

Token formats and ownership

v1.0 and v2.0

There are two versions of access tokens available in the Microsoft identity platform: v1.0 and v2.0. These versions govern what claims are in the token, ensuring that a web API can control what their tokens look like. Web APIs have one of these selected as a default during registration - v1.0 for Azure AD-only apps, and v2.0 for apps that support consumer accounts. This is controllable by applications using the `accessTokenAcceptedVersion` setting in the [app manifest](#), where `null` and `1` result in v1.0 tokens, and `2` results in v2.0 tokens.

What app is a token "for"?

There are two parties involved in an access token request: the client, who requests the token, and the resource (the API) that accepts the token when the API is called. The `aud` claim in a token indicates the resource the token is intended for (its *audience*). Clients use the token but should not understand or attempt to parse it. Resources accept the token.

The Microsoft identity platform supports issuing any token version from any version endpoint - they are not related. This is why a resource setting `accessTokenAcceptedVersion` to `2` means that a client calling the v1.0 endpoint to get a token for that API will receive a v2.0 access token. Resources always own their tokens (those with their `aud` claim) and are the only applications that can change their token details. This is why changing the access token [optional claims](#) for your *client* does not change the access token received when a token is requested for `user.read`, which is owned by the Microsoft Graph resource.

Sample tokens

v1.0 and v2.0 tokens look similar and contain many of the same claims. An example of each is provided here. These example tokens will not [validate](#), however, as the keys have rotated prior to publication and personal information has been removed from them.

v1.0

```

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzIiNiIsImtpZC16Imk2bEdrM0ZaenHSY1ViMkMzbkVRN3N5SEpsWSIsImtpZC16Imk2bEdrM0ZaenHSY1ViMkMzbkVRN3N5SEpsWSJ9.eyJhdWQiOiJlZjFKYTlkNC1mZjc3LTrjM2UtYTAwNS04NDbjM2Y4MzA3NDUiLCJpc3Mi0iJodHRwczovL3N0cy53aw5kb3dzLm51dC9mYt2DY5Mi1l0WM3LTQ0NjAtYtMy0yOWYyOTUyMjIyOS8iLCJpYXQ0jE1MzcymxMDysIm5zIi6MTUzNzIzmZzEwNiwiZXhwIjoxNTM3MjM3MDA2LCJhY3IiOixiwiYmlvIjoiQvhRQwvOE1bQUFBrm0rRS9RVEcrZ0ZuVnhMaldkdzhLKzYxQUdyU091TU1GNmViYu1qN1hPm01ibU0zKdtkc95Rct0dlpSR24yVmFUL2tES1h3NE1JaHJnr1ZxNKJu0hdMwg9UMUxrSVorRnpRVmtKUFBUU9NETjWHFTBENNUERTL0RpQ0RnRTIyM1RjbU12V05hRU1hvU9Uc01Hd1RRPT0iLCJhbXi0lsid2lhIl0sImFwcGlkIjoiNzVkmU3N2YtMTBhMy00ZTU5LTg1ZmQtOGMxMjc1NDRmMTdjIiwiYXBwaWRhY3IiOiiwiZw1haWWi0iJBymVMaUBtaWNyB3NvZnQuY29tIiwiZmFtaWx5X25hbWUioiJmaW5jb2xuIiwiZ212ZW5fbmFtZSI6IkFizSAoTVNGVCKiLCJpZHAIoijodHRwczovL3N0cy53aw5kb3dzLm51dC83MmY50DhiZi04NmYxLTQxWyytOTFhYi0yZDdjzDAxMjIyNdcvIiwiAxBhZGRyIjoiMjIyLjIyM14yMjIuMjIiLCJuYW11IjoiYwJ1bGkiLCJvaWQioiIwMjIyM2I2Yi1hYTfklTQyZDQt0VvJMC0xYjIyJkx0TQ0MzgjICjaCI6IkkilCJzY3Ai0iJ1c2Vyx2ltcGVyc29uYXRpb24iLCJzdwIi0ijsM19yb01TUVUyMjJiVuXTOXlpMmswWHBxcE9ptXo1SDNaQUNVMDu1WEiLCJ0aWQioiJmYtE1ZDV5M11l0WM3LTQ0NjAtYTc0My0yOWyOTU2ZmQ0Mjk1LCJ1bmlxdWVfbmFtZSI6ImFizWxpQG1pY3Jvc29mdC5jb20iLCJ1dGkIoiJGVnNHeF1YSTMwLVR1aWt1dVvvRkFBiwidmVyIjoiMS4wIn0.D3H6pMuTQnoJAGq6AHd

```

View this v1.0 token in [JWT.ms](#).

v2.0

```

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzIiNiIsImtpZC16Imk2bEdrM0ZaenHSY1ViMkMzbkVRN3N5SEpsWSJ9.eyJhdWQiOiI2ZTc0MTcyYi1iZTU2LTQ4NDMtOWZmNC11NjZhMzliYjEyZTMiLCJpc3Mi0iJodHRwczovL2xvZ2luLm1pY3Jvc29mdG9ubGluzS5jb20vNzJm0Tg4YmYtODZmM00MWfMLTkxYWi0tMmQ3Y2QwMTFkYjQ3L3YyLjA1LCJpYXQ0jE1MzcymzEwNDgsIm5zIi6MTUzNzIzMTA00CwiZXhwIjoxNTM3MjM0OTQ4LCJhaW8i0iJBWFFBaS84SUFBQF0QnfA TG8zQ2hNaWY2S09udHRSQjd1QnE0l0RjY1F6amNKR3hQwXkvQzNqRGFOR3hYZDZ3Tk1JVkdSz2h0Um53jfsT2NBbk5aY2p2a295ckZ4Q3R0djmZmTQwUm1vt0ZKNGJDQ0dWdw9DyWcxdsU9UVDiyMjIyZ0h3TFBZUS91Zjc5UVgrMETjAwpkcm1wNj1SY3R6bVE9PSIisMf6cCI6IjZ1NzQxJ1LWj1NTYtNDg0My0zMy0LWU2NmEz0WJiMTJ1MyIsImF6cGFjciI6IjA1LCJuYW11IjoiQWJ1IEpxbmVnbG4iLCJvaWQioiI20TAyMjIzS1mZjFhLTrkNTYtYwjkMS03ZTrmN2QzOGU0NzQ1LCJwcmVmZXJyZWRfdXN1cm5hbWU0iJhYmVsauBtaNyb3NvZnQuY29tIiwiCmgioiJ3jiwi2NwIjoiYwnjZxNz2fxZ3VzZxi1LCJzdWIi0iJIS1pwZmFieVdhZGVPb3VzbG10anJJUTmZ1RtmjIyWDVyc1YzeERxZktRiwiidGkIjoiNzJm0Tg4YmYt0DZmMS00MWfMLTkxYWi0tMmQ3Y2QwMTFkYjQ3IiwiidXRpIjoiZnfpQnFytFBqMGVRYTgyUy1JWUZBQSIsInZlcIi6IjIuMCJ9.pj4N-w_3Us9DrBLfpCt

```

View this v2.0 token in [JWT.ms](#).

Claims in access tokens

JWTs (JSON Web Tokens) 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. Your app shouldn't take a dependency on a claim being present. Examples include `pwd_exp` (not every tenant requires passwords to expire) and `family_name` ([client credential](#) flows are on behalf of applications that don't have names). Claims used for access token validation will always be present.

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>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 can be used to validate this token's signature. 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 or GUID	Identifies the intended recipient of the token - its audience. Your API must validate this value and reject the token if the value doesn't match. In v2.0 tokens, this is always the client ID of the API, while in v1.0 tokens it can be the client ID or the resource URI used in the request, depending on how the client requested the token.
<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 can 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 <code>Issuer</code> 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.
<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.

CLAIM	FORMAT	DESCRIPTION
<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>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>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>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, however, and in human-readable UI as a username. The <code>profile</code> scope is required in order to receive this claim. Present only in v2.0 tokens.
<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 credential flow (v1.0 , v2.0) 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 Azure AD built-in roles . 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 Microsoft 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.</p> <p>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 Microsoft Graph endpoint containing the list of groups for the user.</p>

CLAIM	FORMAT	DESCRIPTION
<code>hasgroups</code>	Boolean	<p>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 six or more groups). Indicates that the client should use the Microsoft Graph API to determine the user's groups (https://graph.microsoft.com/v1.0/users/{userID}/getMembers).</p>
<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.microsoft.com/v1.0/users/{userID}/getMembers" }</pre>
<code>sub</code>	String	<p>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).</p>

CLAIM	FORMAT	DESCRIPTION
<code>oid</code>	String, a GUID	The immutable identifier for the "principal" of the request - the user or service principal whose identity has been verified. In ID tokens and app+user tokens, this is the object ID of the user. In app-only tokens, this is the object ID of the calling service principal. It can also be used to perform authorization checks safely and as a key in database tables. This ID uniquely identifies the principal 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 principals, the <code>profile</code> scope is required in order to receive this claim for users. 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 tenant that the user is signing in to. For work and school accounts, the GUID is the immutable tenant ID of the organization that the user is signing in to. For sign-ins to the personal Microsoft account tenant (services like Xbox, Teams for Life, or Outlook), the value is <code>9188040d-6c67-4c5b-b112-36a304b66dad</code> . To receive this claim, your app must request the <code>profile</code> scope.
<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>	String	Token identifier claim, equivalent to <code>jti</code> in the JWT specification. Unique, per-token identifier that is case-sensitive.
<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.

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, and only 6 if issued via the implicit flow), 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 Microsoft Graph API to retrieve the user's group membership.

```
{
  ...
  "_claim_names": {
    "groups": "src1"
  },
  {
    "_claim_sources": {
      "src1": {
        "endpoint": "[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.

VALUE	DESCRIPTION
<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.
<code>wia</code>	Windows Integrated Authentication
<code>mfa</code>	Multi-factor authentication was used. When this is present the other authentication methods will also be included.
<code>ngcmfa</code>	Equivalent to <code>mfa</code> , used for provisioning of certain advanced credential types.
<code>wiaormfa</code>	The user used Windows or an MFA credential to authenticate.
<code>none</code>	No authentication was done.

Access token lifetime

The default lifetime of an access token is variable. When issued, an access token's default lifetime is assigned a random value ranging between 60-90 minutes (75 minutes on average). The variation improves service resilience by spreading access token demand over a period of 60 to 90 minutes, which prevents hourly spikes in traffic to Azure AD.

Tenants that don't use Conditional Access have a default access token lifetime of 2-hours for clients such as Microsoft Teams and Microsoft 365.

You can adjust the lifetime of an access token to control how often the client application expires the application session, and how often it requires the user to re-authenticate (either silently or interactively). Customers that wish to override default access token lifetime variation can set a static default access token lifetime by using [Configurable token lifetime \(CTL\)](#).

Default token lifetime variation is applied to organizations that have Continuous Access Evaluation (CAE) enabled, even if CTL policies are configured. The default token lifetime for long lived token lifetime ranges from 20 to 28 hours. When the access token expires, the client must use the refresh token to (usually silently) acquire a new refresh token and access token.

Organizations that use [Conditional Access sign-in frequency \(SIF\)](#) to enforce how frequently sign-ins occur cannot override default access token lifetime variation. When using SIF, the time between credential prompts for a client is the token lifetime (ranging from 60 - 90 minutes) plus the sign-in frequency interval.

Here's an example of how default token lifetime variation works with sign-in frequency. Let's say an organization sets sign-in frequency to occur every hour. The actual sign-in interval will occur anywhere between 1 hour to 2.5 hours since the token is issued with lifetime ranging from 60-90 minutes (due to token lifetime variation).

If a user with a token with a one hour lifetime performs an interactive sign-in at 59 minutes (just before the sign-in frequency being exceeded), there is no credential prompt because the sign-in is below the SIF threshold. If a new token is issued with a lifetime of 90 minutes, the user would not see a credential prompt for an additional hour and a half. When a silent renewal attempted of the 90-minute token lifetime is made, Azure AD will require a credential prompt because the total session length has exceeded the sign-in frequency setting of 1 hour. In this example, the time difference between credential prompts due to the SIF interval and token lifetime variation would be 2.5 hours.

Validating tokens

Not all apps should validate tokens. Only in specific scenarios should apps validate a token:

- [Web APIs](#) must validate access tokens sent to them by a client. They must only accept tokens containing their

`aud` claim.

- Confidential web apps like ASP.NET Core must validate ID tokens sent to them via the user's browser in the hybrid flow, before allowing access to a user's data or establishing a session.

If none of the above scenarios apply, your application will not benefit from validating the token, and may present a security and reliability risk if decisions are made based on the validity of the token. Public clients like native apps or SPAs don't benefit from validating tokens - the app communicates directly with the IDP, so SSL protection ensures the tokens are valid.

APIs and web apps must only validate tokens that have an `aud` claim that matches their application; other resources may have custom token validation rules. For example, tokens for Microsoft Graph won't validate according to these rules due to their proprietary format. Validating and accepting tokens meant for another resource is an example of the [confused deputy](#) problem.

If your application needs to validate an `id_token` or an `access_token` according to the above, your app should first validate the token's signature and issuer 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 following information is provided for those who wish to understand the underlying process. 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. 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. For more information about Azure AD authentication libraries and code samples, see the [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 RS256. 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 that correspond to the private 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 the public key, from this document, which corresponds to the private key that has been used to sign a particular token. It can then do signature validation using the correct public key and the indicated algorithm.

NOTE

We recommend using the `kid` claim to validate your token. Though v1.0 tokens contain both the `x5t` and `kid` claims, v2.0 tokens contain only the `kid` claim.

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.

Validate the token is meant for you

- Use the `aud` claim to ensure that the user intended to call your application. If your resource's identifier is not in the `aud` claim, reject it.

Validate the user has permission to access this data

- Use the `roles` and `wids` claims to validate that the user themselves has authorization to call your API. For example, an admin may have permission to write to your API, but not a normal user.
- Check that the `tid` inside the token matches the tenant ID used to store the data in your API.
 - When a user stores data in your API from one tenant, they must sign into that tenant again to access that data. Never allow data in one tenant to be accessed from another tenant.
- Use the `amr` 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.

Never use `email` or `upn` claim values to determine whether the user in an access token should have access to data! Mutable claim values like these can change over time, making them insecure and unreliable for authorization.

Do use immutable claim values `tid` and `sub` or `oid` as a combined key for storing for uniquely identifying your API's data and determining whether a user should be granted access to that data.

- Good: `tid` + `sub`
- Better: `tid` + `oid` - the `oid` is consistent across applications, so an ecosystem of apps can audit user access to data, for instance.

Do not use mutable, human-readable identifiers like `email` or `upn` for uniquely identifying data.

- Bad: `email`
- Bad: `upn`

Validate that the application that signed in the user has permission to access this data

- Use the `scp` claim to validate that the user has granted the calling app permission to call your API.
- Ensure the calling client is allowed to call your API using the `appid` claim.

User and application tokens

Your application may receive tokens for user (the flow usually discussed) 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:

- Use `roles` to see permissions that have been granted to the subject of the token.
- Use `oid` or `sub` to validate that the calling service principal is the expected one.

If your app needs to distinguish between app-only access tokens and access tokens for users, use the `idtyp` optional claim. By adding the `idtyp` claim to the `accessToken` field, and checking for the value `app`, you can detect app-only access tokens. ID tokens and access tokens for users will not have the `idtyp` claim included.

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

Using [token lifetime configuration](#), the lifetime of refresh tokens can be altered. It is normal and expected for some tokens to go without use (e.g. the user does not open the app for 3 months) and therefore expire. Apps will encounter scenarios where the login server rejects a refresh token due to its age.

- `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

Refresh tokens can be revoked by the server due to a change in credentials, or due to use or admin action.

Refresh tokens fall into two classes - those issued to confidential clients (the rightmost column) and those issued to public clients (all other columns).

CHANGE	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 a user via PowerShell	Revoked	Revoked	Revoked	Revoked	Revoked
Single sign-out (v1.0, v2.0) on web	Revoked	Stays alive	Revoked	Stays alive	Stays alive

Non-password-based

A *non-password-based* login is one where the user didn't type in a password to get it. Examples of non-password-based login include:

- Using your face with Windows Hello
- FIDO2 key
- SMS
- Voice
- PIN

Check out [Primary Refresh Tokens](#) for more details on primary refresh tokens.

Next steps

- Learn about [id_tokens](#) in Azure AD.
- Learn about permission and consent ([v1.0](#), [v2.0](#)).

Microsoft identity platform ID tokens

4/12/2022 • 12 minutes to read • [Edit Online](#)

The ID token is the core extension that [OpenID Connect](#) makes to OAuth 2.0. ID tokens are issued by the authorization server and contain claims that carry information about the user. They can be sent alongside or instead of an access token. Information in ID Tokens allows the client to verify that a user is who they claim to be. ID tokens are intended to be understood by third-party applications. ID tokens should not be used for authorization purposes. [Access tokens](#) are used for authorization. The claims provided by ID tokens can be used for UX inside your application, as [keys in a database](#), and providing access to the client application.

Prerequisites

The following article will be beneficial before going through this article:

- OAuth 2.0 and OpenID Connect protocols on the Microsoft identity platform

Claims in an ID token

ID tokens are [JSON web tokens \(JWT\)](#). These ID tokens consist of a header, payload, and signature. The header and signature are used to verify the authenticity of the token, while the payload contains the information about the user requested by your client. The v1.0 and v2.0 ID tokens have differences in the information they carry. The version is based on the endpoint from where it was requested. While existing applications likely use the Azure AD endpoint (v1.0), new applications should use the "Microsoft identity platform" endpoint(v2.0).

- v1.0: Azure AD endpoint: <https://login.microsoftonline.com/common/oauth2/authorize>
 - v2.0: Microsoft identity Platform endpoint: <https://login.microsoftonline.com/common/oauth2/v2.0/authorize>

Sample v1.0 ID token

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6IjdfWnVmMXR2a3dMeF1hSFMzcTzsVwPvWUlHdyIsImtpZCI6IjdfWnVmMXR2a3dm
eF1hSFMzcTzsVwPvWUlHdyJ9.yejHjdWQio1jImTRhNzUwNs05NmU5LTQ5mjctOTfL0C0wNjAxZDbmYzljyWEiLCjcpc3Mi0iJodHRwczoV3N
0cy53aW5kb3dzLm51dC9mYTE1ZDy5Mi110wM3LTQ0NjAtYTc0My0yOWy0TU2zNQ0MjkviIwiawF0IjoxNTM2Mjc1MTI0LClJuYmY0jE1MzY
yNzUxMqJsImV4cCI6MTUzNjI30TAyNCwiYwlvIjoiQvhRQWkv0eLBQFbcXhzdUIrUjREmHJGUxFPRVRPNf1kWgJMRdlrWjh4zL1hhZGVBTB
RMk5rT1Q1axPzmN1d2JXU1hodvNTajZvVDVoetJENldxQXBCNwpLQTzA1o5ay9TvtI3dvY5Y2v0WGZMT3RwTnR0Z2s1RGncdGsrtExzdHo
vSmcrZ1lsBxy5Y1lVVNFhscGhUyZD0DZKbw0rxRkN3PT0i1CJhbXI0lsicnNhI0sImTvYwlsIjoiYwJlbG1Abwljcm9zb2Z0LmNbVsIsImZ
hbWlsev9uYuW1ljoiTglUy29sbisImsdpdmvux25hbWuIo1jByMu1lCjzpHaioIjodHRwczoV3N0cy53aw5kb3dzLm51dc83MmY50Dhzi0
4NmYxLTQxYWt0TFhYi0zDzdjzDAXMwriNdCviwiAxBhZGrYi0jmtMxmJewNy4ymjuiMjIiLCjuYw11i0jYwJlbGk1LcJub25jZSiIje
yMzUyMyIsIm9pZC16Ija10DMyzJz1LwfhwMQtndjkNc05zwNmlTf1Mmj0Ite5NDQzOciSInj0joiSSiSInh1y1i61jvf5jlyu3Nz0c1dnR
fswN1Nv1Lk5MOHYYjhMrjRgc2dfS29vQzJSS1e1Cj0awQ0i0jMyTE1ZDy5Mi110wM3LTQ0NjAtYTc0My0yOWy0TU2zNQ0MjkviLCj1bmL
xdWVfbmFTZSi6IkFzUxpQ61pY3Jvc29mdC5jb20iLCj1dgK0i1jMeGVfNdzHcvRrT3BHU2zUbG40RUFBIwidmVYi0joiMS4wIn0=.UjQrcA
6qn2xb57qzGx_-
D3HcPhQbMOKPDxuas1yKRLNerWD8xkxJLNlVRdASHqEcPyDctbdHccu6Dpkq5f0ibcaQFhejQncAbidJCTz0B2bAbduCTqAzdt9pdgqvMBn
VH1xk3SCM6d4BbtT4BkLLj102LasX7vrknaSjE_C5D17Fg4WrZPw0hII1dB0HEZ_qpNaYXEiy-o94UJ94zCr07GgrqMsFyQqFR7kn-
mn68AjvLcgwsFzvyr_yIK75S_K37vC3QryQ7cNoafDe9upql_6pB2ybMvlwPs_DmbJ8g0om-sPlwyn74Cc1tW3ze-
Xptw_2uwdPgWyafulAfq60

View this v1.0 sample token in [jwt.ms](#)

Sample v2.0 ID token

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IjFMVE16YWtpaGlSbGFf0HoYQkVkv1hlv01xbyJ9.ejY2ZXli0iIyLjAiLCJpc3M
i0iJodHRwczovL2xvZ2luLm1pY3Jvc29mdG9ubGluZS5jb20vOTEyMja0MGQtNm2Nyo0YzViLWIxMTiTmZmZhMzA0YjY2ZGfkL3YyLjA1LCJ
zdWiiOjBQUFBQUBQFBQUBQFBQFJa3pxRlzyU2FTYUZieTc4MmjIdGFRIiwixXVkiJoiNmniMDQwMTgtYTNmNs00NmE3LWI5OTU
tOTQwYzc4ZjvhZWYzIiwiZxHwijoNTM2MzYxNDExLCjpxYXQi0jE1MzYyNzQ3MTEsIm5iZiI6MTUzNjI3NDcxMSwibmFTZSi6IkFizSBMaw5
jb2xuIiwichJ1ZmVycmkvX3VzZxJuYw11IjoiQWjlTG1Ab1Ljcm9zb2Z0LnvbSISim9pZC16IjAwMDAwLTAWMDAtMDAwMC02NmYzLTM
zMzJ1Y2E3ZWE4MSIsInRpZC16IjkxMjIwNDBKLTzNjctNGM1Y11MTEyLTMyT2YTMyNG12NmRhZCISim5vbnnIjoiMT1NTzIiwiYw1Ij0
iRGYVvZYVZDTfpeCFsTUNXTVPNSKjJyRmF0emNHZnZGR2hqs3Y4cTvnMhg3MzJkUjVnQjCvaXN2R1FPN11XQn1qZDhpUURMcSf1R2J2JRGFrExA
1bw5PcmNkcUh1LwnBuHr1cFFtUAnQAQu1aOGpZIn01AFWw-CK5nR0wS1ltmZGz2LduKvhqSOpmt5QsmVo9Y59cLhRxPvb8n-
559CrhZ66_31_UbeKoz6121zJ_8m9FFShSdJoalQr54CreGIjVjtmS3EK9a75BbcplMpu1tfygow39tfjy7EVNW9p1luVrTgVk71YL
prvfzw-CIqw3gHC-T7IK_m_xkr081NERBtaecwhTeN4chPC4W3jdwm_l1xzc48yoQd81L9-
ImX98Egypfr1bm0IBL5spFzL63DZIRRjou8vecJvj1mq-IuHgt0Macx8xjdYLP-Kuu2d9MbNpkCkjuz7p8gwTL5B7n1Udh_dmSviPWrw

View this v2.0 sample token in [jwt.ms](#)

All JWT claims listed below appear in both v1.0 and v2.0 tokens unless stated otherwise.

三一七

The table below shows header claims present in ID tokens.

CLAIM	FORMAT	DESCRIPTION
<code>typ</code>	String - always "JWT"	Indicates that the token is a JWT token.
<code>alg</code>	String	Indicates the algorithm that was used to sign the token. Example: "RS256"
<code>kid</code>	String	Specifies the thumbprint for the public key that can be used to validate this token's signature. Emitted in both v1.0 and v2.0 ID 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 ID tokens for compatibility purposes.

Payload claims

The table below shows the claims that are in most ID tokens by default (except where noted). However, your app can use [optional claims](#) to request more claims in the ID token. Optional claims can range from the `groups` claim to information about the user's name.

CLAIM	FORMAT	DESCRIPTION
<code>aud</code>	String, an App ID GUID	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. This value should be validated. The token should be rejected if it fails to match your app's Application ID.
<code>iss</code>	String, an issuer URI	Identifies the issuer, or "authorization server" that constructs and returns the token. It also identifies the Azure AD tenant for which the user was authenticated. If the token was issued by the v2.0 endpoint, the URL will end in <code>/v2.0</code> . The GUID that indicates that the user is a consumer user from a Microsoft account is 9188040d-6c67-4c5b-b112-36a304b66dad . 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>iat</code>	int, a Unix timestamp	"Issued At" indicates when the authentication for this token occurred.
<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 9188040d-6c67-4c5b-b112-36a304b66dad .

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 in certain circumstances, a resource may reject the token before this time. For example, if 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. To understand how to do this validation, see the OpenID Connect specification .
<code>at_hash</code>	String	The access token hash is included in ID tokens only when the ID token is issued from the <code>/authorize</code> endpoint with an OAuth 2.0 access token. It can be used to validate the authenticity of an access token. To understand how to do this validation, see the OpenID Connect specification . This is not returned on ID tokens from the <code>/token</code> endpoint.
<code>aio</code>	Opaque String	An internal claim used by Azure AD to record data for token reuse. Should be ignored.
<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, however, and in human-readable UI as a username. The <code>profile</code> scope is required in order to receive this claim. Present only in v2.0 tokens.
<code>email</code>	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 <code>email optional claim</code> . 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.

CLAIM	FORMAT	DESCRIPTION
<code>name</code>	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 can be changed, and it's designed to be used only for display purposes. The <code>profile</code> scope is required to receive this claim.
<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	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	Represents the tenant that the user is signing in to. For work and school accounts, the GUID is the immutable tenant ID of the organization that the user is signing in to. For sign-ins to the personal Microsoft account tenant (services like Xbox, Teams for Life, or Outlook), the value is <code>9188040d-6c67-4c5b-b112-36a304b66dad</code> . To receive this claim, your app must request the <code>profile</code> scope.

CLAIM	FORMAT	DESCRIPTION
<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>	String	Token identifier claim, equivalent to <code>jti</code> in the JWT specification. Unique, per-token identifier that is case-sensitive.
<code>ver</code>	String, either 1.0 or 2.0	Indicates the version of the id_token.
<code>hasgroups</code>	Boolean	If present, always true, denoting the user is in at least one group. Used in place of the groups 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 Microsoft Graph API to determine the user's groups (https://graph.microsoft.com/v1.0/users/{userID}/getMemb).
<code>groups:src1</code>	JSON object	<p>For token requests that are not limited in length (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.microsoft.com/v1.0/users/{userID}/getMemb" }</pre> <p>For more info, see Groups coverage claim.</p>

Using claims to reliably identify a user (Subject and Object ID)

When identifying a user (say, looking them up in a database, or deciding what permissions they have), it's critical to use information that will remain constant and unique across time. Legacy applications sometimes use fields like the email address, a phone number, or the UPN. All of these can change over time, and can also be reused over time. For example, when an employee changes their name, or an employee is given an email address that matches that of a previous, no longer present employee. Therefore, it is **critical** that your application not use human-readable data to identify a user - human readable generally means someone will read it, and want to change it. Instead, use the claims provided by the OIDC standard, or the extension claims provided by Microsoft - the `sub` and `oid` claims.

To correctly store information per-user, use `sub` or `oid` alone (which as GUIDs are unique), with `tid` used for routing or sharding if needed. If you need to share data across services, `oid + tid` is best as all apps get the same `oid` and `tid` claims for a given user acting in a given tenant. The `sub` claim in the Microsoft identity platform is "pair-wise" - it is unique based on a combination of the token recipient, tenant, and user. Therefore, two apps that request ID tokens for a given user will receive different `sub` claims, but the same `oid` claims for that user.

NOTE

Do not use the `idp` claim to store information about a user in an attempt to correlate users across tenants. It will not function, as the `oid` and `sub` claims for a user change across tenants, by design, to ensure that applications cannot track users across tenants.

Guest scenarios, where a user is homed in one tenant, and authenticates in another, should treat the user as if they are a brand new user to the service. Your documents and privileges in the Contoso tenant should not apply in the Fabrikam tenant. This is important to prevent accidental data leakage across tenants, and enforcement of data lifecycles. Evicting a guest from a tenant should also remove their access to the data they created in that tenant.

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 Microsoft Graph API to retrieve the user's group membership.

```
{
  ...
  "_claim_names": {
    "groups": "src1"
  },
  {
    "_claim_sources": {
      "src1": {
        "endpoint": "[Url to get this user's group membership from]"
      }
    }
  }
  ...
}
```

ID token lifetime

By default, an ID token is valid for one hour - after one hour, the client must acquire a new ID token.

You can adjust the lifetime of an ID token to control how often the client application expires the application session, and how often it requires the user to re-authenticate either silently or interactively. For more information, read [Configurable token lifetimes](#).

Validating an ID token

Validating an ID token is similar to the first step of [validating an access token](#). Your client can check whether the token has been tampered with. It can also validate the issuer to ensure that the correct issuer has sent back the token. Because ID tokens are always a JWT token, many libraries exist to validate these tokens - we recommend you use one of these rather than doing it yourself. Note that only confidential clients (those with a secret) should validate ID tokens. Public applications (code running entirely on a device or network you don't control such as a user's browser or their home network) don't benefit from validating the ID token. This is because a malicious user can intercept and edit the keys used for validation of the token.

To manually validate the token, see the steps details in [validating an access token](#). The following JWT claims should be validated in the ID token After validating the signature on the token. These claims may also be validated 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

- Review the [OpenID Connect](#) flow, which defines the protocols that emit an ID token.
- Learn about [access tokens](#)
- Customize the JWT claims in your ID token using [optional claims](#).

Microsoft identity platform refresh tokens

4/12/2022 • 2 minutes to read • [Edit Online](#)

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. Refresh tokens are also used to acquire extra access tokens for other resources. Refresh tokens are bound to a combination of user and client, but aren't tied to a resource or tenant. As such, a client can use a refresh token to acquire access tokens across any combination of resource and tenant where it has permission to do so. Refresh tokens are encrypted and only the Microsoft identity platform can read them.

Prerequisites

Before reading through this article, it's recommended that you go through the following articles:

- [ID tokens](#) in the Microsoft identity platform.
- [Access tokens](#) in the Microsoft identity platform.

Refresh token lifetime

Refresh tokens have a longer lifetime than access tokens. The default lifetime for the tokens is 90 days and they replace themselves with a fresh token upon every use. As such, whenever a refresh token is used to acquire a new access token, a new refresh token is also issued. The Microsoft identity platform doesn't revoke old refresh tokens when used to fetch new access tokens. Securely delete the old refresh token after acquiring a new one. Refresh tokens need to be stored safely like access tokens or application credentials.

Refresh token expiration

Refresh tokens can be revoked at any time, because of timeouts and revocations. Your app must handle rejections by the sign-in service gracefully when this occurs. This is done by sending the user to an interactive sign-in prompt to sign in again.

Token timeouts

You can't configure the lifetime of a refresh token. You can't reduce or lengthen their lifetime. Configure sign-in frequency in Conditional Access to define the time periods before a user is required to sign in again. Learn more about [Configuring authentication session management with Conditional Access](#).

Not all refresh tokens follow the rules set in the token lifetime policy. Specifically, refresh tokens used in [single page apps](#) are always fixed to 24 hours of activity, as if they have a `MaxAgeSessionSingleFactor` policy of 24 hours applied to them.

Revocation

Refresh tokens can be revoked by the server because of a change in credentials, user action, or admin action. Refresh tokens fall into two classes: tokens issued to confidential clients (the rightmost column) and tokens issued to public clients (all other columns).

CHANGE	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

CHANGE	PASSWORD-BASED COOKIE	PASSWORD-BASED TOKEN	NON-PASSWORD-BASED COOKIE	NON-PASSWORD-BASED TOKEN	CONFIDENTIAL CLIENT TOKEN
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 <i>via PowerShell</i>	Revoked	Revoked	Revoked	Revoked	Revoked
Admin revokes all refresh tokens for a user <i>via PowerShell</i>	Revoked	Revoked	Revoked	Revoked	Revoked
Single sign-out on web	Revoked	Stays alive	Revoked	Stays alive	Stays alive

Next steps

- Learn about [configurable token lifetimes](#)
- Check out [Primary Refresh Tokens](#) for more details on primary refresh tokens.

Configurable token lifetimes in the Microsoft identity platform (preview)

4/12/2022 • 8 minutes to read • [Edit Online](#)

You can specify the lifetime of a access, ID, or SAML token issued by the Microsoft identity platform. 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. However, we currently do not support configuring the token lifetimes for [managed identity service principals](#).

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.

For examples, read [examples of how to configure token lifetimes](#).

NOTE

Configurable token lifetime policy only applies to mobile and desktop clients that access SharePoint Online and OneDrive for Business resources, and does not apply to web browser sessions. To manage the lifetime of web browser sessions for SharePoint Online and OneDrive for Business, use the [Conditional Access session lifetime](#) feature. Refer to the [SharePoint Online blog](#) to learn more about configuring idle session timeouts.

License requirements

Using this feature requires an Azure AD Premium P1 license. To find the right license for your requirements, see [Comparing generally available features of the Free and Premium editions](#).

Customers with [Microsoft 365 Business licenses](#) also have access to Conditional Access features.

Token lifetime policies for access, SAML, and ID tokens

You can set token lifetime policies for access tokens, SAML 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 lifetime of an access token is variable. When issued, an access token's default lifetime is assigned a random value ranging between 60-90 minutes (75 minutes on average). The default lifetime also varies depending on the client application requesting the token or if conditional access is enabled in the tenant. For more information, see [Access token lifetime](#).

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. From an application's perspective, the validity period of the token is specified by the NotOnOrAfter value of the `<conditions ...>` element in the token. After the validity period of the token has ended, 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.

The subject confirmation NotOnOrAfter specified in the `<SubjectConfirmationData>` element is not affected by the Token Lifetime configuration.

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 re-authenticated with the Microsoft identity platform (either silently or interactively).

Token lifetime policies for refresh tokens and session tokens

You can not set token lifetime policies for refresh tokens and session tokens. For lifetime, timeout, and revocation information on refresh tokens, see [Refresh tokens](#).

IMPORTANT

As of January 30, 2021 you can not configure refresh and session token lifetimes. Azure Active Directory no longer honors refresh and session token configuration in existing policies. New tokens issued after existing tokens have expired are now set to the [default configuration](#). You can still configure access, SAML, and ID token lifetimes after the refresh and session token configuration retirement.

Existing token's lifetime will not be changed. After they expire, a new token will be issued based on the default value.

If you need to continue to define the time period before a user is asked to sign in again, configure sign-in frequency in Conditional Access. To learn more about Conditional Access, read [Configure authentication session management with Conditional Access](#).

Configurable token lifetime properties

A token lifetime policy is a type of policy object that contains token lifetime rules. This policy controls how long access, SAML, and ID tokens for this resource are considered valid. Token lifetime policies cannot be set for refresh and session tokens. If no policy is set, the system enforces the default lifetime value.

Access, ID, and SAML2 token lifetime policy properties

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.

For an example, see [Create a policy for web sign-in](#).

Access, ID, and SAML2 token configuration are affected by the following properties and their respectively set values:

- **Property:** Access Token Lifetime

- **Policy property string:** AccessTokenLifetime
- **Affects:** Access tokens, ID tokens, SAML2 tokens
- **Default:**
 - Access tokens: varies, depending on the client application requesting the token. For example, continuous access evaluation (CAE) capable clients that negotiate CAE-aware sessions will see a long lived token lifetime (up to 28 hours).
 - ID tokens, SAML2 tokens: 1 hour
- **Minimum:** 10 minutes
- **Maximum:** 1 day

Refresh and session token lifetime policy properties

Refresh and session token configuration are affected by the following properties and their respectively set values. After the retirement of refresh and session token configuration on January 30, 2021, Azure AD will only honor the default values described below. If you decide not to use [Conditional Access](#) to manage sign-in frequency, your refresh and session tokens will be set to the default configuration on that date and you'll no longer be able to change their lifetimes.

PROPERTY	POLICY PROPERTY STRING	AFFECTS	DEFAULT
Refresh Token Max Inactive Time	MaxInactiveTime	Refresh tokens	90 days
Single-Factor Refresh Token Max Age	MaxAgeSingleFactor	Refresh tokens (for any users)	Until-revoked
Multi-Factor Refresh Token Max Age	MaxAgeMultiFactor	Refresh tokens (for any users)	Until-revoked
Single-Factor Session Token Max Age	MaxAgeSessionSingleFactor	Session tokens (persistent and nonpersistent)	Until-revoked
Multi-Factor Session Token Max Age	MaxAgeSessionMultiFactor	Session tokens (persistent and nonpersistent)	Until-revoked

You can use PowerShell to find the policies that will be affected by the retirement. Use the [PowerShell cmdlets](#) to see the all policies created in your organization, or to find which apps and service principals are linked to a specific policy.

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`.

Cmdlet reference

These are the cmdlets in the [Azure Active Directory PowerShell for Graph Preview module](#).

Manage policies

You can use the following cmdlets to manage policies.

CMDLET	DESCRIPTION
New-AzureADPolicy	Creates a new policy.
Get-AzureADPolicy	Gets all Azure AD policies or a specified policy.
Get-AzureADPolicyAppliedObject	Gets all apps and service principals that are linked to a policy.
Set-AzureADPolicy	Updates an existing policy.
Remove-AzureADPolicy	Deletes the specified policy.

Application policies

You can use the following cmdlets for application policies.

CMDLET	DESCRIPTION
Add-AzureADApplicationPolicy	Links the specified policy to an application.
Get-AzureADApplicationPolicy	Gets the policy that is assigned to an application.
Remove-AzureADApplicationPolicy	Removes a policy from an application.

Service principal policies

You can use the following cmdlets for service principal policies.

CMDLET	DESCRIPTION
Add-AzureADServicePrincipalPolicy	Links the specified policy to a service principal.
Get-AzureADServicePrincipalPolicy	Gets any policy linked to the specified service principal.
Remove-AzureADServicePrincipalPolicy	Removes the policy from the specified service principal.

Next steps

To learn more, read [examples of how to configure token lifetimes](#).

Application types for the Microsoft identity platform

4/12/2022 • 6 minutes to read • [Edit Online](#)

The Microsoft identity platform 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 in the [application scenarios](#).

The basics

You must register each app that uses the Microsoft identity platform in the Azure portal [App registrations](#). 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 the 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 written primarily in JavaScript, often with a framework like Angular, React, or Vue. The Microsoft identity platform supports these apps by using the [OpenID Connect](#) protocol for authentication and either [OAuth 2.0 implicit grant flow](#) or the more recent [OAuth 2.0 authorization code + PKCE flow](#) for authorization (see below).

The flow diagram below demonstrates the OAuth 2.0 authorization code grant (with details around PKCE omitted), where the app receives a code from the Microsoft identity platform `authorize` endpoint, and redeems it for an access token and a refresh token using cross-site web requests. The access token expires every 24 hours, and the app must request another code using the refresh token. In addition to the access token, an `id_token` that represents the signed-in user to the client application is typically also requested through the same flow and/or a separate OpenID Connect request (not shown here).



To see this scenario in action, check out the [Tutorial: Sign in users and call the Microsoft Graph API from a JavaScript SPA using auth code flow](#).

Authorization code flow vs. implicit flow

For most of the history of OAuth 2.0, the [implicit flow](#) was the recommended way to build single-page apps. With the removal of [third-party cookies](#) and [greater attention](#) paid to security concerns around the implicit flow, we've moved to the authorization code flow for single-page apps.

To ensure compatibility of your app in Safari and other privacy-conscious browsers, we no longer recommend use of the implicit flow and instead recommend the authorization code flow.

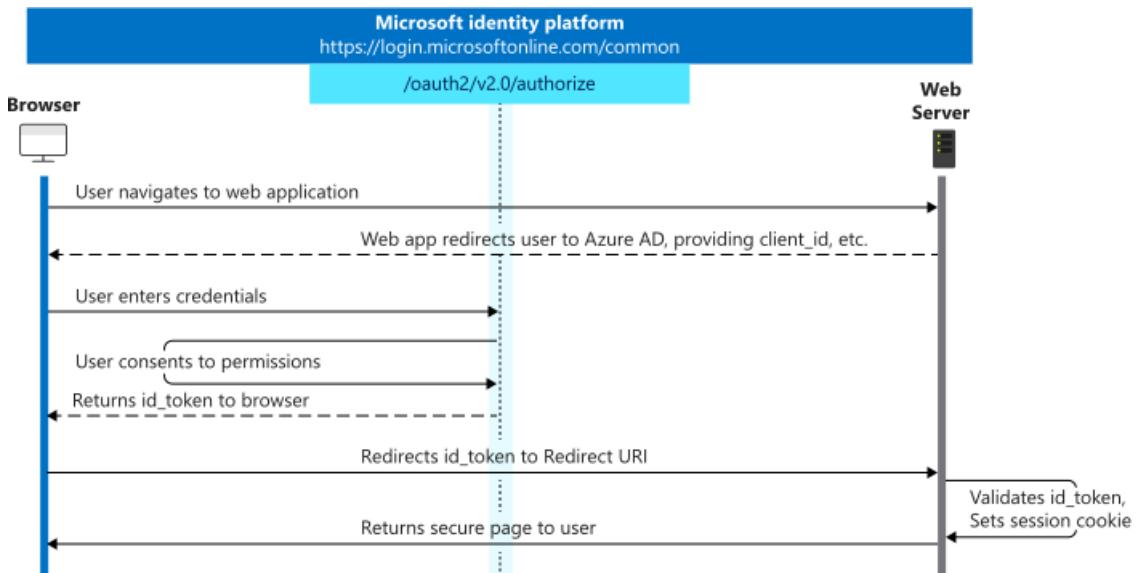
Web apps

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-dffc-4e3f-a6eb-62fe4bd48a58"
  ...
}
```

Further details of different types of tokens used in the Microsoft identity platform 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. A session cookie is set, which can be used to identify the user on subsequent page requests.

To see this scenario in action, try the code samples in the [Web app that signs in users scenario](#).

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 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:

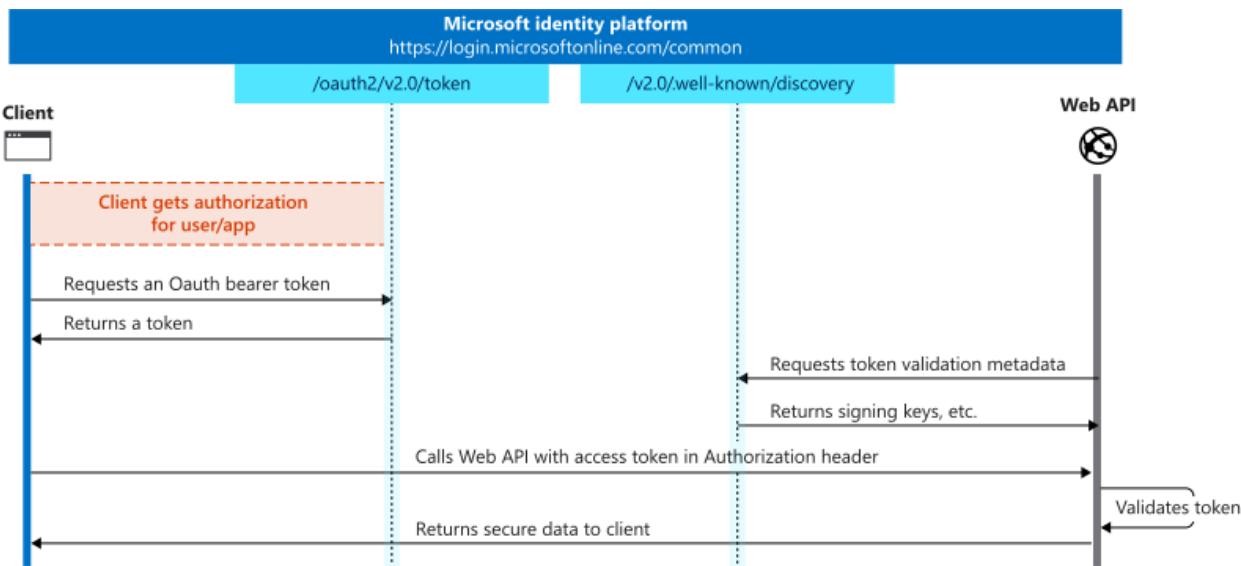
```

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 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 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:



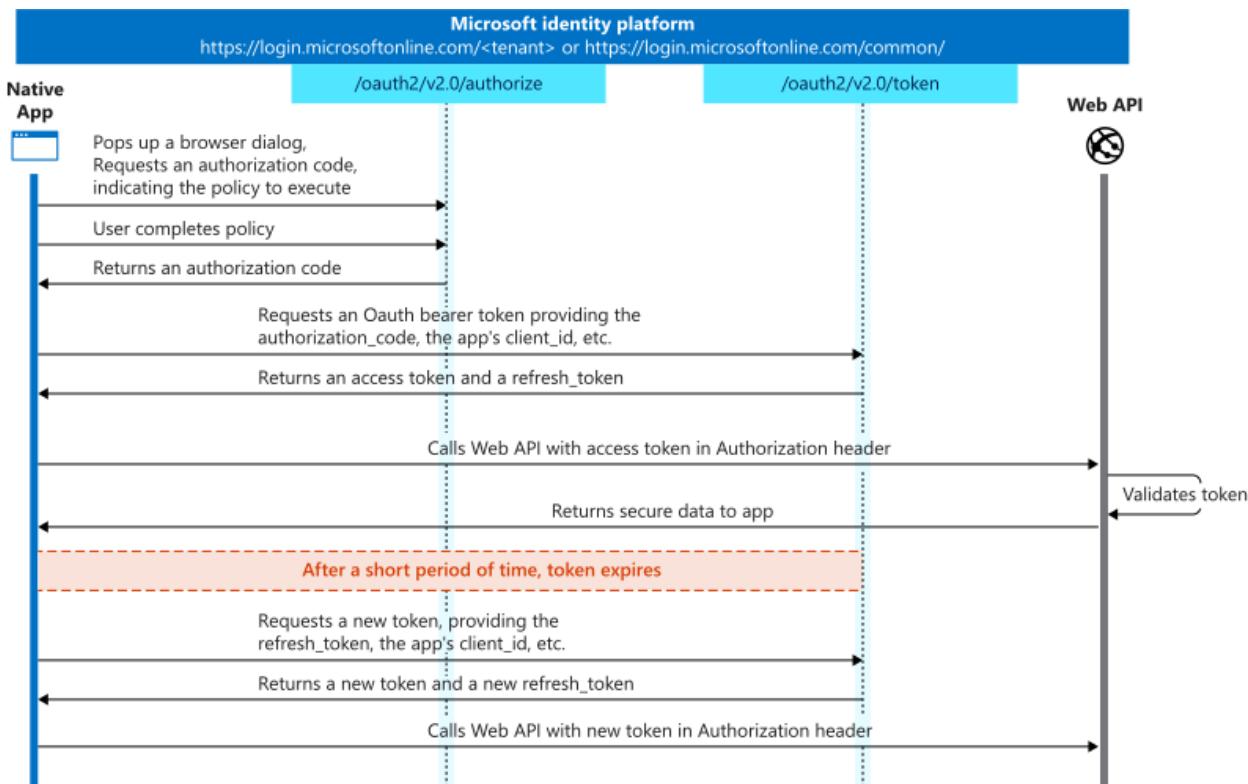
To learn how to secure a web API by using OAuth2 access tokens, check out the web API code samples in the [protected web API scenario](#).

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 the [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 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.



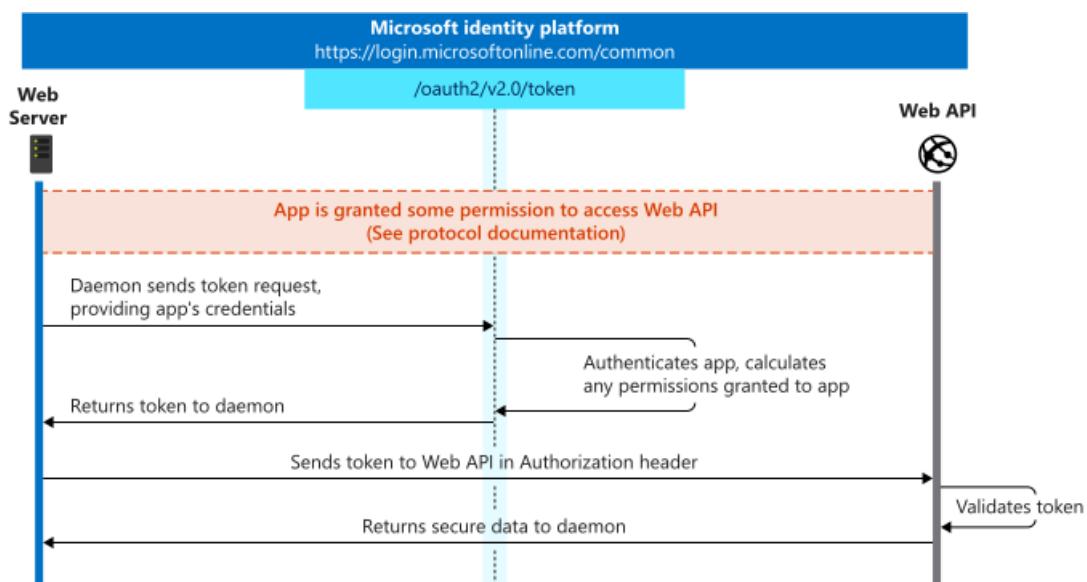
NOTE

If the application uses the default system webview, check the information about "Confirm My Sign-In" functionality and error code AADSTS50199 in [Azure AD authentication and authorization error codes](#).

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 [.NET Core daemon console application using Microsoft identity platform](#).

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](#).

Next steps

Now that you're familiar with the types of applications supported by the Microsoft identity platform, learn more about [OAuth 2.0 and OpenID Connect](#) to gain an understanding of the protocol components used by the different scenarios.

Microsoft identity platform application authentication certificate credentials

4/12/2022 • 4 minutes to read • [Edit Online](#)

The Microsoft identity platform allows an application to use its own credentials for authentication anywhere a client secret could be used, for example, in the OAuth 2.0 [client credentials grant](#) flow and the [on-behalf-of](#) (OBO) flow.

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. This is described in the [OpenID Connect](#) specification for the `private_key_jwt` client authentication option.

If you're interested in using a JWT issued by another identity provider as a credential for your application, please see [workload identity federation](#) for how to set up a federation policy.

Assertion format

To compute the assertion, you can use one of the many JWT libraries in the language of your choice - [MSAL supports this using `.WithCertificate\(\)`](#). The information is carried by the token in its Header, Claims, and Signature.

PARAMETER	REMARK
<code>alg</code>	Should be RS256
<code>typ</code>	Should be JWT
<code>x5t</code>	Base64url-encoded SHA-1 thumbprint of the X.509 certificate thumbprint. For example, given an X.509 certificate hash of <code>84E05C1D98BCE3A5421D225B140B36E86A3D5534</code> (Hex), the <code>x5t</code> claim would be <code>h0BchZi846VCHSJbFAs26Go9VTQ=</code> (Base64url).

Claims (payload)

CLAIM TYPE	VALUE	DESCRIPTION
<code>aud</code>	<code>https://login.microsoftonline.com/{tenant}/v2.0/</code>	The <code>aud</code> claim identifies the recipients that the JWT is intended for (here Azure AD) See RFC 7519, Section 4.1.3 . In this case, that recipient is the login server (<code>login.microsoftonline.com</code>).

CLAIM TYPE	VALUE	DESCRIPTION
<code>exp</code>	1601519414	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 . This allows the assertion to be used until then, so keep it short - 5-10 minutes after <code>nbf</code> at most. Azure AD does not place restrictions on the <code>exp</code> time currently.
<code>iss</code>	{ClientID}	The "iss" (issuer) claim identifies the principal that issued the JWT, in this case your client application. Use the GUID application ID.
<code>jti</code>	(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" value is a case-sensitive string. RFC 7519, Section 4.1.7
<code>nbf</code>	1601519114	The "nbf" (not before) claim identifies the time before which the JWT MUST NOT be accepted for processing. RFC 7519, Section 4.1.5 . Using the current time is appropriate.
<code>sub</code>	{ClientID}	The "sub" (subject) claim identifies the subject of the JWT, in this case also your application. Use the same value as <code>iss</code> .
<code>iat</code>	1601519114	The "iat" (issued at) claim identifies the time at which the JWT was issued. This claim can be used to determine the age of the JWT. RFC 7519, Section 4.1.5 .

Signature

The signature is computed by applying the certificate as described in the [JSON Web Token RFC7519 specification](#).

Example of a decoded JWT assertion

```
{
  "alg": "RS256",
  "typ": "JWT",
  "x5t": "gx8tGsyjcRqKjFPnd7RFwwZI0"
}
.

{
  "aud": "https://login.microsoftonline.com/contoso.onmicrosoft.com/oauth2/v2.0/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 *claims* (payload)
- The last section is the *signature* computed with the certificates from the content of the first two sections

```
"eyJhbGciOiJSUzI1NiIsIng1dCI6Imd40HRHeXN5amNScUtqR1BuZDdSRnd2d1pJMCJ9.eyJhdWQiOiJodHRwczpcL1wvbG9naW4ubWlcm9zb2Z0b25saW51LmNvbVwvam1wcmlldXJob3RtYWlsLm9ubWljcm9zb2Z0LmNvbVwvb2F1dGgyXC90b2tlbiIsImV4cCI6MTQ4NDU5MzM0MSwiZXNzIjoiOTd1MGE1YjctZDc0NS00MGI2LTk0ZmUtNWY3N2QzNWM2ZTA1IiwianRpIjoiMjJiM2JiMjYtZTA0Ni00MmRmLT1jOTYtNjVkJ3MmMxYzgxIiwibmJmIjoxNDg0NTkyNzQxLCJzdWIiOii5N2UwYTViNy1kNzQ1LTQwYjYtOTRmZS01Zjc3ZDM1YzzMDUiFQ.
Gh95kHCOEGq5E_ArMBbDXhwKR577scxYaoJ1P{a lot of characters here}KKJDEg"
```

Register your certificate with Microsoft identity platform

You can associate the certificate credential with the client application in the Microsoft identity platform 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** > **Certificates**.
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

After acquiring a certificate, compute these values:

- `$base64Thumbprint` - Base64-encoded value of the certificate hash
- `$base64Value` - Base64-encoded value of the certificate raw data

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"
  }
]

```

- Save the edits to the application manifest and then upload the manifest to Microsoft identity platform.

The `keyCredentials` property is multi-valued, so you may upload multiple certificates for richer key management.

Using a client assertion

Client assertions can be used anywhere a client secret would be used. So for example, in the [authorization code flow](#), you can pass in a `client_secret` to prove that the request is coming from your app. You can replace this with `client_assertion` and `client_assertion_type` parameters.

PARAMETER	VALUE	DESCRIPTION
<code>client_assertion_type</code>	<code>urn:ietf:params:oauth:client-assertion-type:jwt-bearer</code>	This is a fixed value, indicating that you are using a certificate credential.
<code>client_assertion</code>	JWT	This is the JWT created above.

Next steps

The [MSAL.NET library handles this scenario](#) in a single line of code.

The [.NET Core daemon console application using Microsoft identity platform](#) code sample on GitHub 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 cmdlet. You can also use the [app creation scripts](#) in the sample repo to create certificates, compute the thumbprint, and so on.

Signing key rollover in the Microsoft identity platform

4/12/2022 • 15 minutes to read • [Edit Online](#)

This article discusses what you need to know about the public keys that are used by the Microsoft identity platform 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 the Microsoft identity platform should be able to programmatically handle the key 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 the Microsoft identity platform

The Microsoft identity platform 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: The Microsoft identity platform uses a signing key that consists of a public and private key pair. When a user signs in to an application that uses the Microsoft identity platform for authentication, the Microsoft identity platform creates a security token that contains information about the user. This token is signed by the Microsoft identity platform using its private key before it is sent back to the application. To verify that the token is valid and originated from Microsoft identity platform, the application must validate the token's signature using the public keys exposed by the Microsoft identity platform that is contained in the tenant's [OpenID Connect discovery document](#) or [SAML/WS-Fed federation metadata document](#).

For security purposes, the Microsoft identity platform's signing key rolls on a periodic basis and, in the case of an emergency, could be rolled over immediately. There is no set or guaranteed time between these key rolls - any application that integrates with the Microsoft identity platform should be prepared to handle a key rollover event no matter how frequently it may occur. If your application doesn't handle sudden refreshes, and attempts to use an expired key to verify the signature on a token, your application will incorrectly reject the token. Checking every 24 hours for updates is a best practice, with throttled (once every five minutes at most) immediate refreshes of the key document if a token is encountered that doesn't validate with the keys in your application's cache.

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 and all of the keys specified in the document, since one key may be rolled soon, another may be its replacement, and so forth. The number of keys present can change over time based on the internal architecture of the Microsoft identity platform as we support new platforms, new clouds, or new authentication protocols. Neither the order of the keys in the JSON response nor the order in which they were exposed should be considered meaningful to your app.

Applications that support only a single signing key, or those that require manual updates to the signing keys, are inherently less secure and less reliable. They should be updated to use [standard libraries](#) to ensure that they are always using up-to-date signing keys, among other best practices.

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 / 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 regard 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 (for example, 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 (such as 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) to request tokens 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 files.

```
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 Microsoft identity platform, 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;
}
}
```

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 (Microsoft identity platform) 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 Microsoft identity platform](#) article, or downloading and inspecting the following code sample: [Multi-Tenant Cloud Application for Azure Active Directory](#).

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 the Microsoft identity platform 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 using the following PowerShell scripts.

To check and update signing keys with PowerShell, you'll need the [MSIdentityTools](#) PowerShell Module.

1. Install the [MSIdentityTools](#) PowerShell Module:

```
Install-Module -Name MSIdentityTools
```

2. Sign in by using the **Connect-MgGraph** command with an admin account to consent to the required scopes:

```
Connect-MgGraph -Scope "Application.ReadWrite.All"
```

3. Get the list of available signing key thumbprints:

```
Get-MsIdSigningKeyThumbprint
```

- Pick any of the key thumbprints and configure Azure Active Directory to use that key with your application (get the app ID from the [Azure portal](#)):

```
Update-MsIdApplicationSigningKeyThumbprint -ApplicationId <ApplicationId> -KeyThumbprint <Thumbprint>
```

- Test the web application by signing in to get a new token. The key update change is instantaneous, but make sure you use a new browser session (using, for example, Internet Explorer's "InPrivate," Chrome's "Incognito," or Firefox's "Private" mode) to ensure you are issued a new token.
- For each of the returned signing key thumbprints, run the `Update-MsIdApplicationSigningKeyThumbprint` cmdlet and test your web application sign-in process.
- If the web application signs you in properly, it supports automatic rollover. If it doesn't, modify your application to support manual rollover. Check out [Establishing a manual rollover process](#) for more information.
- Run the following script to revert to normal behavior:

```
Update-MsIdApplicationSigningKeyThumbprint -ApplicationId <ApplicationId> -Default
```

How to perform a manual rollover if your application does not support automatic rollover

If your application doesn't support automatic rollover, you need to establish a process that periodically monitors Microsoft identity platform's signing keys and performs a manual rollover accordingly.

To check and update signing keys with PowerShell, you'll need the [MSIdentityTools](#) PowerShell Module.

- Install the [MSIdentityTools](#) PowerShell Module:

```
Install-Module -Name MSIdentityTools
```

- Get the latest signing key (get the tenant ID from the [Azure portal](#)):

```
Get-MsIdSigningKeyThumbprint -Tenant <tenantId> -Latest
```

- Compare this key against the key your application is currently hardcoded or configured to use.
- If the latest key is different from the key your application is using, download the latest signing key:

```
Get-MsIdSigningKeyThumbprint -Latest -DownloadPath <DownloadFolderPath>
```

- Update your application's code or configuration to use the new key.
- Configure Azure Active Directory to use that latest key with your application (get the app ID from the [portal](#)):

```
Get-MsIdSigningKeyThumbprint -Latest | Update-MsIdApplicationSigningKeyThumbprint -ApplicationId <ApplicationId>
```

- Test the web application by signing in to get a new token. The key update change is instantaneous, but make sure you use a new browser session (using, for example, Internet Explorer's "InPrivate," Chrome's

"Incognito," or Firefox's "Private" mode) to ensure you are issued a new token.

8. If you experience any issues, revert to the previous key you were using and contact Azure support:

```
Update-MsIdApplicationSigningKeyThumbprint -ApplicationId <ApplicationId> -KeyThumbprint  
<PreviousKeyThumbprint>
```

9. After you update your application to support manual rollover, revert to normal behavior:

```
Update-MsIdApplicationSigningKeyThumbprint -ApplicationId <ApplicationId> -Default
```

Microsoft identity platform UserInfo endpoint

4/12/2022 • 3 minutes to read • [Edit Online](#)

The UserInfo endpoint is part of the [OpenID Connect standard](#) (OIDC), designed to return claims about the user that authenticated. For the Microsoft identity platform, the UserInfo endpoint is hosted on Microsoft Graph (<https://graph.microsoft.com/oidc/userinfo>).

Find the .well-known configuration endpoint

You can programmatically discover the UserInfo endpoint using the OpenID Connect discovery document, at <https://login.microsoftonline.com/common/v2.0/.well-known/openid-configuration>. It's listed in the `userinfo_endpoint` field, and this pattern can be used across clouds to help point to the right endpoint. We do not recommend hard-coding the UserInfo endpoint in your app – use the OIDC discovery document to find this endpoint at runtime instead.

As part of the OpenID Connect specification, the UserInfo endpoint is often automatically called by [OIDC compliant libraries](#) to get information about the user. Without hosting such an endpoint, the Microsoft identity platform would not be standards compliant and some libraries would fail. From the [list of claims identified in the OIDC standard](#) we produce the name claim, subject claim, and email when available and consented for.

Consider: Use an ID Token instead

The information available in the ID token that your app can receive is a superset of the information it can get from the UserInfo endpoint. Because you can get an ID token at the same time you get a token to call the UserInfo endpoint, we suggest that you use that ID token to get information about the user instead of calling the UserInfo endpoint. Using the ID token will eliminate one to two network requests from your application launch, reducing latency in your application.

If you require more details about the user, you should call the [Microsoft Graph /user API](#) to get information like office number or job title. You can also use [optional claims](#) to include additional user information in your ID and access tokens.

Calling the UserInfo endpoint

UserInfo is a standard OAuth Bearer token API, called like any other Microsoft Graph API using the access token received when getting a token for Microsoft Graph. It returns a JSON response containing claims about the user.

Permissions

Use the following [OIDC permissions](#) to call the UserInfo API. `openid` is required, and the `profile` and `email` scopes ensure that additional information is provided in the response.

PERMISSION TYPE	PERMISSIONS
Delegated (work or school account)	openid (required), profile, email
Delegated (personal Microsoft account)	openid (required), profile, email
Application	Not applicable

TIP

Copy this URL in your browser to get a token for the UserInfo endpoint as well as an [ID token](#) and replace the client ID and redirect URI with your own. Note that it only requests scopes for OpenID or Graph scopes, and nothing else. This is required, since you cannot request permissions for two different resources in the same token request.

```
https://login.microsoftonline.com/common/oauth2/v2.0/authorize?client_id=<yourClientID>&response_type=token+id_token&redirect_uri=<YourRedirectUri>&scope=user.read+openid+profile+email&response_mode=fragment&state=12345&nonce=678910
```

You can use this access token in the next section.

As with any other Microsoft Graph token, the token you receive here may not be a JWT. If you signed in a Microsoft account user, it will be an encrypted token format. This is because Microsoft Graph has a special token issuance pattern. This does not impact your ability to use the access token to call the UserInfo endpoint.

Calling the API

The UserInfo API supports both GET and POST, per the OIDC spec.

```
GET or POST /oidc/userinfo HTTP/1.1  
Host: graph.microsoft.com  
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJub25jZSI6I...  
Content-Type: application/json
```

UserInfo response

```
{  
  "sub": "OLu859SGc2Sr9ZsqbkG-QbeLgJlb41KcdiPoLYNpSFA",  
  "name": "Mikah Ollenburg", // names all require the "profile" scope.  
  "family_name": " Ollenburg",  
  "given_name": "Mikah",  
  "picture": "https://graph.microsoft.com/v1.0/me/photo/$value",  
  "email": "mikoll@contoso.com" //requires the "email" scope.  
}
```

The claims listed here are all of the claims that the UserInfo endpoint can return. These are the same values that the app would see in the [ID token](#) issued to the app.

Notes and caveats on the UserInfo endpoint

- If you want to call this UserInfo endpoint you must use the v2.0 endpoint. If you use the v1.0 endpoint you will get a token for the v1.0 UserInfo endpoint, hosted on login.microsoftonline.com. We recommend that all OIDC compliant apps and libraries use the v2.0 endpoint to ensure compatibility.
- The response from the UserInfo endpoint cannot be customized. If you'd like to customize claims, please use [claims mapping](#) to edit the information returned in the tokens.
- The response from the UserInfo endpoint cannot be added to. If you'd like to get additional claims about the user, please use [optional claims](#) to add new claims to the tokens.

Next Steps

- [Review the contents of ID tokens](#)
- [Customize the contents of an ID token using optional claims](#)
- [Request an access token and ID token using the OAuth2 protocol](#)

How the Microsoft identity platform uses the SAML protocol

4/12/2022 • 2 minutes to read • [Edit Online](#)

The Microsoft identity platform uses the SAML 2.0 and other protocols to enable applications to provide a single sign-on (SSO) experience to their users. The [SSO](#) and [Single Sign-Out](#) SAML profiles of Azure Active Directory (Azure AD) explain how SAML assertions, protocols, and bindings are used in the identity provider service.

The SAML protocol requires the identity provider (Microsoft identity platform) 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.

The Microsoft identity platform uses the cloud service's **Metadata URI** to retrieve the signing key and the logout URI. In the [Azure portal](#), you can open the app in **Azure Active Directory -> App registrations**, and then in **Manage -> Authentication**, you can update the Logout URL. This way the Microsoft identity platform can send the response to the correct URL.

Azure AD exposes tenant-specific and common (tenant-independent) SSO and single sign-out endpoints. These URLs represent addressable locations--they're 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.

Next steps

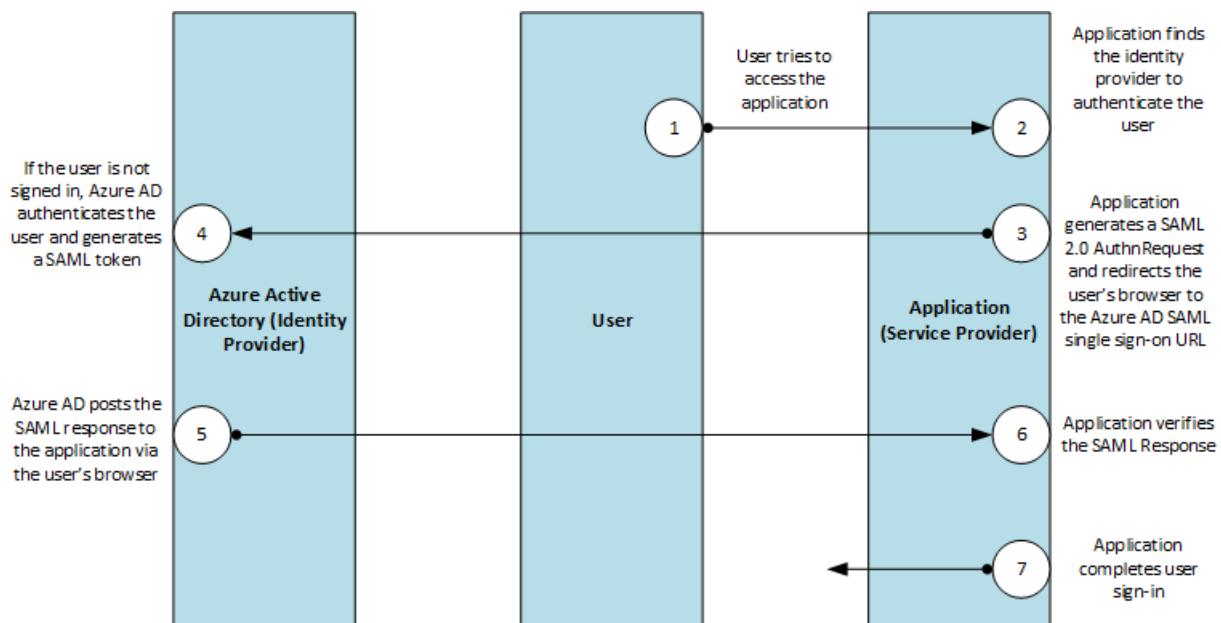
For information about the federation metadata documents that Azure AD publishes, see [Federation Metadata](#).

Single Sign-On SAML protocol

4/12/2022 • 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 (SSO).

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.



NOTE

This article discusses using SAML for single sign-on. For more information on other ways to handle single sign-on (for example, by using OpenID Connect or integrated Windows authentication), see [Single sign-on to applications in Azure Active Directory](#).

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	TYPE	DESCRIPTION
-----------	------	-------------

PARAMETER	TYPE	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>id6c1c178c166d486687be4aa5e482730</code> is a valid ID.
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.

If `SPNameQualifier` is specified, Azure AD will include the same `SPNameQualifier` in the response.

Azure AD ignores the `AllowCreate` attribute.

RequestedAuthnContext

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

A `Signature` element in `AuthnRequest` elements is optional. Azure AD does not validate signed authentication requests if a signature is present. Requestor verification is provided for by only responding to registered Assertion Consumer Service URLs.

Subject

Don't include a `Subject` element. Azure AD doesn't support specifying a subject for a request and will return an error if one is provided.

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://sts.windows.net/<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://sts.windows.net/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://sts.windows.net/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 URI 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-c2d0aee84d12">
    <AuthnContext>
        <AuthnContextClassRef> urn:oasis:names:tc:SAML:2.0:ac:classes:Password </AuthnContextClassRef>
    </AuthnContext>
</AuthnStatement>
```

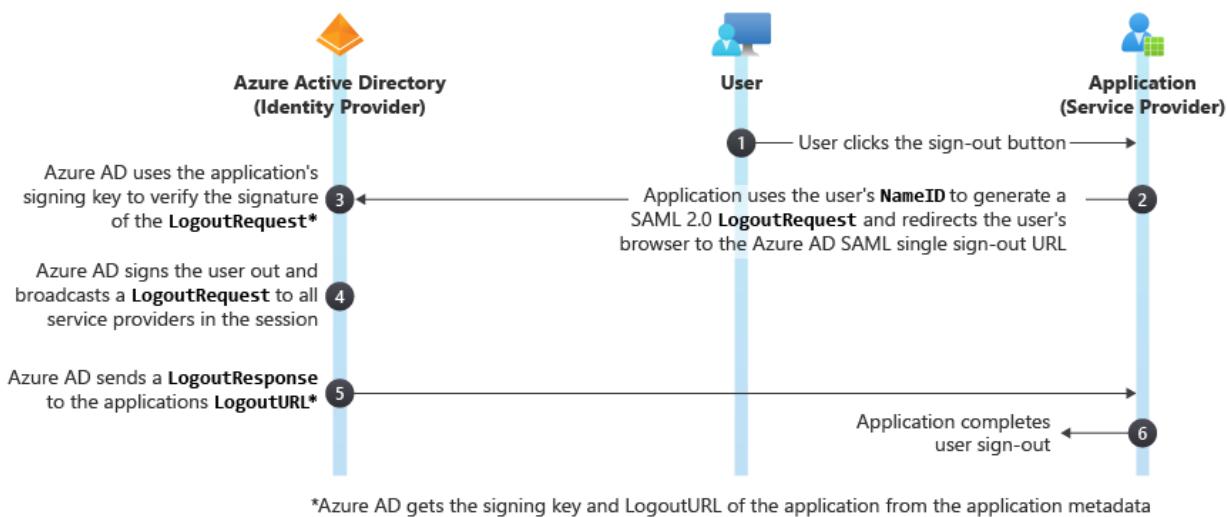
Single Sign-Out SAML Protocol

4/12/2022 • 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. If the app is [added to the Azure App Gallery](#) then this value can be set by default. Otherwise, the value must be determined and set by the person adding the app to their Azure AD tenant. Azure AD uses the **LogoutURL** to redirect users after they're signed out.

Azure AD supports redirect binding (HTTP GET), and not HTTP POST binding.

The following diagram shows the workflow of the Azure AD single sign-out process.



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="2013-03-28T07:10:49.6004822Z" 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>/](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.

SAML token claims reference

4/12/2022 • 5 minutes to read • [Edit Online](#)

The Microsoft identity platform emits several types of security tokens in the processing of each authentication flow. This document describes the format, security characteristics, and contents of SAML 2.0 tokens.

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/claims/authenticationMethod </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/claims/givenName"> <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 Microsoft 365 Distribution Lists.	<pre><Attribute Name="http://schemas.microsoft.com/ws/2008/06/identity/claims/groupMembership"> <AttributeValue>07dd8a60-bf6d-4e17-8844-230b77145381</AttributeValue></pre>
Groups Overage Indicator	<code>groups:src1</code>	For token requests that are not length-limited 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}/users/{userID}</AttributeValue></pre>

NAME	EQUIVALENT JWT CLAIM	DESCRIPTION	EXAMPLE
Identity Provider	idp	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.	<Attribute Name="http://schemas.microsoft.com/identity/claims/identityprovider"> <AttributeValue>https://sts.windows.net/cbb1a5ac-f33b-45fa-9bf5-f37db0fed422</AttributeValue>
IssuedAt	iat	Stores the time at which the token was issued. It is often used to measure token freshness.	<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">
Issuer	iss	Identifies the security token service (STS) that constructs and returns the token. In the tokens that Azure AD returns, the issuer is sts.windows.net. 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.	<Issuer>https://sts.windows.net/cbb1a5ac-f33b-45fa-9bf5-f37db0fed422</Issuer>
Last Name	family_name	Provides the last name, surname, or family name of the user as defined in the Azure AD user object.	<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname"> <AttributeValue>Miller</AttributeValue>
Name	unique_name	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.	<Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"> <AttributeValue>frankm@contoso.com</AttributeValue>
Object ID	oid	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/objectidentifier"> <AttributeValue>528b2ac2-aa9c-45e1-88d4-959b53bc7dd0</AttributeValue>
Roles	roles	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/role">
Subject	sub	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 recommended using this value in a general purpose authorization system. SubjectConfirmation is not a claim. It describes how the subject of the token is verified. Bearer indicates that the subject is confirmed by their possession of the token.	<Subject> <NameID>540rb3XjhFTv6EQTEtkEzcgVmToHkRkZUiSJlmLdVc</NameID> <SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer" /> </Subject>

Name	Equivalent JWT Claim	Description	Example
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.	<pre><Attribute Name="http://schemas.microsoft.com/identity/claims/tenantid"> <AttributeValue>cbb1a5ac-f33b-45fa-9b5f-f37db0fed422</AttributeValue></pre>
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.	<pre><Conditions> NotBefore="2013-03-18T21:32:51.261Z" NotOnOrAfter="2013-03-18T22:32:51.261Z" ></pre>

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/94411234-09af-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:Transforms>
            <ds:DigestMethod Algorithm="https://www.w3.org/2001/04/xmlenc#sha256" />
            <ds:DigestValue>cV1J580UjpD24hEyGuAxrbtgR0VyghCqI32UkER/nDY=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <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>cV1J580UjpD24hEyGuAxrbtgR0VyghCqI32UkER/nDY=</ds:DigestValue>
      </ds:Signature>
      <ds:SignatureValue>j+zPf6mti8Rq4Kyw2NU2nnu0pbJU1z5B/r/zDaKa07FCTdmJuZavIVFF8pspVR6CbzcYM3HOAmLhuWmBAAk6qQUbmKsw+XlmRyb2wd2luZG93cy5uZXQwhcNMTQwMTAxMDcwMDAwWlhCNMTYwMTaxDcwMDAwJwAtMSswKQYDvQ0DeJyH2Nvdw50cy5hY2Nlc3Njb250cm9sLndpbmVd3UhmVMIIBiTjANBgkqhkiG9w0BAQEFAAOCAQ8AMII8CgkCAQEAKSCWg6q9iYxvJ5E2NIhSyOikvqohC02GfipgH0sTSA5Fa1HlQosk9NTztX0yvS/AhsBePQyPgfyVJ6/EgzVuwRk5tr9e3n1uM94fLyg/AXBw09yAdf4dCHTPBCNR1dnR+Qn7/4PY1lwEuuHH0N0w/b1bfdfjhY+C/YM2E3pRxphbB3x//fcueV7ddz2LY1h3wzj00S/7kjP1NCsxCNyQEQOTkbHFHi3mu0u135QwNdhdynd/GTgW8A+6SN1r4hpzjFKFLBntB77ACSiYx+IHK4Mp+NaVeis5wQtSjQTI+XsokxrDqYlwus11SihgbV/STt5SenufuWIDAQABo2iWYDBeegNhQEEvZBvgBDLebM6b38jWdq1Br-BNFEnOs8wLTEmcKGA1UEAxMYLNnjB3udHMuWNjZKXNzY29u9dHjb53akB5kbzdLm51dIIQsR1M0jhefZhKk49YD0SK1TAJBgUrDgMChQUAA4IBAQJC4JApriF77EKC4zF5buBLaQHQ1PNTa1UMDbdNVGKCrSpM65b8h0Nw1ijGGgy/kN8P6jWFdm51lZ0YPT0gzcRZgxDPjtajv1VE02K2ICVTV1rqQr0hEzMSSzsTKxFvNfwN6ADDKn3bv0Vbptp+nBY5UqnI7xbcoHLZ4wD251uj5+lo13YLnsVrmQ16NCByq2nQFNPnJw6t3XUbwBHxp46aL1t/eGf/7Xx6i1y8pJX4DyprfTutDz882RwofGE05t4Cw+zZg70dJ/hH/ODYRMorfxEW+8uKmXMkmX2wyxMKvf1PbTy5LmAU8Jvjs2tLg4rOBcXWLAIarZ/<x509Certificate>
        <x509Data>
        </KeyInfo>
      </ds:Signature>
      <Subject>
        <NameID Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent">_3naDei2LNxUmEcWd0B2InI_jVET1pMLR6iQSvmo</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>
      </AttributeStatement>
    </Assertion>
  </t:RequestedSecurityToken>
</t:RequestSecurityTokenResponse>
```

```

        </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>87dd8a89-bf6d-4e81-8844-230b77145381</AttributeValue>
            <AttributeValue>e129f4d-6b0a-4944-982d-f776000632af</AttributeValue>
            <AttributeValue>3ee07328-52ef-4739-a89b-109708c22fb5</AttributeValue>
            <AttributeValue>329k14b3-1851-4b94-947f-9a4dabc595f4</AttributeValue>
            <AttributeValue>6e32c650-9b0a-4491-b429-6c60d2ca9a42</AttributeValue>
            <AttributeValue>f3a16947-9a58-4e8f-9d47-b70029v07424</AttributeValue>
            <AttributeValue>8e2c86b2-b1ad-476d-9574-544d155aa6ff</AttributeValue>
            <AttributeValue>1bf80264-ff24-4866-b22c-6212e5b9a847</AttributeValue>
            <AttributeValue>4075f9c3-072d-4c32-b542-03e6bc678f3e</AttributeValue>
            <AttributeValue>76f880527-f2cd-46f4-8c52-8jvd8bc749b1</AttributeValue>
            <AttributeValue>9ba31460-44d0-42b5-b90c-47b3fc48e35</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>
            </AuthnContext>
            <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>

```

Next steps

- To learn more about managing token lifetime policy using the Microsoft Graph API, see the [Azure AD policy resource overview](#).
- 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](#)

Microsoft identity platform token exchange scenarios with SAML and OIDC/OAuth

4/12/2022 • 2 minutes to read • [Edit Online](#)

SAML and OpenID Connect (OIDC) / OAuth are popular protocols used to implement Single Sign-On (SSO). Some apps might only implement SAML and others might only implement OIDC/OAuth. Both protocols use tokens to communicate secrets. To learn more about SAML, see [Single Sign-On SAML protocol](#). To learn more about OIDC/OAuth, see [OAuth 2.0 and OpenID Connect protocols on Microsoft identity platform](#).

This article outlines a common scenario where an app implements SAML but calls the Graph API, which uses OIDC/OAuth. Basic guidance is provided for people working with this scenario.

Scenario: You have a SAML token and want to call the Graph API

Many apps are implemented with SAML. However, the Graph API uses the OIDC/OAuth protocols. It's possible, though not trivial, to add OIDC/OAuth functionality to a SAML app. Once OAuth functionality is available in an app, the Graph API can be used.

The general strategy is to add the OIDC/OAuth stack to your app. With your app that implements both standards you can use a session cookie. You aren't exchanging a token explicitly. You're logging a user in with SAML, which generates a session cookie. When the Graph API invokes an OAuth flow, you use the session cookie to authenticate. This strategy assumes the Conditional Access checks pass and the user is authorized.

NOTE

The recommended library for adding OIDC/OAuth behavior is the Microsoft Authentication Library (MSAL). To learn more about MSAL, see [Overview of the Microsoft Authentication Library \(MSAL\)](#). The previous library was called Active Directory Authentication Library (ADAL), however it is not recommended as MSAL is replacing it.

Next steps

- [Authentication flows and application scenarios](#)

Azure Active Directory app manifest

4/12/2022 • 13 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 [Microsoft Graph API](#) or [Microsoft Graph PowerShell SDK](#). 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.
- 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. Go to the [Azure portal](#). Search for and select the **Azure Active Directory** service.
2. 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

This section describes the attributes found in the application manifest.

id attribute

KEY	VALUE TYPE
id	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.

Example:

```
"id": "f7f9acfc-ae0c-4d6c-b489-0a81dc1652dd",
```

accessTokenAcceptedVersion attribute

KEY	VALUE TYPE
accessTokenAcceptedVersion	Nullable Int32

Specifies the access token version expected by the resource. This parameter changes the version and format of

the JWT produced independent of the endpoint or client used to request the access token.

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 `accesstokenAcceptedVersion` to indicate the supported access token format.

Possible values for `accesstokenAcceptedVersion` are 1, 2, or null. If the value is null, this parameter defaults to 1, which corresponds to the v1.0 endpoint.

If `signInAudience` is `AzureADandPersonalMicrosoftAccount`, the value must be `2`.

Example:

```
"accessTokenAcceptedVersion": 2,
```

addIns attribute

KEY	VALUE TYPE
<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 parameter will let services like Microsoft 365 call the application in the context of a document the user is working on.

Example:

```
"addIns": [
  {
    "id": "968A844F-7A47-430C-9163-07AE7C31D407",
    "type": "FileHandler",
    "properties": [
      {
        "key": "version",
        "value": "2"
      }
    ]
  }
],
```

allowPublicClient attribute

KEY	VALUE TYPE
<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 can't determine the client app type. For example, one such scenario is the [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.

Example:

```
"allowPublicClient": false,
```

appId attribute

KEY	VALUE TYPE
appId	String

Specifies the unique identifier for the app that is assigned to an app by Azure AD.

Example:

```
"appId": "601790de-b632-4f57-9523-ee7cb6ceba95",
```

appRoles attribute

KEY	VALUE TYPE
appRoles	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](#).

Example:

```
"appRoles": [
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "Read-only access to device information",
    "displayName": "Read Only",
    "id": "601790de-b632-4f57-9523-ee7cb6ceba95",
    "isEnabled": true,
    "value": "ReadOnly"
  }
],
```

errorUrl attribute

KEY	VALUE TYPE
errorUrl	String

Unsupported.

groupMembershipClaims attribute

KEY	VALUE TYPE
groupMembershipClaims	String

Configures the `groups` 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:

- `"None"`
- `"SecurityGroup"` (for security groups and Azure AD roles)
- `"ApplicationGroup"` (this option includes only groups that are assigned to the application)
- `"DirectoryRole"` (gets the Azure AD directory roles the user is a member of)

- "All" (this will get all of the security groups, distribution groups, and Azure AD directory roles that the signed-in user is a member of).

Example:

```
"groupMembershipClaims": "SecurityGroup",
```

optionalClaims attribute

KEY	VALUE TYPE
optionalClaims	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](#).

Example:

```
"optionalClaims": null,
```

identifierUris attribute

KEY	VALUE TYPE
identifierUris	String Array

User-defined URI(s) that uniquely identify a web app within its Azure AD tenant or verified customer owned domain. When an application is used as a resource app, the identifierUri value is used to uniquely identify and access the resource.

The following API and HTTP scheme-based application ID URI formats are supported. Replace the placeholder values as described in the list following the table.

SUPPORTED APPLICATION ID URI FORMATS	EXAMPLE APP ID URIS
api://<appId>	api://fc4d2d73-d05a-4a9b-85a8-4f2b3a5f38ed
api://<tenantId>/<appId>	api://a8573488-ff46-450a-b09a-6eca0c6a02dc/fc4d2d73-d05a-4a9b-85a8-4f2b3a5f38ed
api://<tenantId>/<string>	api://a8573488-ff46-450a-b09a-6eca0c6a02dc/api
api://<string>/<appId>	api://productapi/fc4d2d73-d05a-4a9b-85a8-4f2b3a5f38ed
https://<tenantInitialDomain>.onmicrosoft.com/<string>	https://contoso.onmicrosoft.com/productsapi
https://<verifiedCustomDomain>/<string>	https://contoso.com/productsapi
https://<string>.<verifiedCustomDomain>	https://product.contoso.com

SUPPORTED APPLICATION ID URI FORMATS	EXAMPLE APP ID URIS
<code>https://<string>.<verifiedCustomDomain>/<string></code>	<code>https://product.contoso.com/productsapi</code>

- `<appId>` - The application identifier (`appId`) property of the application object.
- `<string>` - The string value for the host or the api path segment.
- `<tenantId>` - A GUID generated by Azure to represent the tenant within Azure.
- `<tenantInitialDomain>` - `<tenantInitialDomain>.onmicrosoft.com`, where `<tenantInitialDomain>` is the initial domain name the tenant creator specified at tenant creation.
- `<verifiedCustomDomain>` - A [verified custom domain](#) configured for your Azure AD tenant.

NOTE

If you use the `api://` scheme, you add a string value directly after the `"api://"`. For example, `api://<string>`. That string value can be a GUID or an arbitrary string. If you add a GUID value, it must match either the app ID or the tenant ID. The application ID URI value must be unique for your tenant. If you add `api://<tenantId>` as the application ID URI, no one else will be able to use that URI in any other app. The recommendation is to use `api://<appId>`, instead, or the HTTP scheme.

Example:

```
"identifierUris": "https://contoso.onmicrosoft.com/fc4d2d73-d05a-4a9b-85a8-4f2b3a5f38ed",
```

informationalUrls attribute

KEY	VALUE TYPE
informationalUrls	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](#).

Example:

```
"informationalUrls": {
    "termsOfService": "https://MyRegisteredApp/termsofservice",
    "support": "https://MyRegisteredApp/support",
    "privacy": "https://MyRegisteredApp/privacystatement",
    "marketing": "https://MyRegisteredApp/marketing"
},
```

keyCredentials attribute

KEY	VALUE TYPE
keyCredentials	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 a resource).

Example:

```

"keyCredentials": [
  {
    "customKeyIdentifier":null,
    "endDate":"2018-09-13T00:00:00Z",
    "keyId":"><guid>,
    "startDate":"2017-09-12T00:00:00Z",
    "type":AsymmetricX509Cert",
    "usage":Verify",
    "value":null
  }
],

```

knownClientApplications attribute

KEY	VALUE TYPE
knownClientApplications	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. It 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.

Example:

```

"knownClientApplications": ["f7f9acf-ae0c-4d6c-b489-0a81dc1652dd"],

```

logoUrl attribute

KEY	VALUE TYPE
logoUrl	String

Read only value that points to the CDN URL to logo that was uploaded in the portal.

Example:

```

"logoUrl": "https://MyRegisteredAppLogo",

```

logoutUrl attribute

KEY	VALUE TYPE
logoutUrl	String

The URL to log out of the app.

Example:

```

"logoutUrl": "https://MyRegisteredAppLogout",

```

name attribute

KEY	VALUE TYPE
name	String

The display name for the app.

Example:

```
"name": "MyRegisteredApp",
```

oauth2AllowImplicitFlow attribute

KEY	VALUE TYPE
oauth2AllowImplicitFlow	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

[OAuth 2.0 implicit grant flow](#) in the table of contents and see the topics about implicit flow.

Example:

```
"oauth2AllowImplicitFlow": false,
```

oauth2AllowIdTokenImplicitFlow attribute

KEY	VALUE TYPE
oauth2AllowIdTokenImplicitFlow	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.

Example:

```
"oauth2AllowIdTokenImplicitFlow": false,
```

oauth2Permissions attribute

KEY	VALUE TYPE
oauth2Permissions	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.

Example:

```

"oauth2Permissions": [
  {
    "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"
  }
],

```

oauth2RequiredPostResponse attribute

KEY	VALUE TYPE
oauth2RequiredPostResponse	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.

Example:

```
"oauth2RequirePostResponse": false,
```

parentalControlSettings attribute

KEY	VALUE TYPE
parentalControlSettings	String

- `countriesBlockedForMinors` specifies the countries/regions in which the app is blocked for minors.
- `legalAgeGroupRule` specifies the legal age group rule that applies to users of the app. Can be set to `Allow`, `RequireConsentForPrivacyServices`, `RequireConsentForMinors`, `RequireConsentForKids`, or `BlockMinors`.

Example:

```

"parentalControlSettings": {
  "countriesBlockedForMinors": [],
  "legalAgeGroupRule": "Allow"
},

```

passwordCredentials attribute

KEY	VALUE TYPE
passwordCredentials	Collection

See the description for the `keyCredentials` property.

Example:

```

"passwordCredentials": [
  {
    "customKeyIdentifier": null,
    "endDate": "2018-10-19T17:59:59.6521653Z",
    "keyId": "<guid>",
    "startDate": "2016-10-19T17:59:59.6521653Z",
    "value":null
  }
],

```

preAuthorizedApplications attribute

KEY	VALUE TYPE
preAuthorizedApplications	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.

Example:

```

"preAuthorizedApplications": [
  {
    "appId": "abcdefg2-000a-1111-a0e5-812ed8dd72e8",
    "permissionIds": [
      "8748f7db-21fe-4c83-8ab5-53033933c8f1"
    ]
  }
],

```

publisherDomain attribute

KEY	VALUE TYPE
publisherDomain	String

The verified publisher domain for the application. Read-only.

Example:

```

"publisherDomain": "{tenant}.onmicrosoft.com",

```

replyUrlsWithType attribute

KEY	VALUE TYPE
replyUrlsWithType	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:

- Web
- InstalledClient
- Spa

To learn more, see [replyUrl restrictions and limitations](#).

Example:

```
"replyUrlsWithType": [
  {
    "url": "https://localhost:4400/services/office365/redirectTarget.html",
    "type": "InstalledClient"
  }
],
```

requiredResourceAccess attribute

KEY	VALUE TYPE
requiredResourceAccess	Collection

With dynamic consent, `requiredResourceAccess` drives the admin consent experience and the user consent experience for users who are using static consent. However, this parameter doesn't drive the user consent experience for the general case.

- `resourceAppId` 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.
- `resourceAccess` is an array that lists the OAuth2.0 permission scopes and app roles that the app requires from the specified resource. Contains the `id` and `type` values of the specified resources.

Example:

```
"requiredResourceAccess": [
  {
    "resourceAppId": "00000002-0000-0000-c000-000000000000",
    "resourceAccess": [
      {
        "id": "311a71cc-e848-46a1-bdf8-97ff7156d8e6",
        "type": "Scope"
      }
    ]
  }
],
```

samlMetadataUrl attribute

KEY	VALUE TYPE
samlMetadataUrl	String

The URL to the SAML metadata for the app.

Example:

```
"samlMetadataUrl": "https://MyRegisteredAppSAMLMetadata",
```

signInUrl attribute

KEY	VALUE TYPE
signInUrl	String

Specifies the URL to the app's home page.

Example:

```
"signInUrl": "https://MyRegisteredApp",
```

signInAudience attribute

KEY	VALUE TYPE
signInAudience	String

Specifies what Microsoft accounts are supported for the current application. Supported values are:

- `AzureADMyOrg` - Users with a Microsoft work or school account in my organization's Azure AD tenant (for example, single tenant)
- `AzureADMultipleOrgs` - Users with a Microsoft work or school account in any organization's Azure AD tenant (for example, multi-tenant)
- `AzureADandPersonalMicrosoftAccount` - Users with a personal Microsoft account, or a work or school account in any organization's Azure AD tenant
- `PersonalMicrosoftAccount` - Personal accounts that are used to sign in to services like Xbox and Skype.

Example:

```
"signInAudience": "AzureADandPersonalMicrosoftAccount",
```

tags attribute

KEY	VALUE TYPE
tags	String Array

Custom strings that can be used to categorize and identify the application.

Example:

```
"tags": [
    "ProductionApp"
],
```

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 previously specify 100 redirect URIs 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 xxxxx. 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 error 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 xxxxx application. Error detail: Invalid object identifier 'undefined'. []."
- "Failed to update xxxxx application. Error detail: One or more property values specified are invalid. []."
- "Failed to update xxxxx application. Error detail: Not allowed to set availableToOtherTenants in this api version for update. []."
- "Failed to update xxxxx application. Error detail: Updates to 'replyUrls' property isn't allowed for this application. Use 'replyUrlsWithType' property instead. []."
- "Failed to update xxxxx 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 that 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 actions:

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 core Microsoft identity platform developer concepts.

Use the following comments section to provide feedback that helps refine and shape our content.

Redirect URI (reply URL) restrictions and limitations

4/12/2022 • 5 minutes to read • [Edit Online](#)

A redirect URI, or reply URL, is the location where the authorization server sends the user once the app has been successfully authorized and granted an authorization code or access token. The authorization server sends the code or token to the redirect URI, so it's important you register the correct location as part of the app registration process.

The Azure Active Directory (Azure AD) application model specifies these restrictions to redirect URIs:

- Redirect URIs must begin with the scheme `https`. There are some [exceptions for localhost](#) redirect URIs.
- Redirect URIs are case-sensitive and must match the case of the URL path of your running application. For example, if your application includes as part of its path `.../abc/response-oidc`, do not specify `.../ABC/response-oidc` in the redirect URI. Because the web browser treats paths as case-sensitive, cookies associated with `.../abc/response-oidc` may be excluded if redirected to the case-mismatched `.../ABC/response-oidc` URL.
- Redirect URIs *not* configured with a path segment are returned with a trailing slash ('`/`') in the response. This applies only when the response mode is `query` or `fragment`.

Examples:

- `https://contoso.com` is returned as `https://contoso.com/`
- `http://localhost:7071` is returned as `http://localhost:7071/`

- Redirect URIs that contain a path segment are *not* appended with a trailing slash in the response.

Examples:

- `https://contoso.com/abc` is returned as `https://contoso.com/abc`
- `https://contoso.com/abc/response-oidc` is returned as `https://contoso.com/abc/response-oidc`

Maximum number of redirect URIs

This table shows the maximum number of redirect URIs you can add to an app registration in the Microsoft identity platform.

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 you add to an app registration.

Redirect URIs in application vs. service principal objects

- Always add redirect URIs to the application object only.
- Do not add redirect URI values to a service principal because these values could be removed when the service principal object syncs with the application object. This could happen due to any update operation which triggers a sync between the two objects.

Supported schemes

HTTPS: The HTTPS scheme (`https://`) is supported for all HTTP-based redirect URIs.

HTTP: The HTTP scheme (`http://`) is supported *only* for `localhost` URIs and should be used only during active local application development and testing.

EXAMPLE REDIRECT URI	VALIDITY
<code>https://contoso.com</code>	Valid
<code>https://contoso.com/abc/response-oidc</code>	Valid
<code>https://localhost</code>	Valid
<code>http://contoso.com/abc/response-oidc</code>	Invalid
<code>http://localhost</code>	Valid
<code>http://localhost/abc</code>	Valid

Localhost exceptions

Per [RFC 8252 sections 8.3 and 7.3](#), "loopback" or "localhost" redirect URIs come with two special considerations:

- `http` URI schemes are acceptable because the redirect never leaves the device. As such, both of these URIs are acceptable:
 - `http://localhost/myApp`
 - `https://localhost/myApp`
- Due to ephemeral port ranges often required by native applications, the port component (for example, `:5001` or `:443`) is ignored for the purposes of matching a redirect URI. As a result, all of these URIs are considered equivalent:
 - `http://localhost/MyApp`
 - `http://localhost:1234/MyApp`
 - `http://localhost:5000/MyApp`
 - `http://localhost:8080/MyApp`

From a development standpoint, this means a few things:

- Do not register multiple redirect URIs where only the port differs. The login server will pick one arbitrarily and use the behavior associated with that redirect URI (for example, whether it's a `web`-, `native`-, or `spa`-type redirect).

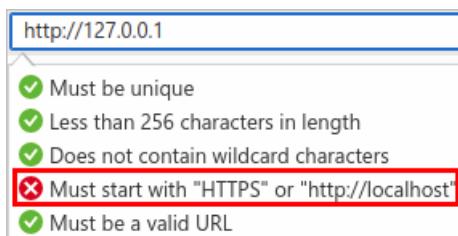
This is especially important when you want to use different authentication flows in the same application registration, for example both the authorization code grant and implicit flow. To associate the correct response behavior with each redirect URI, the login server must be able to distinguish between the redirect URIs and cannot do so when only the port differs.

- To register multiple redirect URIs on localhost to test different flows during development, differentiate them using the *path* component of the URI. For example, `http://localhost/MyWebApp` doesn't match `http://localhost/MyNativeApp`.
- The IPv6 loopback address (`[::1]`) is not currently supported.

Prefer 127.0.0.1 over localhost

To prevent your app from being broken by misconfigured firewalls or renamed network interfaces, use the IP literal loopback address `127.0.0.1` in your redirect URI instead of `localhost`. For example, `https://127.0.0.1`.

You cannot, however, use the **Redirect URIs** text box in the Azure portal to add a loopback-based redirect URI that uses the `http` scheme:



To add a redirect URI that uses the `http` scheme with the `127.0.0.1` loopback address, you must currently modify the [replyUrlsWithType attribute in the application manifest](#).

Restrictions on wildcards in redirect URIs

Wildcard URIs like `https://*.contoso.com` may seem convenient, but should be avoided due to 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.

Wildcard URIs are currently unsupported in app registrations configured to sign in personal Microsoft accounts and work or school accounts. Wildcard URIs are allowed, however, for apps that are configured to sign in only work or school accounts in an organization's Azure AD tenant.

To add redirect URIs with wildcards to app registrations that sign in work or school accounts, use the application manifest editor in **App registrations** in the Azure portal. Though it's possible to set a redirect URI with a wildcard by using the manifest editor, we *strongly* recommend you adhere to section 3.1.2 of RFC 6749 and use only absolute URIs.

If your scenario requires more redirect URIs than the maximum limit allowed, consider the following state parameter approach instead of adding a wildcard redirect URI.

Use a state parameter

If you have several subdomains and your scenario requires that, upon successful authentication, you redirect users to the same page from which they started, using a state parameter might be helpful.

In this approach:

- Create a "shared" redirect URI per application to process the security tokens you receive from the authorization endpoint.
- Your application can send application-specific parameters (such as subdomain 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](#).
- 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.

WARNING

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, like validating the domain name in the redirect URI against the token.

Next steps

Learn about the app registration [Application manifest](#).

Validation differences by supported account types (signInAudience)

4/12/2022 • 3 minutes to read • [Edit Online](#)

When registering an application with the Microsoft identity platform for developers, you're asked to select which account types your application supports. In the application object and manifest, this property is `signInAudience`.

The options include the following values:

- **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 AND PERSONALMICROSOFTACCOUNT
Application ID URI (<code>identifierURIs</code>)	Must be unique in the tenant urn:// schemes are supported Wildcards aren't 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 aren't supported Query strings and fragments are supported Maximum length of 255 characters No limit* on number of identifierURIs	Must be globally unique urn:// schemes aren't supported Wildcards, fragments, and query strings aren't 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 two client secrets
Redirect URIs (<code>replyURLs</code>)	See Redirect URI/reply URL restrictions and limitations for more info.		

PROPERTY	AZUREADMYORG	AZUREADMULTIPLEORGS	AZUREADANDPERSONALMICROSOFTACCOUNT AND PERSONALMICROSOFTACCOUNT
API permissions (<code>requiredResourceAccess</code>)	No more than 50 APIs (resource apps) from the same tenant as the application, no more than 10 APIs from other tenants, and no more than 400 permissions total across all APIs.	No more than 50 APIs (resource apps) from the same tenant as the application, no more than 10 APIs from other tenants, and no more than 400 permissions total across all APIs.	Maximum of 50 resources per application and 30 permissions per resource (for example, Microsoft Graph). Total limit of 200 per application (resources x permissions).
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
Front-channel logout URL	https://localhost is allowed <code>http</code> scheme isn't allowed Maximum length of 255 characters	https://localhost is allowed <code>http</code> scheme isn't allowed Maximum length of 255 characters	https://localhost is allowed, http://localhost fails <code>http</code> scheme isn't allowed Maximum length of 255 characters Wildcards aren't supported
Display name	Maximum length of 120 characters	Maximum length of 120 characters	Maximum length of 90 characters
Tags	Individual tag size must be between 1 and 256 characters (inclusive) No whitespaces or duplicate tags allowed No limit* on number of tags	Individual tag size must be between 1 and 256 characters (inclusive) No whitespaces or duplicate tags allowed No limit* on number of tags	Individual tag size must be between 1 and 256 characters (inclusive) No whitespaces or duplicate tags allowed No limit* on number of tags

* There's a global limit of about 1000 items across all the collection properties on the app object.

Next steps

For more information about application registrations and their JSON manifest, see:

- [Application registration](#)
- [Application manifest](#)

Microsoft identity platform authentication libraries

4/12/2022 • 4 minutes to read • [Edit Online](#)

The following tables show Microsoft Authentication Library support for several application types. They include links to library source code, where to get the package for your app's project, and whether the library supports user sign-in (authentication), access to protected web APIs (authorization), or both.

The Microsoft identity platform has been certified by the OpenID Foundation as a [certified OpenID provider](#). If you prefer to use a library other than the Microsoft Authentication Library (MSAL) or another Microsoft-supported library, choose one with a [certified OpenID Connect implementation](#).

If you choose to hand-code your own protocol-level implementation of [OAuth 2.0 or OpenID Connect 1.0](#), pay close attention to the security considerations in each standard's specification and follow a software development lifecycle (SDL) methodology like the [Microsoft SDL](#).

Single-page application (SPA)

A single-page application runs entirely in the browser and fetches page data (HTML, CSS, and JavaScript) dynamically or at application load time. It can call web APIs to interact with back-end data sources.

Because a SPA's code runs entirely in the browser, it's considered a *public client* that's unable to store secrets securely.

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
Angular	MSAL Angular v2²	msal-angular	Tutorial	✓	✓	GA
Angular	MSAL Angular³	msal-angular	—	✓	✓	GA
AngularJS	MSAL AngularJS³	msal-angularjs	—	✓	✓	Public preview
JavaScript	MSAL.js v2²	msal-browser	Tutorial	✓	✓	GA
JavaScript	MSAL.js 1.0³	msal-core	—	✓	✓	GA
React	MSAL React²	msal-react	Tutorial	✓	✓	GA

¹ [Supplemental terms of use for Microsoft Azure Previews](#) apply to libraries in *Public preview*.

² [Auth code flow with PCKE only \(Recommended\).](#)

³ [Implicit grant flow only.](#)

Web application

A web application runs code on a server that generates and sends HTML, CSS, and JavaScript to a user's web

browser to be rendered. The user's identity is maintained as a session between the user's browser (the front end) and the web server (the back end).

Because a web application's code runs on the web server, it's considered a *confidential client* that can store secrets securely.

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
.NET	MSAL.NET	Microsoft.Identity.Client	—	✗	✓	GA
.NET	Microsoft.IdentityModel	Microsoft.IdentityModel	—	✗ ²	✗ ²	GA
ASP.NET Core	ASP.NET Core	Microsoft.AspNetCore.Authentication	Quickstart	✓	✗	GA
ASP.NET Core	Microsoft.Identity.Web	Microsoft.Identity.Web	Quickstart	✓	✓	GA
Java	MSAL4J	msal4j	Quickstart	✓	✓	GA
Node.js	MSAL Node	msal-node	Quickstart	✓	✓	GA
Python	MSAL Python	msal	Quickstart	✓	✓	GA

¹ Supplemental terms of use for Microsoft Azure Previews apply to libraries in *Public preview*.

² The [Microsoft.IdentityModel](#) library only *validates* tokens - it cannot request ID or access tokens.

Desktop application

A desktop application is typically binary (compiled) code that displays a user interface and is intended to run on a user's desktop.

Because a desktop application runs on the user's desktop, it's considered a *public client* that's unable to store secrets securely.

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
Electron	MSAL Node.js	msal-node	—	✓	✓	Public preview
Java	MSAL4J	msal4j	—	✓	✓	GA
macOS (Swift/Obj-C)	MSAL for iOS and macOS	MSAL	Tutorial	✓	✓	GA
UWP	MSAL.NET	Microsoft.Identity.Client	Tutorial	✓	✓	GA

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW
WPF	MSAL.NET	Microsoft.Identity.Client	Tutorial	✓	✓	GA

¹ Supplemental terms of use for Microsoft Azure Previews apply to libraries in *Public preview*.

Mobile application

A mobile application is typically binary (compiled) code that displays a user interface and is intended to run on a user's mobile device.

Because a mobile application runs on the user's mobile device, it's considered a *public client* that's unable to store secrets securely.

PLATFORM	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
Android (Java)	MSAL Android	MSAL	Quickstart	✓	✓	GA
Android (Kotlin)	MSAL Android	MSAL	—	✓	✓	GA
iOS (Swift/Obj-C)	MSAL for iOS and macOS	MSAL	Tutorial	✓	✓	GA
Xamarin (.NET)	MSAL.NET	Microsoft.Identity.Client	—	✓	✓	GA

¹ Supplemental terms of use for Microsoft Azure Previews apply to libraries in *Public preview*.

Service / daemon

Services and daemons are commonly used for server-to-server and other unattended (sometimes called *headless*) communication. Because there's no user at the keyboard to enter credentials or consent to resource access, these applications authenticate as themselves, not a user, when requesting authorized access to a web API's resources.

A service or daemon that runs on a server is considered a *confidential client* that can store its secrets securely.

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW ¹
.NET	MSAL.NET	Microsoft.Identity.Client	Quickstart	✗	✓	GA
Java	MSAL4J	msal4j	—	✗	✓	GA

LANGUAGE / FRAMEWORK	PROJECT ON GITHUB	PACKAGE	GETTING STARTED	SIGN IN USERS	ACCESS WEB APIs	GENERALLY AVAILABLE (GA) OR PUBLIC PREVIEW
Node	MSAL Node	<code>msal-node</code>	Quickstart			GA
Python	MSAL Python	<code>msal-python</code>	Quickstart			GA

¹ [Supplemental terms of use for Microsoft Azure Previews](#) apply to libraries in *Public preview*.

Next steps

For more information about the Microsoft Authentication Library, see the [Overview of the Microsoft Authentication Library \(MSAL\)](#).

Azure AD Authentication and authorization error codes

4/12/2022 • 33 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.

For example, if you received the error code "AADSTS50058" then do a search in <https://login.microsoftonline.com/error> for "50058". You can also link directly to a specific error by adding the error code number to the URL: <https://login.microsoftonline.com/error?code=50058>.

Handling error codes in your application

The [OAuth2.0 spec](#) provides guidance on how to handle errors during authentication using the `error` portion of the error response.

Here is a sample error response:

```
{  
  "error": "invalid_scope",  
  "error_description": "AADSTS70011: The provided value for the input parameter 'scope' is not valid. The scope https://example.contoso.com/activity.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",  
  "error_uri": "https://login.microsoftonline.com/error?code=70011"  
}
```

PARAMETER	DESCRIPTION
<code>error</code>	An error code string that can be used to classify types of errors that occur, and should 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. Never use this field to react to an error in your code.
<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.
<code>error_uri</code>	A link to the error lookup page with additional information about the error. This is for developer usage only, do not present it to users. Only present when the error lookup system has additional information about the error - not all error have additional information provided.

The `error` field has several possible values - review the protocol documentation links and OAuth 2.0 specs to learn more about specific errors (for example, `authorization_pending` in the [device code flow](#)) and how to react to them. Some common ones are listed here:

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>	Some of the authentication material (auth code, refresh token, access token, PKCE challenge) was invalid, unparseable, missing, or otherwise unusable	Try a new request to the <code>/authorize</code> endpoint to get a new authorization code. Consider reviewing and validating that app's use of the protocols.
<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.

ERROR CODE	DESCRIPTION	CLIENT ACTION
<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. During development, this usually indicates an incorrectly setup test tenant or a typo in the name of the scope being requested.
<code>interaction_required</code>	The request requires user interaction. For example, an additional authentication step is required.	Retry the request with the same resource, interactively, so that the user can complete any challenges required.
<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.

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.
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.

ERROR	DESCRIPTION
AADSTS28002	Provided value for the input parameter scope '{scope}' is not valid when requesting an access token. Please specify a valid scope.
AADSTS28003	Provided value for the input parameter scope cannot be empty when requesting an access token using the provided authorization code. Please specify a valid scope.
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.
AADSTS500021	Access to '{tenant}' tenant is denied. AADSTS500021 indicates that the tenant restriction feature is configured and that the user is trying to access a tenant that is not in the list of allowed tenants specified in the header <code>Restrict-Access-To-Tenant</code> . For more information, see Use tenant restrictions to manage access to SaaS cloud applications .
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. To learn more, see the troubleshooting article for error AADSTS50003 . 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. To learn more, see the troubleshooting article for error AADSTS50011 .
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.

ERROR	DESCRIPTION
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.
AADSTS500212	NotAllowedByOutboundPolicyTenant - The user's administrator has set an outbound access policy that does not allow access to the resource tenant.
AADSTS500213	NotAllowedByInboundPolicyTenant - The resource tenant's cross-tenant access policy does not allow this user to access this tenant.
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	UnableToGeneratePairwiseldentifierWithMissingSalt - The salt required to generate a pairwise identifier is missing in principle. Contact the tenant admin.
AADSTS50043	UnableToGeneratePairwiseldentifierWithMultipleSalts

ERROR	DESCRIPTION
AADSTS50048	SubjectMismatchesIssuer - 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	<p>This error can result from two different reasons:</p> <ul style="list-style-type: none"> • IdsLocked - The account is locked because the user tried to sign in too many times with an incorrect user ID or password. The user is blocked due to repeated sign-in attempts. See Remediate risks and unlock users. • Or, sign-in was blocked because it came from an IP address with malicious activity. <p>To determine which failure reason caused this error, sign in to the Azure portal. Navigate to your Azure AD tenant and then Monitoring -> Sign-ins. Find the failed user sign-in with Sign-in error code 50053 and check the Failure reason.</p>
AADSTS50055	InvalidPasswordExpiredPassword - The password is expired. The user's password is expired, and therefore their login or session was ended. They will be offered the opportunity to reset it, or may ask an admin to reset it via Reset a user's password using Azure Active Directory .
AADSTS50056	Invalid or null password: password does not exist in the directory for this user. The user should be asked to enter their password again.
AADSTS50057	UserDisabled - The user account is disabled. The user object in Active Directory backing this account has been disabled. An admin can re-enable this account through PowerShell
AADSTS50058	<p>UserInformationNotProvided - Session information is not sufficient for single-sign-on. 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 encountered 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	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.
AADSTS50061	SignoutInvalidRequest - Unable to complete signout. The request was invalid.
AADSTS50064	CredentialAuthenticationError - Credential validation on username or password has failed.

ERROR	DESCRIPTION
AADSTS50068	SignoutInitiatorNotParticipant - Sign out has failed. The app that initiated sign out is not a participant in the current session.
AADSTS50070	SignoutUnknownSessionIdentifier - Sign out has failed. The sign out request specified a name identifier that didn't match the existing session(s).
AADSTS50071	SignoutMessageExpired - The logout request has expired.
AADSTS50072	UserStrongAuthEnrollmentRequiredInterrupt - User needs to enroll for second factor authentication (interactive).
AADSTS50074	UserStrongAuthClientAuthNRequiredInterrupt - Strong authentication is required and the user did not pass the MFA challenge.
AADSTS50076	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.
AADSTS50079	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.
AADSTS50085	Refresh token needs social IDP login. Have user try signing-in again with username -password
AADSTS50086	SasNonRetryableError
AADSTS50087	SasRetryableError - A transient error has occurred during strong authentication. Please try again.
AADSTS50088	Limit on telecom MFA calls reached. Please try again in a few minutes.
AADSTS50089	Authentication failed due to flow token expired. Expected - auth codes, refresh tokens, and sessions expire over time or are revoked by the user or an admin. The app will request a new login from the user.
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. To learn more, see the troubleshooting article for error AADSTS50105 .

ERROR	DESCRIPTION
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.
AADSTS501241	Mandatory Input '{paramName}' missing from transformation id '{transformId}'. This error is returned while Azure AD is trying to build a SAML response to the application. NameID claim or NamelIdentifier is mandatory in SAML response and if Azure AD failed to get source attribute for NameID claim, it will return this error. As a resolution, ensure you add claim rules in Azure Portal > Azure Active Directory > Enterprise Applications > Select your application > Single Sign-On > User Attributes & Claims > Unique User Identifier (Name ID).
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.
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.

ERROR	DESCRIPTION
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	KmsilInterrupt - This error occurred due to "Keep me signed in" interrupt when the user was signing-in. This is an expected part of the login flow, where a user is asked if they want to remain signed into their current browser to make further logins easier. For more information, see The new Azure AD sign-in and "Keep me signed in" experiences rolling out now! . You can 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.
AADSTS501481	The Code_Verifier does not match the code_challenge supplied in the authorization request.
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.

ERROR	DESCRIPTION
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.
AADSTS50170	MissingExternalClaimsProviderMapping - The external controls mapping is missing.
AADSTS50173	FreshTokenNeeded - The provided grant has expired due to it being revoked, and a fresh auth token is needed. Either an admin or a user revoked the tokens for this user, causing subsequent token refreshes to fail and require reauthentication. Have the user sign in again.
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.
AADSTS50194	Application '{appId}'('{appName}') is not configured as a multi-tenant application. Usage of the /common endpoint is not supported for such applications created after '{time}'. Use a tenant-specific endpoint or configure the application to be multi-tenant.
AADSTS50196	LoopDetected - A client loop has been detected. Check the app's logic to ensure that token caching is implemented, and that error conditions are handled correctly. The app has made too many of the same request in too short a period, indicating that it is in a faulty state or is abusively requesting tokens.
AADSTS50197	ConflictingIdentities - The user could not be found. Try signing in again.

ERROR	DESCRIPTION
AADSTS50199	<p>CmsilInterrupt - For security reasons, user confirmation is required for this request. Because this is an "interaction_required" error, the client should do interactive auth. This occurs because a system webview has been used to request a token for a native application - the user must be prompted to ask if this was actually the app they meant to sign into. To avoid this prompt, the redirect URI should be part of the following safe list:</p> <p>http:// https:// msauth://(iOS only) msauthv2://(iOS only) chrome-extension:// (desktop Chrome browser only)</p>
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 URL 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.
AADSTS53011	User blocked due to risk on home tenant.
AADSTS54000	MinorUserBlockedLegalAgeGroupRule
AADSTS54005	OAuth2 Authorization code was already redeemed, please retry with a new valid code or use an existing refresh token.
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.
AADSTS65002	Consent between first party application '{applicationId}' and first party resource '{resourceId}' must be configured via preauthorization - applications owned and operated by Microsoft must get approval from the API owner before requesting tokens for that API. A developer in your tenant may be attempting to reuse an App ID owned by Microsoft. This error prevents them from impersonating a Microsoft application to call other APIs. They must move to another app ID they register in https://portal.azure.com .
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). To learn more, see the troubleshooting article for error AADSTS650056 .
AADSTS650052	The app needs access to a service <code>(\"{name}\")</code> that your organization <code>\\"{organization}\\"</code> has not subscribed to or enabled. Contact your IT Admin to review the configuration of your service subscriptions.
AADSTS650054	The application asked for permissions to access a resource that has been removed or is no longer available. Make sure that all resources the app is calling are present in the tenant you are operating in.

ERROR	DESCRIPTION
AADSTS650056	Misconfigured application. This could be due to one of the following: the client has not listed any permissions for '{name}' 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. Or, check the certificate in the request to ensure it's valid. Please contact your admin to fix the configuration or consent on behalf of the tenant. Client app ID: {id}. Please contact your admin to fix the configuration or consent on behalf of the tenant.
AADSTS650057	Invalid resource. The client has requested access to a resource which is not listed in the requested permissions in the client's application registration. Client app ID: {appId} ({appName}). Resource value from request: {resource}. Resource app ID: {resourceAppId}. List of valid resources from app registration: {regList}.
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. To learn more, see the troubleshooting article for error AADSTS70001 .
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.
AADSTS700030	Invalid certificate - subject name in certificate is not authorized. SubjectNames/SubjectAlternativeNames (up to 10) in token certificate are: {certificateSubjects}.
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
AADSTS700054	Response_type 'id_token' is not enabled for the application. The application requested an ID token from the authorization endpoint, but did not have ID token implicit grant enabled. Go to Azure Portal > Azure Active Directory > App registrations > Select your application > Authentication > Under 'Implicit grant and hybrid flows', make sure 'ID tokens' is selected.
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.
AADSTS700084	The refresh token was issued to a single page app (SPA), and therefore has a fixed, limited lifetime of {time}, which cannot be extended. It is now expired and a new sign in request must be sent by the SPA to the sign in page. The token was issued on {issueDate}.
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.
AADSTS70043	The refresh token has expired or is invalid due to sign-in frequency checks by conditional access. The token was issued on {issueDate} and the maximum allowed lifetime for this request is {time}.
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. To learn more, see the troubleshooting article for error AADSTS75005 .

ERROR	DESCRIPTION
AADSTS7500514	A supported type of SAML response was not found. The supported response types are 'Response' (in XML namespace 'urn:oasis:names:tc:SAML:2.0:protocol') or 'Assertion' (in XML namespace 'urn:oasis:names:tc:SAML:2.0:assertion'). Application error - the developer will handle this error.
AADSTS750054	SAMLRequest or SAMLResponse must be present as query string parameters in HTTP request for SAML Redirect binding. To learn more, see the troubleshooting article for error AADSTS750054 .
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. To learn more, see the troubleshooting article for error AADSTS75011 .
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.
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.

ERROR	DESCRIPTION
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.
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	NotSupportedException - Unable to create the algorithm.

ERROR	DESCRIPTION
AADSTS9001023	The grant type is not supported over the /common or /consumers endpoints. Please use the /organizations or tenant-specific endpoint.
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.
AADSTS900144	The request body must contain the following parameter: '{name}'. Developer error - the app is attempting to sign in without the necessary or correct authentication parameters.
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.
AADSTS90020	The SAML 1.1 Assertion is missing ImmutableID of the user. Developer error - the app is attempting to sign in without the necessary or correct authentication parameters.
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.
AADSTS9002332	Application '{principalId}'('{{principalName}}) is configured for use by Azure Active Directory users only. Please do not use the /consumers endpoint to serve this request.
AADSTS90024	RequestBudgetExceededError - A transient error has occurred. Try again.
AADSTS90027	We are unable to issue tokens from this API version on the MSA tenant. Please contact the application vendor as they need to use version 2.0 of the protocol to support this.
AADSTS90033	MsodsServiceUnavailable - The Microsoft Online Directory Service (MSODS) is not available.

ERROR	DESCRIPTION
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.
AADSTS900432	Confidential Client is not supported in Cross Cloud request.
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	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: ./azuread-dev/v1-protocols-oauth-code.md . 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.
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. This error also might occur if the users are synced, but there is a mismatch in the <code>ImmutableID</code> (<code>sourceAnchor</code>) attribute between Active Directory and Azure AD. 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.
AADSTS900382	Confidential Client is not supported in Cross Cloud request.
AADSTS90095	AdminConsentRequiredRequestAccess- In the Admin Consent Workflow experience, an interrupt that appears when the user is told they need to ask the admin for consent.
AADSTS90099	The application '{appId}' ({appName}) has not been authorized in the tenant '{tenant}'. Applications must be authorized to access the customer tenant before partner delegated administrators can use them. Provide pre-consent or execute the appropriate Partner Center API to authorize the application.
AADSTS900971	No reply address provided.
AADSTS90100	InvalidRequestParameter - The parameter is empty or not valid.
AADSTS901002	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

ERROR	DESCRIPTION
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.
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

ERROR	DESCRIPTION
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.
AADSTS130008	NgcDeviceIsNotFound - The device referenced by the NGC key wasn't found.
AADSTS135010	KeyNotFound
AADSTS135011	Device used during the authentication is disabled.
AADSTS140000	InvalidRequestNonce - Request nonce is not provided.
AADSTS140001	InvalidSessionKey - The session key is not valid.
AADSTS165004	Actual message content is runtime specific. Please see returned exception message for details.
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.

ERROR	DESCRIPTION
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.
AADSTS7000215	Invalid client secret is provided. Developer error - the app is attempting to sign in without the necessary or correct authentication parameters.
AADSTS7000222	InvalidClientSecretExpiredKeysProvided - The provided client secret keys are expired. Visit the Azure portal to create new keys for your app, or consider using certificate credentials for added security: https://aka.ms/certCreds
AADSTS700005	InvalidGrantRedeemAgainstWrongTenant - Provided Authorization Code is intended to use against other tenant, thus rejected. OAuth2 Authorization Code must be redeemed against same tenant it was acquired for (/common or /{tenant-ID} as appropriate)
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.
AADSTS100007	AAD Regional ONLY supports auth either for MSIs OR for requests from MSAL using SN+I for 1P apps or 3P apps in Microsoft infrastructure tenants.
AADSTS1000031	Application {appDisplayName} cannot be accessed at this time. Contact your administrator.
AADSTS7000112	UnauthorizedClientApplicationDisabled - The application is disabled.
AADSTS7000114	Application 'appIdentifier' is not allowed to make application on-behalf-of calls.

ERROR	DESCRIPTION
AADSTS7500529	The value 'SAMLId-Guid' is not a valid SAML ID - Azure AD uses this attribute to populate the InResponseTo 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, id6c1c178c166d486687be4aaf5e482730 is a valid ID.

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.

Admin consent on the Microsoft identity platform

4/12/2022 • 4 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.

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*.

```
https://login.microsoftonline.com/{tenant}/v2.0/adminconsent  
?client_id=6731de76-14a6-49ae-97bc-6eba6914391e  
&scope=https://graph.microsoft.com/Calendars.Read https://graph.microsoft.com/Mail.Send  
&redirect_uri=http://localhost/myapp/permissions  
&state=12345
```

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>organizations</code> as seen in the example. Do not use 'common', as personal accounts cannot provide admin consent except in the context of a tenant. To ensure best compatibility with personal accounts that manage tenants, use the tenant ID when possible.
<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 URLs 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 (both user and app). In order to request app permissions, you must use the `/.default` value. If you don't want admins to see a given permission in the admin consent screen all the time when you use `/.default`, the best practice is to not put the permission in the required permissions section. Instead you can use dynamic consent to add the permissions you want to be in the consent screen at run time, rather than using `/.default`.

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
&scope=https://graph.microsoft.com/Calendars.Read https://graph.microsoft.com/Mail.Send
&state=12345
```

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>scope</code>	The set of permissions that were granted access to, for the application.
<code>admin_consent</code>	Will be set to <code>True</code> .

Error response

```

http://localhost/myapp/permissions
?admin_consent=True
&tenant=fa15d692-e9c7-4460-a743-29f2956fd429
&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
&state=12345

```

Adding to the parameters seen in a successful response, error parameters are seen as below.

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 error.
<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> 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](#)

National clouds

4/12/2022 • 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 Azure cloud, Azure ActiveDirectory(Azure AD) is deployed in the following national clouds:

- Azure Government
- Azure China 21Vianet
- Azure Germany ([Closed on October 29, 2021](#)). Learn more about [Azure Germany migration](#).

The individual national clouds and the global Azure cloud are cloud *instances*. Each cloud instance is separate from the others and has its own environment and *endpoints*. Cloud-specific endpoints include OAuth 2.0 access token and OpenID Connect ID token request endpoints, and URLs for app management and deployment, like the Azure portal.

As you develop your apps, use the endpoints for the cloud instance where you'll deploy the application.

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 PORTAL ENDPOINT
Azure portal for US Government	https://portal.azure.us
Azure portal China operated by 21Vianet	https://portal.azure.cn
Azure portal (global service)	https://portal.azure.com

Application endpoints

You can find the authentication endpoints for your application in the Azure portal.

1. Sign in to the [Azure portal](#).
2. Select **Azure Active Directory**.
3. Under **Manage**, select **App registrations**, and then select **Endpoints** in the top menu.

The **Endpoints** page is displayed showing the authentication endpoints for the application registered in your Azure AD tenant.

Use the endpoint that matches the authentication protocol you're using in conjunction with the **Application (client) ID** to craft the authentication request specific to your application.

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 China operated by 21Vianet	https://login.partner.microsoftonline.cn
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 global Azure:

- Authorization common endpoint is <https://login.microsoftonline.com/common/oauth2/v2.0/authorize>.
- Token common endpoint is <https://login.microsoftonline.com/common/oauth2/v2.0/token>.

For single-tenant applications, replace "common" in the previous URLs with your tenant ID or name. An example is <https://login.microsoftonline.com/contoso.com>.

Azure Germany (Microsoft Cloud Deutschland)

If you haven't migrated your application from Azure Germany, follow [Azure Active Directory information for the migration from Azure Germany](#) to get started.

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](#).

Some services and features in the global Azure cloud might be unavailable in other cloud instances like the national clouds.

To find out which services and features are available in a given cloud instance, see [Products available by region](#).

To learn how to build an application by using the Microsoft identity platform, follow the [Single-page application \(SPA\) using auth code flow tutorial](#). Specifically, this app will sign in a user and get an access token to call the Microsoft Graph API.

Next steps

Learn how to use the [Microsoft Authentication Library \(MSAL\) in a national cloud environment](#).

National cloud documentation:

- [Azure Government](#)
- [Azure China 21Vianet](#)
- [Azure Germany \(Closed on October 29, 2021\)](#)

Support and help options for developers

4/12/2022 • 2 minutes to read • [Edit Online](#)

If you need an answer to a question or help in solving a problem not covered in our documentation, it might be time to reach out to experts for help. Here are several suggestions for getting answers to your questions as you develop applications that integrate with the Microsoft identity platform.

Create an Azure support request



Explore the range of [Azure support options and choose the plan](#) that best fits you. There are two options to create and manage support requests in the Azure portal:

- If you already have an Azure Support Plan, [open a support request here](#).
- If you're not an Azure customer, you can open a support request with [Microsoft Support for business](#).

Post a question to Microsoft Q&A



Get answers to your identity app development questions directly from Microsoft engineers, Azure Most Valuable Professionals (MVPs), and members of our expert community.

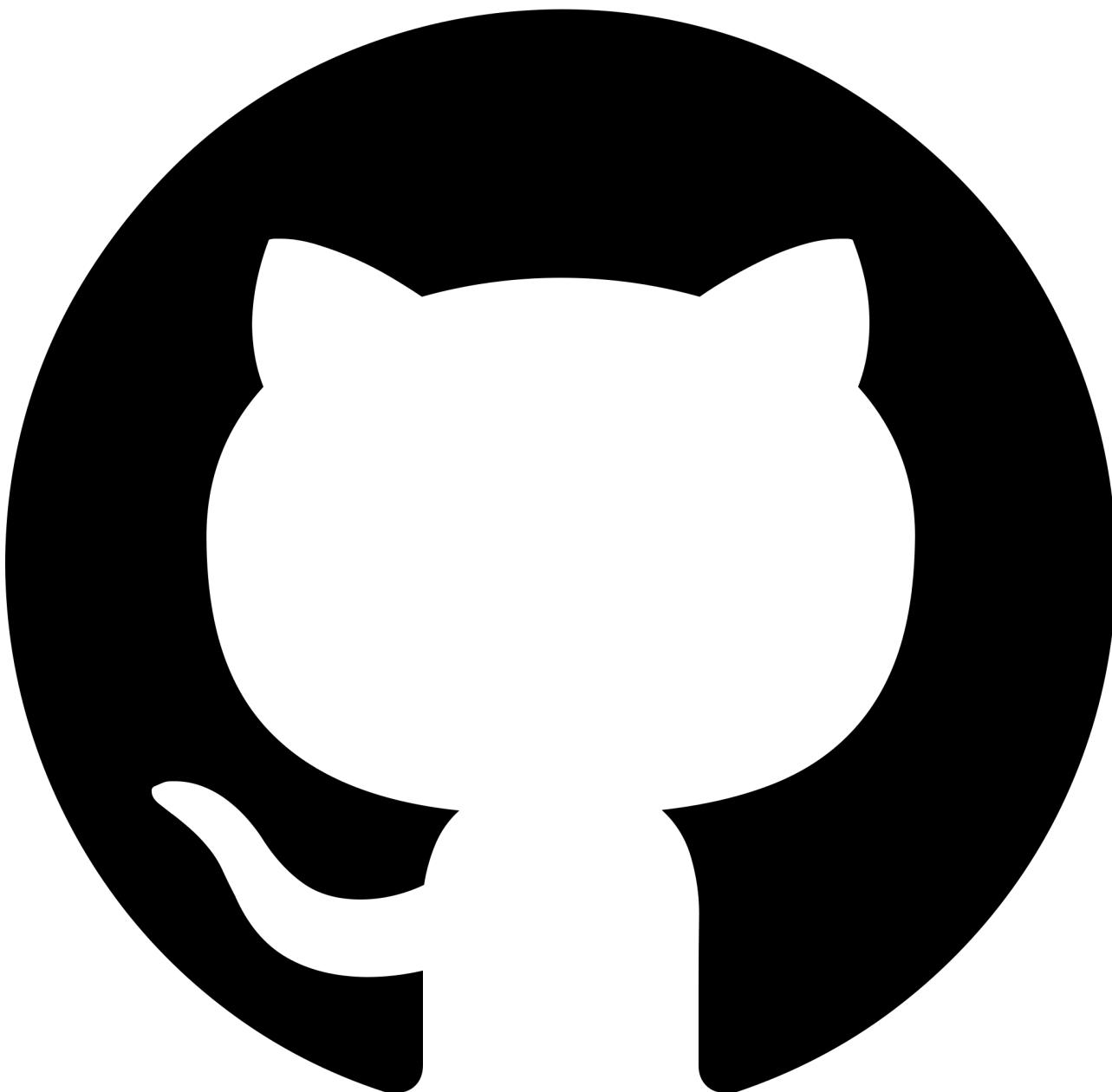
[Microsoft Q&A](#) is Azure's recommended source of community support.

If you can't find an answer to your problem by searching Microsoft Q&A, submit a new question. Use one of the following tags when you ask your [high-quality question](#):

COMPONENT/AREA	TAGS
Microsoft Authentication Library (MSAL)	[msal]
Open Web Interface for .NET (OWIN) middleware	[azure-active-directory]
Azure AD B2B / External Identities	[azure-ad-b2b]
Azure AD B2C	[azure-ad-b2c]
Microsoft Graph API	[azure-ad-graph]

COMPONENT/AREA	TAGS
All other authentication and authorization areas	[azure-active-directory]

Create a GitHub issue



If you need help with one of the Microsoft Authentication Libraries (MSAL), open an issue in its repository on GitHub.

MSAL	GITHUB ISSUES URL
MSAL for Android	https://github.com/AzureAD/microsoft-authentication-library-for-android/issues
MSAL Angular	https://github.com/AzureAD/microsoft-authentication-library-for-js/issues
MSAL for iOS and macOS	https://github.com/AzureAD/microsoft-authentication-library-for-objc/issues

MSAL	GITHUB ISSUES URL
MSAL Java	https://github.com/AzureAD/microsoft-authentication-library-for-java/issues
MSAL.js	https://github.com/AzureAD/microsoft-authentication-library-for-js/issues
MSAL.NET	https://github.com/AzureAD/microsoft-authentication-library-for-dotnet/issues
MSAL Node	https://github.com/AzureAD/microsoft-authentication-library-for-js/issues
MSAL Python	https://github.com/AzureAD/microsoft-authentication-library-for-python/issues
MSAL React	https://github.com/AzureAD/microsoft-authentication-library-for-js/issues

Stay informed of updates and new releases



- [Azure Updates](#): Learn about important product updates, roadmap, and announcements.
- [What's new in docs](#): Get to know what's new in the Microsoft identity platform documentation.
- [Azure Active Directory Identity Blog](#): Get news and information about Azure AD.
- [Tech Community](#): Share your experiences, engage, and learn from experts.

Share your product ideas

Have an idea for improving the Microsoft identity platform? Browse and vote for ideas submitted by others or submit your own:

<https://feedback.azure.com/d365community/forum/22920db1-ad25-ec11-b6e6-000d3a4f0789>

What's new for authentication?

4/12/2022 • 17 minutes to read • [Edit Online](#)

Get notified of updates to this page by pasting this URL into your RSS feed reader:

`https://docs.microsoft.com/api/search/rss?search=%22Azure+Active+Directory+breaking+changes+reference%22&locale=en-us`

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

ADFS users will see additional login prompts to ensure that the correct user is signed in.

Effective date: December 2021

Endpoints impacted: Integrated Windows Authentication

Protocol impacted: Integrated Windows Authentication

Change

Today, when a user is sent to ADFS to authenticate, they will be silently signed into any account that already has a session with ADFS. This silent sign-in occurs even if the user intended to sign into a different user account. To reduce the frequency of this incorrect sign in occurring, starting in December Azure AD will send the

`prompt=login` parameter to ADFS if the Web Account Manager in Windows provides Azure AD a `login_hint` during sign-in, which indicates a specific user is desired for sign-in.

When the above requirements are met (WAM is used to send the user to Azure AD to sign in, a `login_hint` is included, and the [ADFS instance for the user's domain supports `prompt=login`](#)) the user will not be silently signed in, and instead asked to provide a username to continue signing into ADFS. If they wish to sign into their existing ADFS session, they can select the "Continue as current user" option displayed below the login prompt. Otherwise, they can continue with the username that they intend to sign in with.

This change will be rolled out in December 2021 over the course of several weeks. It does not change sign in behavior for:

- Applications that use IWA directly
- Applications using OAuth
- Domains that are not federated to an ADFS instance

October 2021

Error 50105 has been fixed to not return `interaction_required` during interactive authentication

Effective date: October 2021

Endpoints impacted: v2.0 and v1.0

Protocol impacted: All user flows for apps [requiring user assignment](#)

Change

Error 50105 (the current designation) is emitted when an unassigned user attempts to sign into an app that an admin has marked as requiring user assignment. This is a common access control pattern, and users must often find an admin to request assignment to unblock access. The error had a bug that would cause infinite loops in well-coded applications that correctly handled the `interaction_required` error response. `interaction_required` tells an app to perform interactive authentication, but even after doing so Azure AD would still return an `interaction_required` error response.

The error scenario has been updated, so that during non-interactive authentication (where `prompt=none` is used to hide UX), the app will be instructed to perform interactive authentication using an `interaction_required` error response. In the subsequent interactive authentication, Azure AD will now hold the user and show an error message directly, preventing a loop from occurring.

As a reminder, Azure AD does not support applications detecting individual error codes, such as checking strings for `AADSTS50105`. Instead, [Azure AD guidance](#) is to follow the standards and use the [standardized authentication responses](#) such as `interaction_required` and `login_required`. These are found in the standard `error` field in the response - the other fields are for human consumption during troubleshooting.

You can review the current text of the 50105 error and more on the error lookup service:

<https://login.microsoftonline.com/error?code=50105> .

AppId Uri in single tenant applications will require use of default scheme or verified domains

Effective date: October 2021

Endpoints impacted: v2.0 and v1.0

Protocol impacted: All flows

Change

For single tenant applications, adding or updating the AppId URI validates that the domain in the HTTPS scheme URI is listed in the verified domain list in the customer tenant or that the value uses the default scheme (`api://{appId}`) provided by Azure AD. This could prevent applications from adding an AppId URI if the domain isn't in the verified domain list or the value does not use the default scheme. To find more information on verified domains, refer to the [custom domains documentation](#).

The change does not affect existing applications using unverified domains in their AppID URI. It validates only new applications or when an existing application updates an identifier URIs or adds a new one to the identifierUri collection. The new restrictions apply only to URIs added to an app's identifierUris collection after 10/15/2021. AppId URIs already in an application's identifierUris collection when the restriction takes effect on 10/15/2021 will continue to function even if you add new URIs to that collection.

If a request fails the validation check, the application API for create/update will return a `400 badrequest` to the client indicating `HostNameNotOnVerifiedDomain`.

The following API and HTTP scheme-based application ID URI formats are supported. Replace the placeholder values as described in the list following the table.

SUPPORTED APPLICATION ID URI FORMATS	EXAMPLE APP ID URIS
<code>api://<appId></code>	<code>api://fc4d2d73-d05a-4a9b-85a8-4f2b3a5f38ed</code>
<code>api://<tenantId>/<appId></code>	<code>api://a8573488-ff46-450a-b09a-6eca0c6a02dc/fc4d2d73-d05a-4a9b-85a8-4f2b3a5f38ed</code>
<code>api://<tenantId>/<string></code>	<code>api://a8573488-ff46-450a-b09a-6eca0c6a02dc/api</code>
<code>api://<string>/<appId></code>	<code>api://productapi/fc4d2d73-d05a-4a9b-85a8-4f2b3a5f38ed</code>
<code>https://<tenantInitialDomain>.onmicrosoft.com/<string></code>	<code>https://contoso.onmicrosoft.com/productsapi</code>
<code>https://<verifiedCustomDomain>/<string></code>	<code>https://contoso.com/productsapi</code>
<code>https://<string>.<verifiedCustomDomain></code>	<code>https://product.contoso.com</code>
<code>https://<string>.<verifiedCustomDomain>/<string></code>	<code>https://product.contoso.com/productsapi</code>

- `<appId>` - The application identifier (appId) property of the application object.
- `<string>` - The string value for the host or the api path segment.
- `<tenantId>` - A GUID generated by Azure to represent the tenant within Azure.
- `<tenantInitialDomain>` - `<tenantInitialDomain>.onmicrosoft.com`, where `<tenantInitialDomain>` is the initial domain name the tenant creator specified at tenant creation.
- `<verifiedCustomDomain>` - A [verified custom domain](#) configured for your Azure AD tenant.

NOTE

If you use the `api://` scheme, you add a string value directly after the "`api://`". For example, `api://<string>`. That string value can be a GUID or an arbitrary string. If you add a GUID value, it must match either the app ID or the tenant ID. The application ID URI value must be unique for your tenant. If you add `api://<tenantId>` as the application ID URI, no one else will be able to use that URI in any other app. The recommendation is to use `api://<appId>`, instead, or the HTTP scheme.

August 2021

Conditional Access will only trigger for explicitly requested scopes

Effective date: August 2021, with gradual rollout starting in April.

Endpoints impacted: v2.0

Protocol impacted: All flows using [dynamic consent](#)

Applications using dynamic consent today are given all of the permissions they have consent for, even if they were not requested in the `scope` parameter by name. This can cause an app requesting e.g. only `user.read`, but with consent to `files.read`, to be forced to pass the Conditional Access assigned for the `files.read` permission.

In order to reduce the number of unnecessary Conditional Access prompts, Azure AD is changing the way that unrequested scopes are provided to applications so that only explicitly requested scopes trigger Conditional Access. This change may cause apps reliant on Azure AD's previous behavior (namely, providing all permissions even when they were not requested) to break, as the tokens they request will be missing permissions.

Apps will now receive access tokens with a mix of permissions in this - those requested, as well as those they have consent for that do not require Conditional Access prompts. The scopes of the access token is reflected in the token response's `scope` parameter.

This change will be made for all apps except those with an observed dependency on this behavior. Developers will receive outreach if they are exempted from this change, as them may have a dependency on the additional conditional access prompts.

Examples

An app has consent for `user.read`, `files.readwrite`, and `tasks.read`. `files.readwrite` has Conditional Access policies applied to it, while the other two do not. If an app makes a token request for `scope=user.read`, and the currently signed in user has not passed any Conditional Access policies, then the resulting token will be for the `user.read` and `tasks.read` permissions. `tasks.read` is included because the app has consent for it, and it does not require a Conditional Access policy to be enforced.

If the app then requests `scope=files.readwrite`, the Conditional Access required by the tenant will trigger, forcing the app to show an interactive auth prompt where the Conditional Access policy can be satisfied. The token returned will have all three scopes in it.

If the app then makes one last request for any of the three scopes (say, `scope=tasks.read`), Azure AD will see that the user has already completed the Conditional access policies needed for `files.readwrite`, and again issue a token with all three permissions in it.

June 2021

The device code flow UX will now include an app confirmation prompt

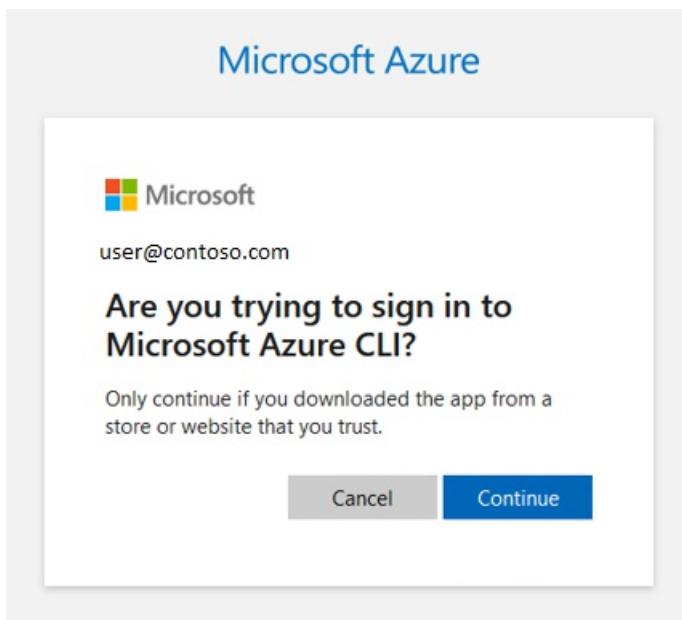
Effective date: June 2021.

Endpoints impacted: v2.0 and v1.0

Protocol impacted: The [device code flow](#)

As a security improvement, the device code flow has been updated to add an additional prompt, which validates that the user is signing into the app they expect. This is added to help prevent phishing attacks.

The prompt that appears looks like this:



May 2020

Bug fix: Azure AD will no longer URL-encode the state parameter twice

Effective date: May 2021

Endpoints impacted: v1.0 and v2.0

Protocol impacted: All flows that visit the `/authorize` endpoint (implicit flow and authorization code flow)

A bug was found and fixed in the Azure AD authorization response. During the `/authorize` leg of authentication, the `state` parameter from the request is included in the response, in order to preserve app state and help prevent CSRF attacks. Azure AD incorrectly URL-encoded the `state` parameter before inserting it into the response, where it was encoded once more. This would result in applications incorrectly rejecting the response from Azure AD.

Azure AD will no longer double-encode this parameter, allowing apps to correctly parse the result. This change will be made for all applications.

Azure Government endpoints are changing

Effective date: May 5th (Finishing June 2020)

Endpoints impacted: All

Protocol impacted: All flows

On 1 June 2018, the official Azure Active Directory (Azure AD) Authority for Azure Government changed from <https://login-us.microsoftonline.com> to <https://login.microsoftonline.us>. This change also applied to Microsoft 365 GCC High and DoD, which Azure Government Azure AD also services. If you own an application within a US Government tenant, you must update your application to sign users in on the `.us` endpoint.

Starting May 5th, Azure AD will begin enforcing the endpoint change, blocking government users from signing into apps hosted in US Government tenants using the public endpoint (microsoftonline.com). Impacted apps will begin seeing an error `AADSTS900439` - `USGClientNotSupportedOnPublicEndpoint`. This error indicates that the app is attempting to sign in a US Government user on the public cloud endpoint. If your app is in a public cloud tenant and intended to support US Government users, you will need to [update your app to support them explicitly](#). This may require creating a new app registration in the US Government cloud.

Enforcement of this change will be done using a gradual rollout based on how frequently users from the US Government cloud sign in to the application - apps signing in US Government users infrequently will see enforcement first, and apps frequently used by US Government users will be last to have enforcement applied. We expect enforcement to be complete across all apps in June 2020.

For more details, please see the [Azure Government blog post on this migration](#).

March 2020

User passwords will be restricted to 256 characters.

Effective date: March 13, 2020

Endpoints impacted: All

Protocol impacted: All user flows.

Users with passwords longer than 256 characters that sign in directly to Azure AD (as opposed to a federated IDP like ADFS) will be unable to sign in starting March 13, 2020, and be asked to reset their password instead. Admins may receive requests to help reset the users password.

The error in the sign in logs will be AADSTS 50052: InvalidPasswordExceedsMaxLength

Message:

The password entered exceeds the maximum length of 256. Please reach out to your admin to reset the password.

Remediation:

The user is unable to login because their password exceeds the permitted maximum length. They should contact their admin to reset the password. If SSPR is enabled for their tenant, they can reset their password by following the "Forgot your password" link.

February 2020

Empty fragments will be appended to every HTTP redirect from the login endpoint.

Effective date: February 8, 2020

Endpoints impacted: Both v1.0 and v2.0

Protocol impacted: OAuth and OIDC flows that use response_type=query - this covers the [authorization code flow](#) in some cases, and the [implicit flow](#).

When an authentication response is sent from login.microsoftonline.com to an application via HTTP redirect, the service will append an empty fragment to the reply URL. This prevents a class of redirect attacks by ensuring that the browser wipes out any existing fragment in the authentication request. No apps should have a dependency on this behavior.

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 [on-behalf-of 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 videos

4/12/2022 • 2 minutes to read • [Edit Online](#)

Learn the basics of modern authentication, the Microsoft identity platform, and the Microsoft Authentication Libraries (MSAL).

Microsoft identity platform for developers

Learn the key components and capabilities of the Microsoft identity platform.

[What is the Microsoft identity platform?](#)(14:54)

[The basics of modern authentication - Microsoft identity platform](#)(12:28)

[Overview: Implementing single sign-on in mobile applications - Microsoft Identity Platform](#) (20:30)

[Modern authentication: how we got here – Microsoft identity platform](#)(15:47)

Developer training series

In the Identity for Developers video series, Matthijs Hoekstra and Kyle Marsh provide a guided introduction to the Microsoft identity platform. Learn the key components and capabilities of the platform and how to use its authentication libraries to get started adding modern, secure authentication to your apps.

With content curated and honed over the many training sessions they've conducted, this series is a good place to start for any developer getting started with identity in Azure.

1 - [Overview of the Microsoft identity platform for developers](#) (33:55)

2 - [How to authenticate users of your apps with the Microsoft identity platform](#) (29:09)

3 - [Microsoft identity platform's permissions and consent framework](#) (45:08)

4 - [How to protect APIs using the Microsoft identity platform](#) (33:17)

5 - [Application roles and security groups on the Microsoft identity platform](#) (15:52)

Authentication fundamentals

If you're new to concepts like identity providers, security tokens, claims, and audience, this video series can help clear up the concepts and components in modern authentication.

1 - Basics: The concepts of modern authentication (4:33)

2 - Modern authentication for web applications (6:02)

3 - Web single sign-on (4:13)

4 - Federated web authentication (6:19)

5 - Native client applications - Part 1 (8:12)

6 - Native client applications - Part 2 (5:33)

Microsoft identity platform basics

Learn about the components of the Microsoft identity platform, the Microsoft Authentication Libraries (MSAL), and how these components interact with Azure Active Directory. The One Dev Question videos are 1-2 minutes in length.

[Microsoft identity platform overview](#)

[What is the MSAL family of libraries?](#)

[Scopes explained](#)

[What are brokers](#)

[What redirect URLs do](#)

[Tenants explained](#)

[Role of Azure AD](#)

[Role of Azure AD app objects](#)

[Organizational and personal Microsoft account differences](#)

[SPA and web app differences](#)

[What are Application Permissions vs Delegated Permissions?](#)

[What is Microsoft identity platform OpenID Connect certified?](#)

[What are the different Azure Active Directory app types and how do they compare?](#)

[If you use MSAL, what essential protocol concepts should you know?](#)

[What is the difference between ID tokens, access tokens, refresh tokens, and session tokens?](#)

[What is the relationship between an authorization request and tokens?](#)

[What aspects of using protocols does the MSAL libraries make easier?](#)

Migrate from v1.0 to v2.0

Learn about migrating to the latest version of the Microsoft identity platform, including moving from the Active Directory Authentication Library (ADAL) to MSAL.

[Why migrate from ADAL to MSAL](#)

[Migrating your ADAL codebase to MSAL](#)

[Advantages of MSAL over ADAL](#)

[What are the differences between v1 and v2 authentication?](#)