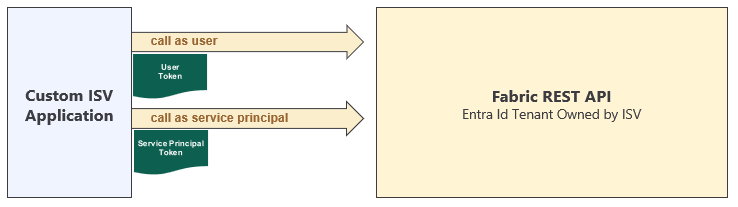
# Chapter 02 - Authentication and Authorization

Every call to the Fabric REST APIs executes under the identity of a specific security principal. The Fabric REST APIs support two types of security principal identity which are **user principals** and **service principals**. When you interact with the Entra Id Service to acquire an access token, the way in which you structure your request will determine whether the service returns a user token or a service principal token. The identity of the security principal found inside the access token is what actually determines the identity used to execute an API call.



When calling the Fabric REST APIs, you must decide whether to execute API calls as a user or as a service principal. In general, it is a better choice to execute API calls as a service principal. This is especially true when developing multitenant applications. That’s because there are several common problems when authenticating and executing API calls as a user.

The first problem with user authentication is that you need an actual human being to be present to interact with the Entra Id sign in process. Think about a scenario in which your application needs to complete a task at night with unattended execution when all the users are asleep. In network environments where administrators have enabled multifactor authentication (MFA), your code will often fail when attempting to acquire a user token.

While issues with MFA impacting user authentication are the most common, there are several other issues that can create problems as well. Here are some other things to keep in mind.

* When attempting to acquire a user token without interaction, developers often resort to using an insecure authentication flow where the user name and password are passed across the network.
* A user token can never elevate your code to run with a greater level of permissions than the current user.
* There is always the possibility that a user will leave the organization. This often leads to problems when that user is the owner or lone administrator of workspaces, connections and workspace items.

In a perfect world, you would be able to execute every API call as a service principal. But in the imperfect world in which we live, that is not always the case. To execute API calls as a service principal, you or your organization’s IT department must either create a custom Entra Id application or create a managed identity in Microsoft Azure. Furthermore, you need a user with Fabric administrator permissions who can access the Fabric Admin portal to configure a service principal with access to the Fabric REST APIs. If you cannot meet the requirements to configure a service principal, your only option will be to call Fabric REST API as a user.

There will also be edge cases in which a specific Fabric API endpoint does not support service principals. This is another case where executing API calls as a user is your only option.

As of October 2024, Microsoft’s efforts to add service principal support across all the Fabric REST APIs is still a work in progress. While many endpoints already support execution by a service principal, there are others that do not. Full support in the Fabric REST APIs for executing API calls as a service principal is expected to be completed by the end of 2024.

## Fabric REST API Permission Scopes

When you call the Entra Id service to acquire an access token, the request must include one or more permission scopes. Therefore, you should have a general understanding of how these permission scopes are created. Let’s start with an examination of how permission scopes are created specifically for the Fabric REST APIs.

A permission scope is a string value which begins with a **resource URI**. The role of the resource URI is to identify the target resource or API. In the Microsoft public cloud, the Fabric REST APIs have the following resource URI.

* **https://api.fabric.microsoft.com/**

Note that the resource URI for the Fabric REST API will change slightly when you are developing for Fabric environments in sovereign clouds and government clouds.

To create a Fabric permission scope, you parse the resource URI together with a permission name. Here are a few examples of permission scopes used with the Fabric REST APIs.

* **https://api.fabric.microsoft.com/.default**
* **https://api.fabric.microsoft.com/Workspace.ReadWrite.All**
* **https://api.fabric.microsoft.com/Item.ReadWrite.All**

Now that you’ve learned what permission scopes are, the next step is learning how to put them to use when acquiring access tokens in an authentication flow. The choice of which Fabric REST API permission scopes to use depends on whether you are acquiring an access token for a user or a service principal.

## Authenticate and Acquire Access Tokens for a Service Principal

There are two ways to set up a custom application to execute Fabric REST API calls as a service principal. The first way is to create a custom application in the Entra Id Service. When you do this, the Entra Id service will automatically create a service principal for you. The second way is using a Microsoft Azure subscription to create a managed identity. Creating a managed identity also results in the automatic creation of a service principal.

So what’s the difference between creating a custom application in the Entra Id Service versus creating a managed identity in Microsoft Azure? That’s an important question you should be able to answer. Let’s examine the requirements of each approach to illustrate the benefits and drawbacks.

Let’s start by covering the requirements of using an Entra Id application to authenticate as a service principal. First, you must create custom application in Entra Id service and configured it with either a client secret or a client certificate. The client secret or client certificate must be made available to the application at runtime because it must be passed to the Entra Id Service to authenticate when acquiring an access token. If you meet these requirements, the Entra Id Service will return an access token which includes the service principal object Id.

Now let’s contrast creating an Entra Id application to creating a managed identity. First, you will need an Azure subscription to create a managed identity. The managed identity must also be associated with Azure resource such as a Web App, a Function App or a virtual machine (VM). Once the managed identity has been created and properly configured, it can use used to acquire an access token for a service principal. The most significant benefit is that it eliminates the need for your application deal with a client secret or a client certificate.

Using a custom Entra Id application requires your application to deal with secret credentials, while a managed identity does not. Microsoft recommends using managed identities over custom Entra Id applications in production because it eliminates the need for creating and protecting secret authentication credentials.

### Acquire Access Tokens with a Custom Entra Id Application

You use **Client Credentials Flow** to acquire an access token for a service principal with an Entra Id application. This is the authentication flow which requires you to pass a client secret or a client certificate to authenticate. When using Client Credentials Flow, you should always use the default permission scope of the Fabric REST APIs which is **https://api.fabric.microsoft.com/.default**.

In C#, you can use a string array to create a set of one or more permission scopes.

public static readonly string[] Default = new string[] {

"https://api.fabric.microsoft.com/.default"

};

While you can write code that directly interacts with the Entra Id Service to acquire access tokens, Microsoft recommends instead that you use the Microsoft Authentication Libraries (MSAL). By integrating one of the MSAL libraries into your development projects, it will make your code easier to write. It will also make your code more secure because MSAL libraries are fully tested and frequently updated to deal with emerging security threats.

Let’s look at an example that uses the .NET version of MSAL named **Microsoft.Identity.Client** which can be added to a .NET project as a NuGet package. The **GetAccessTokenForServicePrincipal** method shown in the following code listing implements Client Credentials Flow to acquire an access token for a service principal.

private static string GetAccessTokenForServicePrincipal() {

string clientId = AppSettings.ServicePrincipalAuthClientId;

string clientSecret = AppSettings.servicePrincipalAuthClientSecret;

string tenantId = AppSettings.ServicePrincipalAuthTenantId;

string tenantSpecificAuthority = "https://login.microsoftonline.com/" + tenantId;

string[] scopes = new string[] { "https://api.fabric.microsoft.com/.default" };

var appConfidential = ConfidentialClientApplicationBuilder.Create(clientId)

.WithClientSecret(clientSecret)

.WithAuthority(tenantSpecificAuthority)

.Build();

return appConfidential.AcquireTokenForClient(scopes).ExecuteAsync().Result.AccessToken;

}

The **GetAccessTokenForServicePrincipal** method retrieves application configuration settings for the client Id, the client secret and the tenant Id for an Entra Id application. These three application settings are then passed to the Entra Id service in the call to **AcquireTokenForClient** when acquiring access tokens using Client Credentials Flow.

If you are not developing with C# and.NET, you should be aware that Microsoft provides other versions of MSAL for developers using other programming languages, platforms and frameworks. You can read more about the full scope of MSAL library support at the following URL.

* [**https://learn.microsoft.com/en-us/entra/identity-platform/msal-overview**](https://learn.microsoft.com/en-us/entra/identity-platform/msal-overview)

In the past, the biggest security concern in large organizations has been to ensure that user credentials are never compromised. However, things have changed over the last few years as user authentication has been strengthened through MFA. Microsoft now sees that compromised security credentials for a service principal poses a greater potential threat.

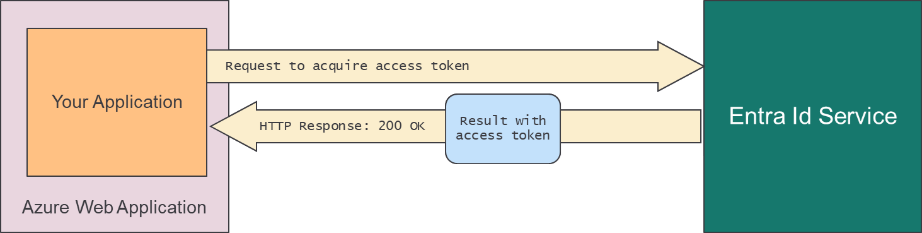
Crafty attackers have been able to discover and steal client secrets in places where a client secret should never be found. The obvious example of this is a forgetful developer who mistakenly published source code with a client secret to a public GitHub repository. The threat of compromised service principal credentials has led many companies to create defensive security policies such as one where source files checked into source control are scanned to ensure the code doesn’t contain any client secrets.

The issue discussed here leads to an important question. Does your organization really want to be responsible for creating and protecting service principal credentials such as client secrets or client certificates? If your answer is NO, then you’ll want to read on into the next section where you will learn how using managed identities can eliminate the requirements of your application to work with client secrets or client certificates.

### Acquire Access Tokens with a Managed Identity

The primary advantage of using a managed identity to execute API calls as a service principal is that your application doesn’t need to deal with a client secret or a client certificate to authenticate. Instead, you leave it up to the Entra Id Service to authenticate a service principal using other means. Let’s step through the fundamentals of how authentication works with a managed identity.

A managed identity is associated with an Azure resource such as a Web app, a Function app or a virtual machine. When your application runs on an Azure resource such as a Web app, it can authenticate with the Entra Id service using any managed identity that’s been associated with that Azure resource.



When your application authenticates using a managed identity, the Entra Id Service is able to verify where the token request originated. If it can verify that the token request originated from an Azure resource with an associated managed identity, authentication succeeds and an access token is returned to the caller.

Microsoft Azure supports both System-assigned Managed Identities and User-assigned Managed Identities. While both types of managed identity can be used to acquire access tokens for the Fabric REST APIs, you should understand some of the fundamental differences between them.

**System-assigned Managed Identities (SAMIs)** are easier to use because there is always a one-to-one relationship between the SAMI and an Azure resource. For example, an Azure Web application can only have one SAMI. You can create a SAMI with just a few clicks from the **Settings > Identity** page of an Azure resource in the Azure portal. Once it is created, it is ready to use.

A screenshot of a computer

Description automatically generated

When you authenticate using a SAMI, you don’t have to pass a client Id to identity the managed identity. That’s because there can only be one SAMI associated with the hosting Azure resource. Developers working on the .NET platform can leverage **Azure.Identity** library to implement the authentication flow required for a managed identity. Here is a code sample which uses a SAMI to authenticate and acquire an access token for a service principal.

private string GetAccessTokeForSami() {

ManagedIdentityCredential credential = new Azure.Identity.ManagedIdentityCredential();

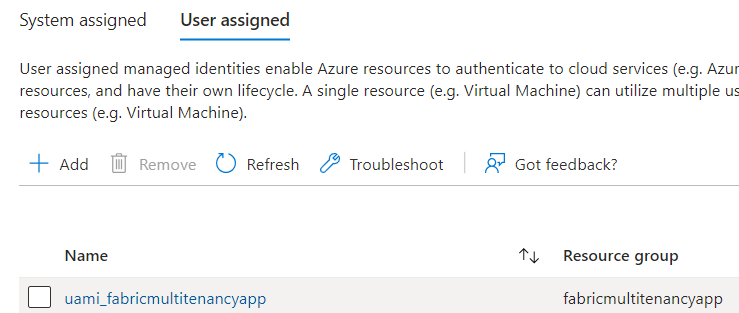
Azure.Core.AccessToken token = credential.GetToken(new Azure.Core.TokenRequestContext(scopes));

return token.Token;

}

**User-assigned Managed Identities (UAMIs)** provide more flexibility because they allow for many-to-many associations to Azure resources. For this reason, Microsoft generally recommends using UAMI instead of SAMIs. With UAMIs, you can create a set of managed identities and associate them all with a single Azure resource. Likewise, you can associate a single UAMI with multiple Azure resources.

Working with UAMIs requires an extra step. First, you create the UAMI from the **Managed Identity** page in the Azure portal. After creating the UAMI, you must then navigate in the Azure portal to the **Settings > Identity** page of an Azure resource. From there you can click the **Add** button to create the association between the UAMI and that Azure resource.



When using a UAMI, you must pass a Client Id to authenticate and acquire an access token. That’s because there isn’t a default one-to-one mapping between the hosting Azure resource and an associated UAMI.

private string GetAccessTokeForUAMI(string[] Scopes) {

string clientId = AppSettings.managedIdentityClientId;

string[] scopes = new string[] { "https://api.fabric.microsoft.com/.default" };

ManagedIdentityCredential credential = new Azure.Identity.ManagedIdentityCredential(clientId);

Azure.Core.AccessToken token = credential.GetToken(new Azure.Core.TokenRequestContext(Scopes));

return token.Token;

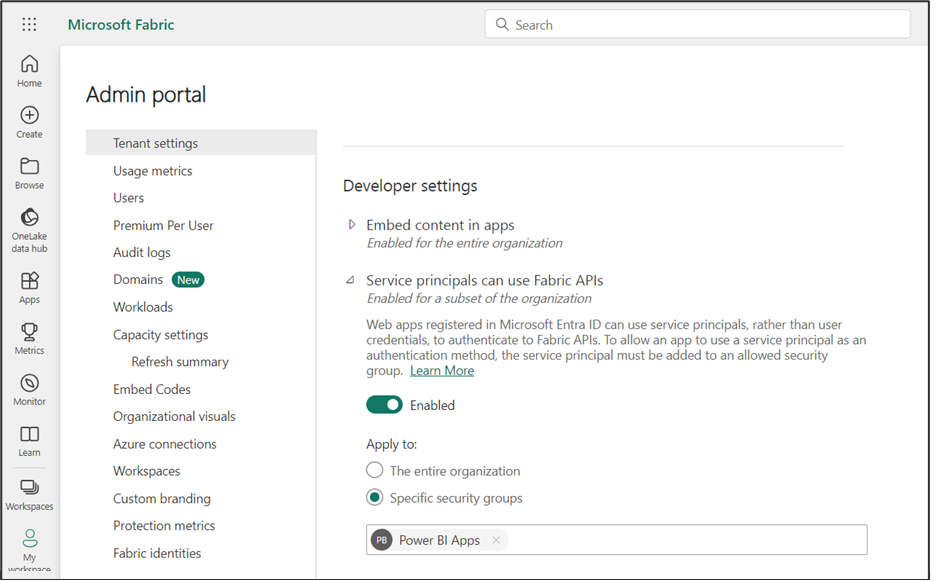
}

You have just seen that authenticating with a UAMI requires a Client Id while a SAMI does not. However, the more significant aspect is that neither SAMIs nor UAMIs require you to pass a client secret or client certificate to authenticate. Both SAMIs and UAMIs eliminate your application’s need to track secret credentials.

### Configure Service Principal Access to the Fabric REST API

Up to this point, you’ve learned how to authenticate and acquire access tokens for a service principal. You can accomplish this using either a custom Entra Id application or a managed identity. However, there still remains an extra configuration step that needs accomplished before a service principal has the permissions it needs to call the Fabric REST APIs. If you miss this step, calling to the Fabric REST APIs under the identity of the service principal will fail with an Access Denied error.

By default, a service principal does not possess the required permissions to call the Fabric REST APIs. Instead, a service principal must be configured in the Fabric admin portal using the tenant-level setting **Service Principals can use Fabric REST API**.



One complication with the **Service Principals can use Fabric REST API** setting is that you cannot directly add a service principal in the Fabric Admin portal. Instead, you can only configure this setting with users and Entra Id groups. The way to configure permissions for a service principal is by following these steps.

* Create an Entra Id group
* Add this Entra Id group to the **Service Principals can use Fabric REST API** setting
* Add service principals as members of the Entra Id group.

A close-up of a sign

Description automatically generated

Once you have configured a service principal with the **Service Principals can use Fabric REST API** setting, it will have the permissions it needs to call the Fabric REST APIs. But this does not mean that the service principal will automatically have access to any workspaces or workspace items. That’s because a service principal has no default access to any workspaces. In this sense, a service principal is treated just like a user.

If you want to enable a service principal with access to a specific workspace, you must assign the service principal to a workspace role. You can add a service principal into a workspace role such as Admin, Member, Contributor or Visitor. Once you add a service principal to a workspace in a role such as Admin or Member, the service principal will have the permissions it needs to begin creating workspace items.

In multitenant application development, it’s a best practice to automate the creation of workspaces using service principals. When you create a workspace using a service principal, it’s automatically assigned to the Admin role which provides full read-write access to the workspace and the workspace items inside.

### Configure Service Principal with Admin Access

The Fabric REST APIs provide a special set of Admin APIs that are made available to a smaller set of privileged users and service principals. The Fabric Admin APIs provide the caller with the ability to discover and inspect every workspace and every connection across the current Entra Id tenant. This is far more powerful than the non-Admin APIs because the caller can access every workspace without the requirement of being assigned workspace roles.

A screenshot of a computer

Description automatically generated

Many organization use the Fabric Admin APIs to build custom administrative applications that manage and monitor their Fabric environment. The Fabric Admin APIs can be used to discover workspaces and to inspect the workspace items inside. The Fabric Admin APIs can also inspect every connection across the tenant and alert the IT staff when users are creating connections that violate company-specific policies.

Now let’s discuss what is required to call the Fabric Admin APIs. That depends on whether the caller is a user or a service principal. To call the Fabric Admin API as a user, the user must first be configured in Entra Id in the role assignment of **Fabric Administrator**. This topic will be covered in greater detail later in this chapter.

Things are a bit different when configuring Admin API access for a service principal. Configuring Admin API access for service principal is accomplished in the **Admin API settings** section of the Fabric Admin portal using another tenant-wide setting named **Service principals can access read-only admins APIs**.

A screenshot of a computer

Description automatically generated

You can separate the Fabric REST APIs into two categories. There are the Admin APIs and the non-Admin APIs. As it turns out, the non-Admin APIs have their own name which is the **Fabric User APIs**. This can be confusing at first because the name suggests the User API for users not service principals. That’s not true. The Fabric User APIs are meant for both users and service principal.

The key to remember is that the Fabric Admin APIs allow you to discover and access every workspace and every connection on a tenant-wide basis. This is quite different than with the Fabric User APIs where caller can only access workspaces and connections in which they’ve been configured with access through role assignments.

## Authenticate and Acquire Access Tokens for a User

You have just learned how to authenticate and acquire access tokens to execute API calls as a service principal. Now it’s time to examine how things change when authenticating and acquiring access tokens to execute API calls as a user. Authenticating as a user is more complex because it relies on authentication flows which require interactive behavior on the part of the user. For example, users are often forced to interactively respond to MFA challenges by typing an access code into their mobile phone.

Writing code to acquire user tokens also introduces a second complication that doesn’t apply to service principals. Any request to acquire a user token must include a granular set of delegated permission scopes. Now it’s time to discuss how delegated permission scopes fit into the delegated access model which is part of Entra Id.

### Delegated Permission Scopes

The user authorization model in Entra Id is based upon the concept of **delegated access**. With delegated access, API calls are not executed with the same identity and the same permissions as the user. Instead, API calls executed with a user token are said to execute **on behalf of the user** which is very different than **by the user**.

When you execute an API call with a user token, the call executes with an application identity which is distinct from the user identity. Furthermore, the API call doesn’t execute with the full set of permissions that have been granted to the user. Instead, the API call executes with a subset of the user’s permissions as defined by the set of delegated permission scopes found inside the access token.

Let’s start with an example. Consider the following set of delegated permission scopes.

public static readonly string[] TenantProvisioning = new string[] {

"https://api.fabric.microsoft.com/Workspace.ReadWrite.All",

"https://api.fabric.microsoft.com/Connection.ReadWrite.All",

"https://api.fabric.microsoft.com/Item.ReadWrite.All",

"https://api.fabric.microsoft.com/Item.Execute.All",

“https://api.fabric.microsoft.com/Item.Reshare.All”

};

As you can see, this set of delegated permission scopes includes the **Workspace.ReadWrite.All** permission. This delegated permission scope grants your code read-write access to every workspace in which the user has been assigned permissions with read-write access. But the **Workspace.ReadWrite.All** permission does not give your code access to any workspaces that are not accessible to the current user.

Think about the scenario where the user has been granted read-only access to a specific workspace. In this case, your code will be limited to read-only access despite the presence of the **Workspace.ReadWrite.All** permission in the access token. The key point is that delegated permission scopes will never provide your code with a level of permissions greater than the current user.

The previous example includes **item-generic permission scopes** such as **Item.ReadWrite.All**. Item-generic permission scopes are used to grant permissions to every type of workspace item including lakehouses, notebooks, data pipelines, semantic models and reports.

public static readonly string[] GeneralProvisioningScopes = new string[] {

"https://api.fabric.microsoft.com/Item.ReadWrite.All",

"https://api.fabric.microsoft.com/Item.Execute.All",

“https://api.fabric.microsoft.com/Item.Reshare.All”

};

You can use **item-specific permission scopes** if you’re concerned with the security principle of least privilege and you would rather configure delegated permissions in a more granular fashion. Think about developing a solution where your code needs access to semantic models and reports but not for any other type of workspace items. You could use the following set of delegated permission scopes in a token acquisition request to achieve that end.

public static readonly string[] ItemSpecificProvisioningScopes = new string[] {

"https://api.fabric.microsoft.com/SemanticModel.ReadWrite.All",

"https://api.fabric.microsoft.com/SemanticModel.Execute.All",

“https://api.fabric.microsoft.com/SemanticModel.Reshare.All”,

"https://api.fabric.microsoft.com/Report.ReadWrite.All",

"https://api.fabric.microsoft.com/Report.Execute.All",

“https://api.fabric.microsoft.com/Report.Reshare.All”

}

Imagine you are writing code to acquire user tokens. Wouldn’t it be helpful for you to see a complete reference which lists all the delegated permission scopes supported by the Fabric REST APIs? Here is a handy little developer trick. You can run the following PowerShell script which uses the Microsoft Graph PowerShell SDK to generate a complete list of delegated permission scope support by the Fabric REST API.

if (!(Get-MgContext)) { Connect-MgGraph }

$outputFile = "$PSScriptRoot\FabricServicePermissions.txt"

$fabricServiceAppId = "00000009-0000-0000-c000-000000000000"

$filter = "appId eq '" + $fabricServiceAppId + "'"

$fabricService = Get-MgServicePrincipal -Filter $filter -Property Oauth2PermissionScopes

$fabricServicePerms = $fabricService | Select-Object -ExpandProperty Oauth2PermissionScopes

"--- Fabric Service API Delegated Permission Scopes ---" | Out-File -FilePath $outputFile

$permList = $fabricServicePerms | Sort-Object Type, Value | Format-Table Value, Type, Id

$permList | Out-File -FilePath $outputFile -Append

Notepad $outputFile

The first time you run the script, you’ll be prompted to sign in to establish a connection to the Microsoft Graph API. Once connected, this script calls **Get-MgServicePrincipal** to obtain a reference to the service principal for the Fabric Service. After that, the script queries the service principal’s **Oauth2PermissionScopes** property which returns a list of delegated permission scopes which is then displayed as a simple text file in Notepad.

A screenshot of a computer

Description automatically generated

There are over 100 different delegated permission scopes defined by the Fabric REST APIs. If your application is designed to execute API calls using user tokens, you must ensure your application acquires access token which contain all the delegated permission scopes that required by your code. If you miss a delegated permission scope required by a particular Fabric API endpoint, calls to that endpoint will be rejected with an Access Denied error.

Remember, you can alwayts eliminate the complexity of determining which delegated permission scopes to include by designing your application to exeute API calls using a service principal.

### Implement User Authentication Flows using MSAL

The way to implement user authentication flows changes significantly depending upon the type of project you are developing. For example, there are versions of MSAL to use when you’re developing a web application with server-side code. There are other versions of MSAL to use when you’re developing a single page application (SPA) with a framework such as React.js where all the programming logic is written in client-side code. The **FabricIsvPlaybook** developer sample uses a version of MSAL that is mainly used in desktop application and mobile applications.

Earlier you saw how to acquire tokens for a service principal using the .NET version of MSAL named **Microsoft.Identity.Client**. Now we will use the same library to implement a user authentication flow to acquire user tokens. The **GetAccessTokenForUser** method shown in the following listing demonstrates how to implement an interactive authentication flow to acquire a user token.

private const string tenantCommonAuthority = "https://login.microsoftonline.com/organizations";

private static string GetAccessTokenForUser(string[] scopes) {

string clientId = AppSettings.UserAuthClientId;

string redirectUri = AppSettings.UserAuthRedirectUri;

// create new public client application

var appPublic = PublicClientApplicationBuilder.Create(clientId)

.WithAuthority(tenantCommonAuthority)

.WithRedirectUri(redirectUri)

.Build();

// redirect user to browser with Entra Id Login experience

var authResult = appPublic.AcquireTokenInteractive(scopes).ExecuteAsync().Result;

// return access token to caller

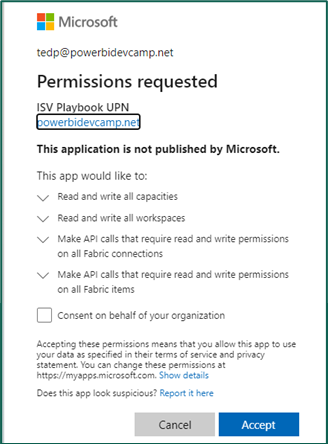
return authResult.AccessToken;

}

The code in the **GetAccessTokenForUser** method starts by creating a **PublicClientApplication** object which is initialized using the Client Id and Redirect URI of a custom Entra Id application. The code then calls **AcquireTokenInteractive** which starts the user authentication flow into motion. MSAL.NET responds by redirecting the user in the browser to a login page in the Entra Id Service where the user is prompted to sign in. Once the user completes the interactive sign in process, the call to **AcquireTokenInteractive** returns a response to the caller which contains an access token.

### Obtain User Consent for Delegate Permissions

Another important issue that can potentially impact an application’s ability to acquire user tokens is user consent. The fundamental idea is that the delegated access model requires that the user gets to approve of any delegated permission scope requested by an application. As a consequence, the Entra Id Service prompts the user with the **Permissions requested** consent dialog the first time a delegated permission scope is requested by a specific user.



The Entra Id Service remembers the list of delegated permission scopes to which the user has consented. That usually means that a user only see the **Permissions requested** consent dialog the first time they log into an application. However, this is not always the case.

An application can start by requesting a minimal set of delegated permission scopes in the initial request for a user token. If a user then invokes a command which require a greater set of permissions, the application can request an access token with additional delegated permissions scopes to which the user has not yet consented. Entra Id will then prompt the user with the **Permissions requested** consent dialog again to ensure the user has consented to each and every delegated permission scope that is added to an access token acquired by the client application.

### Admin-restricted permissions

Earlier you learned the Fabric REST APIs provide a special set of Admin APIs made available to a set of privileged users and service principals. These Admin APIs allow the caller with the ability to discover and inspect every workspace and connection with a tenant-wide scope.

There are two important requirements for executing Fabric Amin APIs with a user token. First, the user account must be assigned to the **Fabric Administrator** role in Entra Id as shown in the following screenshot.

A screenshot of a computer

Description automatically generated

The second requirement is that the user tokens contains one of the Admin delegated permission scopes which are **Tenant.Read.All** and **Tenant.ReadWrite.All**.

public static readonly string[] FabricAdminScopes = new string[] {

"https://api.fabric.microsoft.com/Tenant.ReadWrite.All"

}

Note that any attempt to acquire an access token with the delegated permission scope of **Tenant.Read.All** or **Tenant.ReadWrite.All** will fail if the user hasn’t been configured as a Fabric administrator in the Entra Id Service.

### Authenticate Users with Microsoft’s Azure PowerShell Application

Image you’re in a situation in which you want to start experimenting with the Fabric REST APIs, but you’re working in a Fabric environment in which you cannot create your own Entra Id applications. This could be the case if you don’t have the required level of permissions in your organization’s Entra Id directory. Or maybe there’s just too much bureaucratic red tape involved with getting the IT department in your organization to create a custom Entra Id application on your behalf.

Fortunately, Microsoft provides a built-in Entra Id application which you can use to authenticate users and to acquire access tokens for the Fabric REST API. This Entra Id application is the **Azure PowerShell application** which is automatically installed and available in every Entra Id tenant. You can acquire user tokens by using the Azure PowerShell application which has a Client Id of **1950a258-227b-4e31-a9cf-717495945fc2** and a configured Redirect Uri of **http://localhost**.

The first benefit of authenticating users with Azure PowerShell application is that it eliminates your need for creating a custom Entra Id application. This can be valuable for developers in environments in which they cannot create their own Entra Id applications. A second benefit is that acquiring access tokens with the Azure PowerShell application simplifies dealing with delegated permission scopes.

When acquiring user tokens with the Azure PowerShell application, you only need to request a single delegated permission scope named **user\_impersonation**. The important thing to understand is that **user\_impersonation** is the all-powerful delegated permission scope. In other words, it automatically executes your API calls with the same level of permissions that have been granted to the current user.

The **GetAccessTokenForAzurePowershell** method shown in the following code listing illustrates using the **Azure PowerShell application** to acquire user tokens with the **user\_impersonation** delegated permission scope.

private static string GetAccessTokenForAzurePowershell() {

// Azure PowerShell application uses the same client Id across all Entra Id tenants

const string azurePowershellClientId = "1950a258-227b-4e31-a9cf-717495945fc2";

const string azurePowershellRedirectUri = "http://localhost";

string[] scopes = { "https://api.fabric.microsoft.com/user\_impersonation" };

// create new public client application

var appPublic = PublicClientApplicationBuilder.Create(azurePowershellClientId)

.WithRedirectUri(azurePowershellRedirectUri)

.WithAuthority(tenantCommonAuthority)

.Build();

// redirect user to browser with Entra Id Login experience

var authResult = appPublic.AcquireTokenInteractive(scopes).ExecuteAsync().Result;

// return access token result to caller

return authResult.AccessToken;

}

Keep in mind that the Azure PowerShell application is something that is best used in development environments. It should mainly be used in scenarios in which you’re creating a POC or just experimenting with the Fabric REST APIs. The Azure PowerShell application is not meant to be used when you deploy a custom application to production.

Remember you can only use the Azure PowerShell application in desktop applications because it’s configured with a Redirect Uri of **http://localhost**. When developing a web application, you cannot authenticate users with the Azure PowerShell application because you are not able to configure a custom Redirect Uri.

## The EntraIdTokenManager class

The **FabricIsvPlaybook** developer sample contains a class named **EntraIdTokenManager** which encapsulates all the application’s code to authenticate and acquire access tokens from the Entra Id Service. Other classes in the project such as **FabricRestApi** can acquire an access token by calling the **GetFabricAccessToken** method.

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

The **EntraIdTokenManager** class supports three different authentication modes.

1. Authenticate as a service principal using a confidential Entra Id application
2. Authenticate as a user using a public Entra Id application
3. Authenticate as a user using the built-in Azure PowerShell application

The **FabricIsvPlaybook** project makes it easy to switch between authentication modes. Inside the source file named **EntraIdTokenManager,** there is a C# enumeration named **AppAuthenticationMode** which contains three named constants for the three possible authentication modes.

public enum AppAuthenticationMode {

ServicePrincipalAuth,

UserAuth,

UserAuthWithAzurePowershell

}

In the **AppSettings** class, there’s a static property named **AuthenticationMode** which is based on enum named **AppAuthenticationMode**. The default value is set to **UserAuthWithAzurePowershell**.

public static AppAuthenticationMode AuthenticationMode = AppAuthenticationMode.UserAuthWithAzurePowershell;

The default authentication mode setting of **UserAuthWithAzurePowershell** has been chosen because it’s the only one that works automatically without requiring you to create a custom application in Entra Id. The other two possible authentication mode settings require you to create a custom Entra Id application and to update constants in the **AppSettings** class with metadata such as the application’s Client Id.

### Create a Custom Entra Id Application for User Authentication

The enable user authentication, you must create a new application in the Entra Id Service. You should create this application as a public application by adding a **Redirect URI** using the **Public client/native (mobile or desktop)** setting and a Redirect URI value of **http://localhost** as shown in the following screenshot.

A screenshot of a computer

Description automatically generated

Once you’ve created the new Entra Id application, you need to copy and paste the Client Id and Redirect Uri into the two constants named **UserAuthClientId** and **UserAuthRedirectUri** in the **AppSettings** class.

// Public client application created in Entra Id Service for user auth

public const string UserAuthClientId = "22222222-2222-2222-2222-222222222222";

public const string UserAuthRedirectUri = "http://localhost";

Once you have configured **UserAuthClientId** and **UserAuthRedirectUri** in the **AppSettings** class, you can then set the **AuthenticationMode** to **UserAuth** before acquiring an access token.

// set app auth mode

AppSettings.AuthenticationMode = AppAuthenticationMode.UserAuth;

// acquire access token

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

When **AuthenticationMode** is set to **UserAuth,** a call to **GetFabricAccessToken** will use your custom Entra Id application to authenticate the user. The first time you authenticate using this application, you will be prompted by the **Requested permission** dialog due to the Entra Id policy of user consent with delegated access. Once you click the **Accept** button to consent, Entra Id will return an access token at the tail end of the authentication flow.

### Create a Custom Entra Id Application for Service Principal Authentication

The enable service principal authentication, you must create a new Entra Id application and configure it with a client secret. When creating an Entra Id application for a service principal, there is no need to add a Redirect Uri. That’s because Redirect Uris are only used with user authentication.

Once you have created an Entra Id application, you must then update three constants in the **AppSettings** class named **ServicePrincipalAuthTenantId**, **ServicePrincipalAuthClientId** and **ServicePrincipalAuthClientSecret**.

// Condifential client application created in Entra Id Service for service principal auth

public const string ServicePrincipalAuthTenantId = "33333333-3333-3333-3333-333333333333";

public const string ServicePrincipalAuthClientId = "44444444-4444-4444-4444-444444444444";

public const string ServicePrincipalAuthClientSecret = "ADD\_CLIENT\_SECRET\_HERE";

Make sure you configure your service principal with access to the Fabric REST APIs as discussed earlier this chapter in the section titled **Configure Service Principal Access to the Fabric REST API**. If you forget this step, you will still be able to acquire an access token for the service principal. However, all your API calls to the Fabric REST APIs will fail with an Access Denied error.

Once you have configured constants in the **AppSettings** class, you can then set the **AuthenticationMode** to **ServicePrincipalAuth** before acquiring an access token. After that, calling **GetFabricAccessToken** will return an access token with the identity of the service principal.

// set app auth mode

AppSettings.AuthenticationMode = AppAuthenticationMode.ServicePrincipalAuth;

// acquire access token

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

This guidance document will not dive further into the code of the **EntraIdTokenManager** class. That is left as an exercise for any reader interested in learning more about how to implement authentication flows with MSAL.

The **EntraIdTokenManager** class leverages MSAL token caching support where access tokens and refresh tokens for users are cached locally. This token caching support provides the development-time convenience of not having to interactively sign in each time you run a test with the **FabricIsvPlaybook** application. The **FabricIsvPlaybook** application can silently acquire access tokens either from the local token cache or by using a refresh token from the local cache to acquire a new access token from the Entra Id Service without requiring user interaction.

## Transmit an Access Token in a Fabric REST API Call

This chapter has already covered authenticating and acquiring access tokens for the Fabric REST APIs. The chapter will now conclude by showing how to include the access token when executing Fabric REST API calls. Examine the following code which demonstrates a simple “Hello World” example of calling the **List Workspaces** API.

// get access token

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

// parse together Authorization header value

string authorizationHeader = "Bearer " + accessToken;

// create HttpClient and set request headers

HttpClient client = new HttpClient();

client.DefaultRequestHeaders.Add("Authorization", authorizationHeader);

client.DefaultRequestHeaders.Add("Accept", "application/json");

// create REST URI to call List Workspaces API

string restUri = "https://api.fabric.microsoft.com/v1/workspaces";

// execute GET request to call List Workspaces API

HttpResponseMessage response = client.GetAsync(restUri).Result;

The string for the **Authorization** header is parsed together using the word **Bearer** followed by a space and the access token as shown in the following screenshot.

