# Developing Secure Application with Azure AD

If you worked through the Developer Quick Start in the previous chapter, you had the experience of creating and configuring an Azure AD application. After that, you created an ASP.NET MVC application which implemented an authentication flow to acquire an access token which made it possible to call the Power BI Service API and implement Power BI embedding. While creating and successfully testing the **AppOwnsDataApp** is a good start, it only scratched the surface of what you need to know about Azure AD and security programming.

If you’re planning on developing applications that implement Power BI embedding, your understanding of Azure AD is critical. However, gaining a thorough understanding of how to best implement an authentication flow for all the common development scenarios you might encounter is a challenging journey. To that end, this chapter is designed to teach the fundamentals of authentication and authorization with Azure AD. This chapter is written to teach you the theory and terminology while also teaching you the practical side and essential programing techniques. Your journey to master authentication with Azure AD will now begin with a primer on OAuth 2.0 to OpenID Connect.

## Understanding OAuth 2.0 and OpenID Connect

Azure AD implements authentication and authorization services using two key open standards named OAuth 2.0 and OpenID Connect. *OAuth 2.0* is an authorization framework based on distributing access tokens to client applications. *OpenID Connect* is a specification which layers on top of OAuth 2.0 to fill in a few missing pieces with respect to user authentication, identity and security token validation.

The authorization framework of OAuth 2.0 defines four roles in the authorization process including the client, the resource owner, the authorization server and resource servers as shown in Figure 3.1. The *client* is the application you are developing and the *resource owner* is the user who is using your application. The *resource servers* (aka *resources*) represent secured endpoints on the Internet such as the Microsoft Graph API and the Power BI Service API.

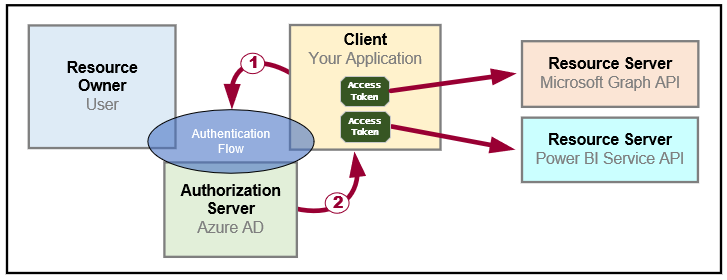


Figure 3.1: OAuth 2.0 provides a framework for distributing access tokens to client applications.

### Access Tokens and Refresh Tokens

The authorization server generates access tokens using a format defined by OAuth 2.0. An access token will always contain several IDs including an ID for the client application and the ID of the authorization server. In many cases, the access token also includes an ID for the user who is acting as the resource owner. However, this is not always the case because you can also implement an authentication flow to acquire an app-only access token as you did in the previous chapter. Since app-only authentication does not involve establishing user identity, the resulting access token will not contain a user ID.

Access tokens can be divided into two different categories. An access token which contains a user identity is known as a *user token*. An access token without a user identity is known as an *app-only token*. As it turns out, implementing authentication flows to acquire user tokens can be far more complicated than implementing an authentication flow to acquire an app-only tokens. Acquiring an app-only token usually involves just a single call from the client application to the authorization server. Acquiring user tokens often involves a sequence of calls back and forth between the client application and the authorization server to provide interactive behavior allowing to user to log in and to consent to the permissions required by the client application.

An access token is often referred to as a *bearer token*. The key point here is that any party that obtains an access token (*i.e. the bearer*) can take advantage of the permissions that have been granted inside. You can make the analogy that an access token is like cash and not like your ATM card which requires you to know a PIN to use it. An attacker who can capture your access token has the ability to use it and to compromise your application's security enforcement policies. Therefore, access tokens should always passed across the network in an encrypted form using SSL and HTTPS and never in clear text with just HTTP.

An access token is created with *expiration* date/time value whichdefines the lifetime for which it is valid. Access tokens are given a relatively short lifetime to decrease the surface area for attackers. Azure AD generates access tokens with a lifetime of 65 minutes. Once an access token expires, any attempt to use it with result in an access denied error.

Now you understand that an access token has a relatively short lifetime which is typically about an hour. This poses a problem of how to manage access token expiration. While a client application could deal with access token expiration by repeatedly executing another authentication flow to acquire a new access token, this would lead to a less-than-ideal user experience as the user would be prompted to login interactively once an hour.

The OAuth 2.0 framework deals with the problem of access token expiration by providing support for *refresh tokens*. When a client application executes an user-based authentication flow, the authorization server passes a refresh token in addition to the access token. While an access token expires in about an hour, a refresh token is valid for a much longer period. The default lifetime for a refresh token created by Azure AD is 14 days and this default configuration value can be extended up to 90 days.

Once the original access token expires, the client application can use the refresh token as a credential when it calls Azure AD to acquire a new access token. You can think of a refresh token as a voucher that the client application uses to obtain a new access token. A key point is that refresh tokens make it possible to keep the lifetime of an access token short while also minimizing the number of times the user must login interactively.

Given that the lifetime of a refresh token extends across many days, it is common for the client application to save refresh tokens to a database or into persistent browser storage so it can be used to acquire new access tokens across user sessions. Consider an example where a user logins into a client application on Monday which executes an authentication flow and saves the resulting refresh token into persistent browser storage. When the user navigates to the same web application later that week, the client application can retrieve the refresh token and use it to silently acquire a fresh access token without requiring the user to login interactively. Once again, the use of refresh tokens is key in OAuth 2.0 because they make it possible to reduce the number of times a user is prompted with an interactive login request.

### Authentication Flows

At the heart of OAuth 2.0 lies the process of transferring an access token from the authorization server to the client application in a safe and secure manner. This process is known as an *authentication flow*. The client application implements an authentication flow to retrieve access tokens from the authorization server. When you develop a client application, you must choose the type of authentication flow depending on upon the details of your development scenario. The authentication flow you implement for a web application will be quite different from the authentication flow you implement for a desktop application.

OAuth 2.0 defines the following types of authentication flows to handle scenarios for web applications, desktop applications, mobile phones, and smart devices.

* **User Password Credential Flow**: used authenticate the user in desktop and mobile applications
* **Client Credentials Flow**: used to authenticate the application with an app-only identity
* **Implicit Flow**: used to authenticate the user in an SPA (i.e. single page application with only client-side code)
* **Authorization Code Flow**: used to authenticate the user in web applications

Once the client application has executed an authentication flow to acquire an access token, the client application then must transmit the access token whenever it executes an HTTP operation against the target resource server. The resource server trusts the authorization server and therefore trusts whatever information it finds inside the access token. This allows the resource server being accessed to inspect each access token to discover what permissions have been granted to the caller.

A key principle in OAuth 2.0 is that of *delegated access* in which an application can make an API call on behalf of a user. This is different from a scenario using impersonation because your application does not make an API call using the identity and permissions of the user. Instead, the client application establishes its own separate identity which is granted a subset of the user's permissions. The key point is that an application using delegated access calls APIs *on behalf of the user* instead of calling *as the user*.

### Open ID Connect and ID Tokens

The OAuth 2.0 framework has a few shortcomings with respect to user authentication and identity. The main problem has to do with verifying the authenticity of an access token. While the OAuth 2.0 framework provides a way for a resource server to verify that an access token is valid, it does not provide the client application with the same ability. When using OAuth 2.0 without OpenID connect, there is no standard way for a client application to verify that an access token was created by a trusted authorization server.

The inability of a client application to validate the authenticity of an access token poses a critical security vulnerability. In the early years when companies were using OAuth 2.0 without OpenID Connect, the software industry saw a number of successful attacks on high-profile web sites such as Facebook. The most common examples have been cross-site request forgery (CSRF) attacks where an attacker replaces the user’s access token with a fraudulent access token associated with the attacker’s private resources. The success of cross-site request forgery (CSRF) attacks against OAuth 2.0 Authorization servers made it clear that the OAuth 2.0 framework needed to be extended to provide client applications with the ability to validate the authenticity of access tokens to against common attacks.

The OAuth 2.0 shortcomings with respect to user authentication and token validation are addressed by OpenID Connect. OpenID Connect defines a token validation scheme where the client application is able to download a set of keys from the authorization server which makes it possible to validate the authenticity of all the security tokens passed to the client application during an authentication flow. This is essential in eliminating vulnerabilities such as CSRF attacks.

In addition to providing the means to validate security tokens, OpenID Connect defines a third type of security token known as an *ID token*. When a client application executes a user-based authentication flow, the authorization server passes back three security tokens including the access token, the refresh token and the ID token as shown in figure 3.2.

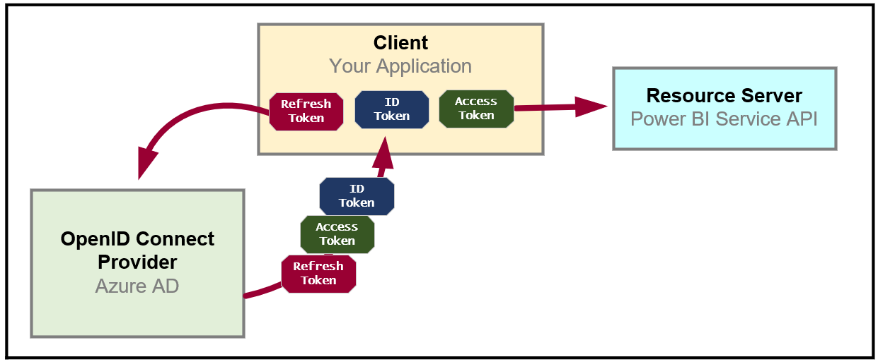


Figure 3.2: OpenID Connect adds support for the ID token as well as token validation.

While it’s important that you understand the role of both OAuth 2.0 and OpenID Connect, it's not as confusing as it sounds. You don't have to think about OAuth 2.0 and OpenID Connect as two separate standards. Instead, OAuth 2.0 and OpenID Connect combine to create a single protocol with one set of rules. When you hear developers talk about OAuth 2.0 with Azure AD, it’s implied that they are talking about *OAuth 2.0* combined with *OpenID Connect*.

### Registering Client Applications

The OAuth 2.0 framework requires you to register the client application with the authorization server before it can be used. When a client application is registered, the authorization server assigns it an ID and a friendly name as shown in Figure 3.3. Depending upon the type of client application you are developing, you might also need to configure the client application with permissions, reply URLs and/or credentials.



Figure 3.3 provide a high-level view of the data tracked by azure AD when you register an application.

The process of registering an application is somewhat analogous to creating a new user account. A new user account in Azure AD is created with a login ID and a set of credentials allowing the user to successfully log in and establish an identity. In a similar fashion, a registered application is created with an application ID and an optional set of credentials which allows the client application to authenticate with the authorization server to establish its identity.

### Delegated Permissions and User Consent

As you have learned, the OAuth 2.0 framework gives a client application the ability to make API calls on behalf of the user with a subset of the user’s permissions. The permissions a client application uses to run on behalf of a user are known as *delegated permissions*. The idea is that users delegate to client applications to accomplish work on their behalf.

An user token contains a set of one ore more delegated permission known as *scopes*. When a resource server responds to an incoming API call, it inspects the access tokens and the delegated permission scopes inside to determine what permissions have been granter to the caller.

Each delegated permission scope is defined as a specific permission associated with a specific resource server. Delegated permission scopes are identified using string-based IDs that includes an identifier for the resource server combined together with an identifier for the permission itself. The Power BI Service API acts as an OAuth 2.0 resource server and is identified using the following string value.

https://analysis.windows.net/powerbi/api

The Power BI Service API defines delegated permission scopes which are identified by adding its resource server ID together with a permission ID. Here are three examples of permission scopes defined by the Power BI Service API.

https://analysis.windows.net/powerbi/api/Dashboard.Read.All

https://analysis.windows.net/powerbi/api/Dataset.Read.All

https://analysis.windows.net/powerbi/api/Report.Read.All

When programming with C#, delegated permission scopes can be combined using a string array. Here is an example of the same three delegated permission scopes combined into a string array named *ReadWorkspaceAssets*.

string[] ReadWorkspaceAssets = new string[] {

"https://analysis.windows.net/powerbi/api/Dashboard.Read.All",

"https://analysis.windows.net/powerbi/api/Dataset.Read.All",

"https://analysis.windows.net/powerbi/api/Report.Read.All"

};

When a client application executes an user-based authentication flow, it must pass a *scope* parameter to the authorization server to indicate which delegated permissions it requires to successfully complete its work. This is where the concept of user consent enters the picture. As long the as the authorization server sees evidence that the user has previously consented to the client application to use the delegated permissions requested, the authorization server will add the requested permission scopes to the access token returned to the client application. But that begs an important question. How does a user first consent to the delegated permissions requested a client application?

An essential aspect of the OAuth 2.0 framework is that it defines the interactive behavior that authorization servers must implement to obtain the user’s consent to whatever delegated permissions are requested by the client application. This interactive behavior is typically seen by users the first time they log into a client application. After a user logs in for the first time with a user name and password, the authorization server prompts the user with a consent screen which describes the requested permissions and provides a button which the user can click to consent. By clicking the consent button, the user effectively grants the client application the delegated permissions it has requested.

The need for a user to interactively consent and grant permissions during login is typically a one-time event. After a user has consented to the requested set of delegated permissions during the first login, the authentication server tracks these permission grants which make it unnecessary to prompt the user for consent in future logins. However, that is not always the case because a client application can be written to use incremental consent.

The concept of *incremental consent* is important because it allows a client application to request only the permissions it needs at the time. Consider a scenario in which a client application requires three standard permissions in order to perform its day-to-day work. However, in rare circumstances the client application also requires a fourth permission which yields administrator-level privileges to ready and write all data at the tenant level.

https://analysis.windows.net/powerbi/api/Tenant.ReadWrite.All

The client application can be written to request just the three standard permissions during the first login. In the less-common scenario when the client application needs to use the fourth permission, it can execute an authentication flow which requests the fourth delegated permission and the user will be automatically prompted to consent to the fourth permission which has not been previously granted.

When you register a client application, you can optionally configure a default set of delegated permissions. Configuring delegated permissions as part of the client application registration is not required because a client application can always explicitly request the exact set of delegated permissions it requires when it executes an authentication flow. However, a client application also has the option of passing a special string identifier to the scope parameter to request the default set of delegated permissions which have been configured as part of the client application registration.

https://analysis.windows.net/powerbi/api/.default

### Redirect URIs

The OAuth 2.0 framework provides support for *redirect URIs* (aka *Reply URLs*) to defend against attacks in user-based authentication flows. You will find that most authentication flows used to acquire an access token on behalf of a user require the use of redirect URIs. Therefore, it’s important that you understand how they’re used and that you appreciate the protection that they provide.

At the end of user-based authentication flow after the user has signed in, the authorization server must redirect the browser to a URL where it can pass sensitive data such as an access token or authorization code back to the client application. If an attacker is able to trick the authorization server into redirecting browser to the URL of a private sever owned by the attacker, the access token could be stolen for malicious purposes. The purpose of a redirect URI is to prevent this type of attack.

When you register a client application, you can configure it with one or more redirect URIs to tell the authorization server where your application is running on the Internet. The key concept is that the authorization server will only pass sensitive data to a URL that has have been registered as redirect URI.

A developer must take two separate actions to successfully execute an authentication flow that requires a redirect URI. First, the developer must configure the redirect URI as part of the client application registration. Second, developer must implement the client application to pass the redirect URL to the authorization server when it executes the authentication flow. An authentication flow will fail if the redirect URI passed by the client application isn’t an exact match to a redirect URI that has been registered with the client application. OAuth 2.0 authorization servers are notorious for failing authentication flows when the redirect URI is slightly different than the registered redirect URI such as the case when one redirect has a trialing backslash but the other does not.

### Public Clients versus Confidential Web Clients

The OAuth 2.0 framework defines two different types of client applications. These are public clients and confidential web clients. The big difference between the two is that confidential web clients can keep a secret while public clients cannot. Let’s look at a simple example to make the distinction between these two types of client applications more clear.

A Web application which contains server-side code and server-side configuration files can hold a credential such an application secret which plays the role of an application’s password during the authentication process. However, this credential is kept a secret because it is never passed to the user device where the client application is running. Therefore, the Web application is considered to be a confidential client.

Now compare the confidential web client application to a native application which has been installed on the user’s device such as a mobile phone or a Windows PC. If installing a native application includes copying credentials such as an application secret to the user’s device, it opens up a significant security hole because the credential is far more vulnerable to being discovered by an attacker. They key principal here is that you cannot keep something a secret once it has been copied to the user’s device. That is why desktop applications and native mobile applications are typically configured as public client applications because they cannot contain application secrets without introducing security vulnerabilities.

So far, this chapter has explained OAuth 2.0 and OpenID Connect in a generic way. All the terms and concepts covered so far will help you as you begin to develop client applications that authenticate users and acquire access tokens using the Internet’s must popular identify providers which use OAuth 2.0 including Google, Twitter, LinkedIn, Facebook, Instagram and Spotify. However, you will find that many details and OAuth 2.0 feature support vary as you move from one identity provider to the next. This chapter will now transition to examining how OAuth 2.0 and OpenID Connect are implement using Azure AD in the Microsoft cloud.

## Understanding the Microsoft Identity Platform (2.0)

Now that you’ve learned the fundamentals of OAuth 2.0 and OpenID Connect, it's time to put all this theory to work and begin discussing how to work with Azure AD to register and develop client applications. As you get started, you should understand that Azure AD currently supports two different implementations of OAuth 2.0 and OpenID Connect. Microsoft’s original of OAuth 2.0 and OpenID Connect implementation is known as the Azure Active Directory developer platform (v1.0). More recently, Microsoft released an improved implementation known as the Microsoft identity platform (v2.0).

The *Azure Active Directory developer platform (v1.0)* has been generally available for over 8 years and it is still heavily used today in production applications. The platform is based on a set of authorization web services provided by Azure AD known as the v1.0 endpoint. Microsoft also provides the *Azure Active Directory Authentication Libraries (ADAL)* which make it easier to write the code to implement authentication flows against the v1.0 endpoint.

The v1.0 endpoint and ADAL pose a few important limitations that motivated Microsoft to create a new v2.0 endpoint and a new set of authentication libraries know as the Microsoft Authentication Libraries (MSAL). The new Azure AD v2.0 endpoint and MSAL are collectively known as the *Microsoft identity platform (v2.0)*.

Microsoft ran a preview program with the v2.0 endpoint and MSAL for over a year in 2018 before releasing them and making then generally available (GA) in May of 2019. Today, Microsoft recommends using the Azure AD v2.0 endpoint and MSAL over using the original Azure AD v1.0 endpoint and ADAL. The main advantages of the Azure AD v2.0 endpoint over the v1.0 endpoint can be summarized in the following list.

* The v2.0 endpoint authentication focuses on scopes in a way that is more compliant with the OAuth 2.0 framework.
* The v2.0 endpoint supports incremental consent while the v1.0 endpoint does not.
* The v2.0 endpoint adds support for new authentication flows such as Device Code Flow.
* The v2.0 endpoint supports user authentication with personal accounts in addition to school and work accounts.

In the short term, you can use either the v1.0 endpoint and ADAL or the v2.0 endpoint and MSAL to develop applications that implement Power BI embedding. However, there are several reasons why you should favor the v2.0 endpoint and MSAL over the v1.0 endpoint and ADAL. First, Microsoft is continuing to invest in the v2.0 endpoint and MSAL to add features and support for new authentication flows to whereas Microsoft has discontinued any new investment in the v1.0 endpoint and ADAL. Second, the v2.0 endpoint and MSAL are better in terms of their design and their compliance with OAuth 2.0 and OpenID Connect. Third, the use of the v1.0 endpoint and ADAL will become deprecated over the next few years and applications that use them will eventually need to be rewritten to use the v2.0 endpoint and MSAL. For these reasons, this document will exclusively focus on the Azure AD v2.0 endpoint and MSAL.

### Creating Azure AD Applications

The way to register a client application with Azure AD is to create a new Azure AD application. For production scenarios you can create new Azure AD applications by running a PowerShell script or by writing code against the Microsoft Graph API. When you are just getting started with Azure AD, the easiest way to create an Azure AD application is to use the Azure portal which is accessible by navigating to *https://portal.azure.com*.

If you navigate to the Azure portal and click the **Azure Active Directory** link in the left navigation, you will see the **App registrations** link as shown in figure 3.4. If you then click the **App registrations** link, you will see the **App registrations** view that allows you to see Azure AD applications that have been created the current Azure AD tenant. The **App registrations** view also provides a **New registration** button which you can click to navigate to the **Register an application** page where you can enter the information create new a new Azure AD application by hand.

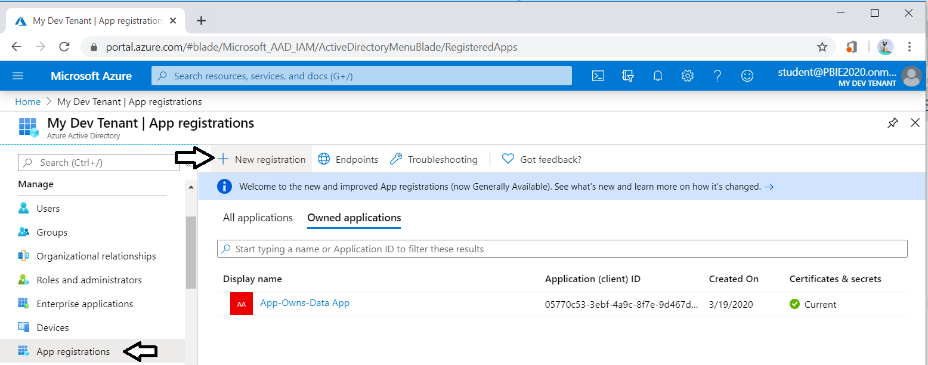


Figure 3.4: The Azure portal makes it possible to view and manage

When creating a new Azure AD application on the **Register an application** page, you begin by entering the application name as shown in figure 3.5. Next you must select an option to determine what types of user accounts are supported for login. The default choice is **Single tenant** which restricts login to user accounts that exist with the same tenant in which the application is registered. Single tenant is the best choice when developing enterprise applications that will only be used by users in a single organization. The second choice it **Multitenant** which allows login by users with work and school accounts that exist in any Azure AD tenant. This choice is often used by independent software vendors (ISVs) when developing applications that need to be accessible to users across all the Azure tenants managed by their customer base. The last choice allows for login by personal accounts as will as school and work accounts.

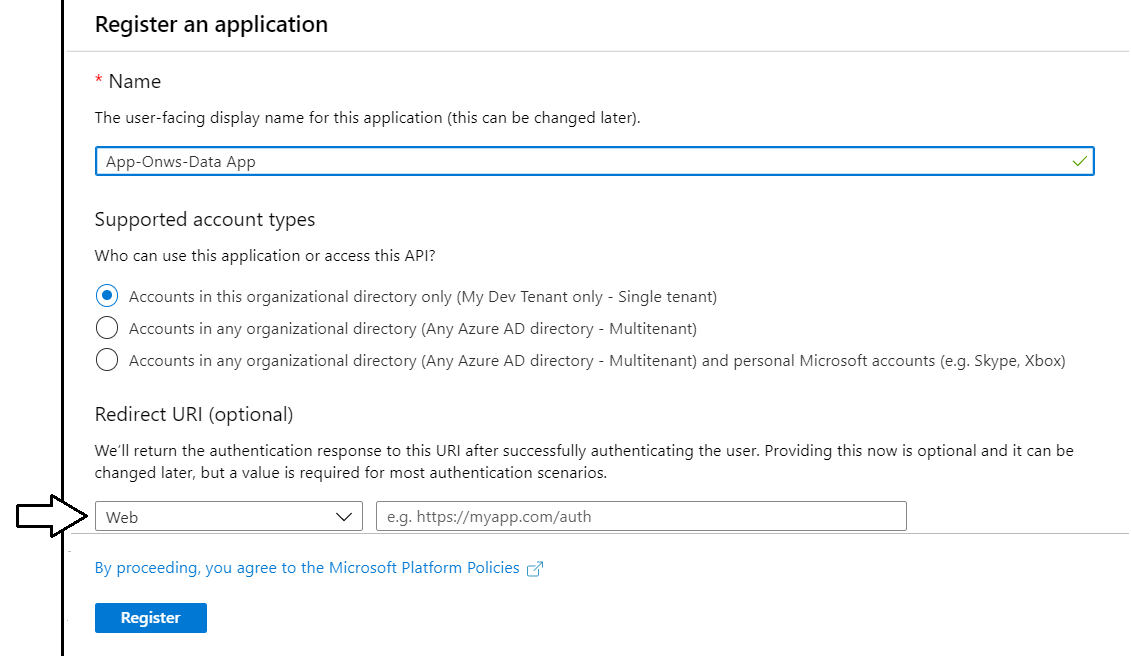


Figure 3.5: When creating an Azure AD application, you must select an application type of Web or Public client.

If you examine the screenshot in figure 3.5, you can see that the **Register an application** page provides a dropdown menu in the **Redirect URI** section with a choice of either **Web** or **Public client/native (mobile and desktop)**. What’s not overly obvious is that this dropdown menu is what you use to make a critical choice between creating a public client application versus creating a confidential client application.

You select **Web** to create an application as a confidential web client or **Public client/native (mobile and desktop)** to create an application as a public client. Throughout this chapter, you will learn what development scenarios call for creating your application as a public client and which ones call for creating your application as a confidential web client.

When you click the **Register** button to create a new application, Azure AD will generate a new GUID to serve as the Application ID as shown in Figure 3.6. The Application ID is important because the client applications you develop will be required to pass this GUID-based ID to Azure AD to identify itself whenever it implements an authentication flow to acquire an access token.

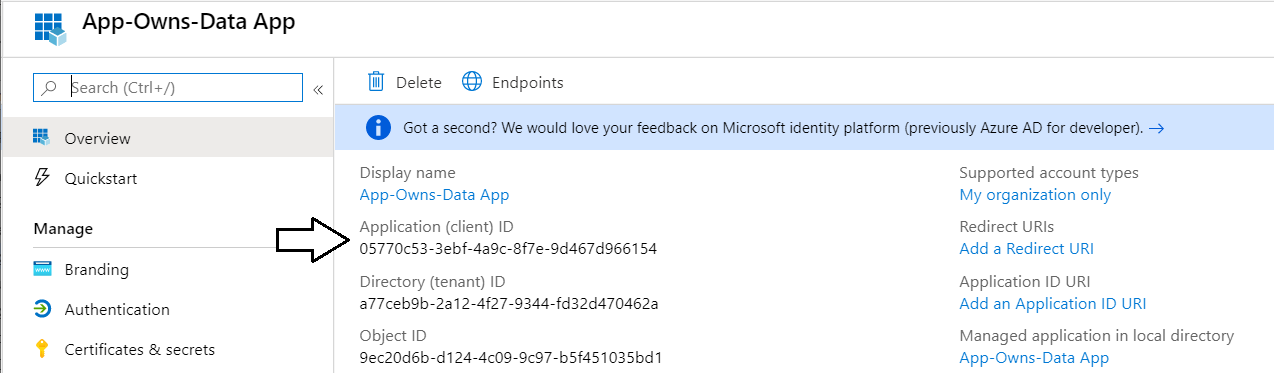


Figure 3.6: Whenever you create a new application, Azure AD will automatically assign it a unique Application ID.

There is an unfortunate terminology issue and potential confusion surrounding the ID for an Azure AD application. Sometimes this GUID is referred to as the **Application ID** while other times it’s called the **Client ID** instead. This might lead you to the obvious questions "what's the difference between an Application ID and a Client ID?". The answer is there is no difference. They're just two different names for the same thing. The world would be a much better place if everyone had agreed to just use one of these names. But sadly, it's too late for that. Think of it as having an uncle named Bob that also goes by the name of Robert.

Once you have initially created a new Azure AD application in the Azure portal, there is often additional configuration that you need to complete before you can actually use the application. The screenshot shown in Figure 3.7 shows some of the links you can use to navigate to the views that make it possible to configure a new Azure AD application with authentication properties, credentials and permissions. You will see several examples of this throughout this section as you begin to configure Azure AD application so that you can implement authentication flows.

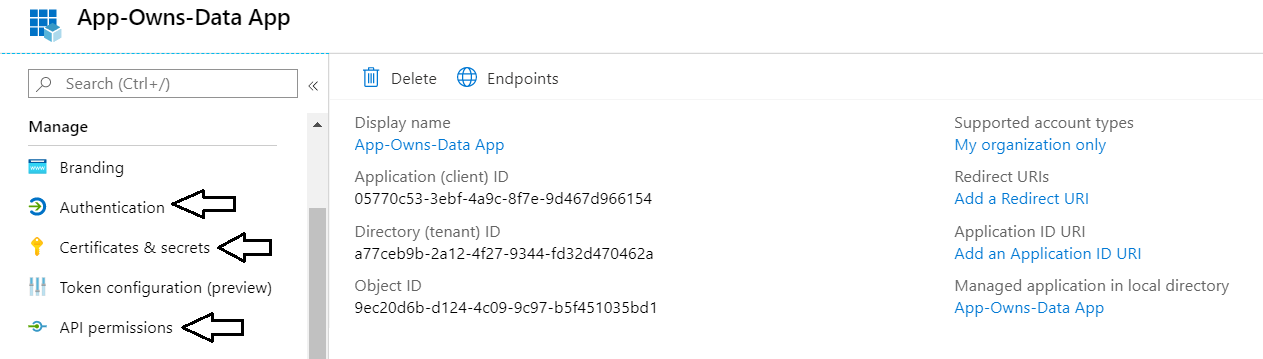


Figure 3.7: The Azure portal provides views to configure Authentication, Certificates & secrets and API permissions.

As you begin to work with Azure AD applications, it's important to differentiate between the application itself and another important object known as the *service principal*. The service principal object acts as the local representation of an Azure AD application within the context of a specific Azure AD tenant. The screenshot in Figure 3.8 shows the summary page for an Azure AD application which displays the GUID for the application ID and other GUID named the Object ID. This Object ID is what is used to identity the service principal.

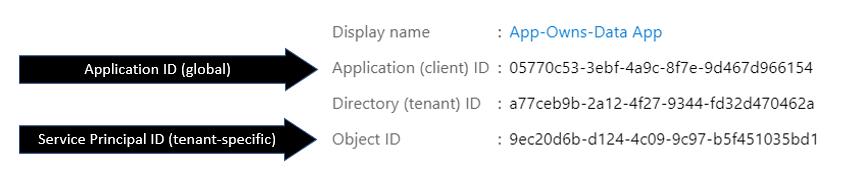


Figure 3.8: The Application ID is a global identifier while the Object ID is a local identifier for the service principal.

Azure AD must track a service principal ID in addition to an application ID due to its support of multitenant applications. Given that a multitenant application can execute within the context of many different tenants, Azure AD requires that an application establishes a separate identity for each tenant in which it runs. The first time a multitenant application runs in the context of a new tenant, Azure AD automatically creates a new service principle object. While all tenants identify the application itself using the global application ID, each tenant gets its own service principle with a unique object ID.

### Creating Applications using PowerShell Scripts

While you can create and configure Azure AD application by hand in the Azure portal, that can become tedious and error prone. As a developer, you should become familiar with PowerShell scripting with the [Azure AD PowerShell module](https://docs.microsoft.com/en-us/powershell/azure/active-directory/install-adv2?view=azureadps-2.0). This PowerShell module provides administrative cmdlets that allow you to create and configure Azure AD applications. If you are running on Windows 10 you can install this essential PowerShell module by running the following PowerShell command from an administrative command prompt.

Install-Module AzureAD

Once you have installed the AzureAD PowerShell module, you can login interactively and create a session by executing the cmdlet named *Connect-AzureAD*.

Connect-AzureAD

If you call *Connect-AzureAD* without passing any parameters, you will be prompted with a browser-based dialog to login using your organizational user account and password. Once you have logged in, you can execute other cmdlets in the AzureAD module to create, configure and view Azure AD applications.

Calling *Connect-AzureAD* without passing any parameters creates an interactive login experience which is great when you do not want to hardcode credentials in a PowerShell script. However, having to enter a user name and password can be tedious when you are constantly running a PowerShell script during the authoring and testing phase. If the situation calls for it, you can hardcode a user name and password into your PowerShell script to avoid having to enter credentials each time you run a script.

$userName = "user1@tenant1.onMicrosoft.com"

$password = "pass@word1"

$securePassword = ConvertTo-SecureString –String $password –AsPlainText -Force

$credential = New-Object –TypeName System.Management.Automation.PSCredential `

–ArgumentList $userName, $securePassword

Connect-AzureAD -Credential $credential

You can create a new Azure AD application with PowerShell by executing the cmdlet named *New-AzureADApplication*. Here is a simple example of calling *New-AzureADApplication* with a minimal set of parameters to create a new Azure AD application as a public client.

New-AzureADApplication `

-DisplayName "My First Native App" `

-PublicClient $true `

-AvailableToOtherTenants $false `

-ReplyUrls @("https://localhost/app1234")

There are many different parameters you can pass when calling *New-AzureADApplication*. The example you just saw involved passing a minimal set of four parameters named *DisplayName*, *PublicClient*, *AvailableToOtherTenants* and *ReplyUrls*. Depending on the type of authentication flow you are implementing, you might need to pass other parameters as well.

The *DisplayName* parameter is used to provide the text for the application's friendly name. The *PublicClient* parameter is used to indicate whether you want to create the application as a public client versus a confidential client. You can create a public client by passing a value of *$true* for the *PublicClient* parameter. You pass a value of *$false* to create a new Azure AD application as a confidential client.

The *AvailableToOtherTenants* parameter is used to indicate whether you are creating a single-tenant application or a multitenant application. If you pass a value of $false to the *AvailableToOtherTenants* parameter, you will create a single-tenant application that is only accessible to users in the same tenant where the application was created. The use of single-tenant applications is common in enterprise development scenarios where the application only supports users inside a single organization.

If you pass a value of $true to the *AvailableToOtherTenants* parameter, you will create a multitenant application that is accessible to users in other Azure AD tenants. The use of multitenant applications is common among ISVs because they can create a single application that can be used across multiple customers that all have their own Azure AD tenants.

Keep in mind that working with multitenant applications introduces complexity into the way you configure Azure AD applications as well as the way you write the code to authenticate users. Therefore, you should always work with single-tenant applications unless you really need multitenant support.

The *New-AzureADApplication* cmdlet accepts a *ReplyUrls* parameter which allows you to configure a new Azure AD application with one or more redirect URIs. For example, the redirect URI for a production application could be *https://myAzureWebApp.azurewebsites.net*. The redirect URI for an application you are currently testing and debugging in Visual Studio could be *https://localhost:44300*. Remember that you are not restricted to one redirect URI. You can configure an application with more than one redirect URI in scenarios where it makes sense.

In the case of a desktop application running as a public client, your application might require a redirect URI even though it’s not actually running at an endpoint on the Internet. In this type of scenario, the redirect URI registered for a public client application just needs to be a string value formatted as a URI such as [*https://localhost/app1234*](https://localhost/app1234). While the redirect URI for a desktop application for a public client cannot be verified in the same fashion as the redirect URI for a web application running on the Internet, it still serves a purpose during an authentication flow.

When a public client application authenticates using an interactive login, it must pass a redirect URI that matches one of the redirect URIs registered with the application. Azure AD will return an access denied error if you pass a redirect URI that does not match one of the redirect URIs that has been registered with the application. Remember that Azure AD will return access denied errors in cases where the reply URL matching fails due to case sensitivity or a missing backslash.

Note that you cannot pass an application ID when creating a new Azure AD application. Instead, Azure AD always takes on the responsibility of generating a new GUID for the application ID whenever a new application is crreated. When you call the *New-AzureADApplication* cmdlet, it returns an object that represents the new Azure AD application. This application object provides many properties including an *AppId* property which you can read to discover the application ID for a new Azure AD application that you have just created.

After creating an new application with *New-AzureADApplication*, you can create the application's service principal by calling *New-AzureADServicePrincipal*. When you call *New-AzureADServicePrincipal*, you must pass the application ID as shown in the following PowerShell script. Note that the PowerShell script also assigns ownership of the Azure AD application to the logged on user.

# log in user and capture authentication result

$authResult = Connect-AzureAD

# get more info about the logged in user

$user = Get-AzureADUser -ObjectId $authResult.Account.Id

# create Azure AD Application

$aadApplication = New-AzureADApplication `

-DisplayName "My First Native App" `

-PublicClient $true `

-AvailableToOtherTenants $false `

-ReplyUrls @("https://localhost/app1234")

# retrieve Application ID for the new Azure AD application

$appId = $aadApplication.AppId

# create service principal for application

$serviceServicePrincipal = New-AzureADServicePrincipal -AppId $appId

# assign current user as application owner

Add-AzureADApplicationOwner -ObjectId $aadApplication.ObjectId -RefObjectId $user.ObjectId

### The Microsoft Authentication Libraries (MSAL)

It's possible to implement an authentication flow without any assistance from an external library. After all, an authentication flow is just a standardized sequence of HTTP requests sent back and forth between the client application and the Azure AD v2.0 endpoint. As long as your programming language and development platform support sending HTTP requests and handling HTTP responses, you can write all the code that's required to acquire access tokens from Azure AD. But just because you can do something (*that’s painful and unnecessary*) doesn't mean you should.

Microsoft provides the *Microsoft Authentication Libraries (MSAL)* to help developers meet the requirements of implementing authentication flows using the Azure AD v2.0 endpoint. There is one version of MSAL for .NET developers (*MSAL.NET*) which can be used to implement authentication flows with managed languages such as C#. There is a second version of MSAL for JavaScript (MSAL.JS) used to implement implicit authentication flows in single page applications (SPAs) created with JavaScript frameworks such as React.js and AngularJS. There are several other versions of MSAL for other platforms as well as such iOS, Android, Java and Python.

To add MSAL.NET to a Visual Studio project, install the NuGet package *Microsoft.Identity.Client*. This NuGet package adds the MSAL.NET library to assist you with implementing authentication flows and acquiring access tokens. There is also a GitHub repository which contains the source code for MSAL.NET along with a few other valuable developer resources which is accessible through the following URL.

<https://github.com/AzureAD/microsoft-authentication-library-for-dotnet>

To use MSAL.JS you must include a script link to main JavaScript library file named *msal.js*. You can obtain a copy of msal.js from the GitHub repository where Microsoft maintains the source code, distribution files and documentation for this library. You can browse to this GitHub repository using the following URL.

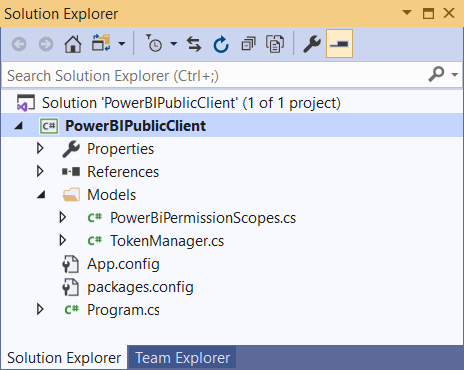
<https://github.com/AzureAD/microsoft-authentication-library-for-js>

MSAL adds value to the development process by abstracting away many of the low-level details required to implement authentication flows with Azure AD. When you're programming with MSAL, you don't have to worry about explicitly sending HTTP requests to the Azure AD v2.0 endpoint or parsing the HTTP response to extract the access token. MSAL does that for you behind the scenes. If that isn't enough, MSAL provides additional value by adding support for token caching and for transparently using refresh tokens to acquire new access tokens as cached access tokens expire.

Now that you’ve learned the concepts and terminology associated with Azure AD development, it's now time to walk through the process of creating a client application with Visual Studio to demonstrate the C# code required to implement an Azure AD authentication flow using MSAL. By the end of this chapter, you have will see several different types of client applications and, along the way, you will learn how to implement the most common authentication flows used in public client application and in confidential client applications.

## Developing Public Clients

Let’s begin by examining the .NET Framework C# console application named **PowerBiPublicClient**. This project has been provided in the **Demos** folder of the GitHub repository associated with this whitepaper. Once you download the source files **Demos** folder, you can open the **PowerBiPublicClient** project using any edition of Visual Studio 2019 or Visual Studio 2017. The basic project structure is shown in the following screenshot.



The *PowerBiPublicClient* project contains three C# source files named *PowerBiPermissionScopes.cs*, *TokenManager.cs* and *Program.cs*. The project also contains an application configuration file named *App.config*. This project has also been configured with the required .NET libraries by installing the following 3 NuGet packages.

* **Microsoft.Identity.Client**: Microsoft Authentication Library (MSAL.NET)
* **Microsoft.PowerBI.Api**: Power BI .NET SDK
* **System.Configuration.ConfigurationManager**: Package used to read configuration values from *App.config*.

If you open the *App.config* file you can see that there are pre-defined *add* elements in the *appSettings* section named *application-id*, *redirect-uri*, *aad-account-name* and *aad-account-password*. In order to test the PowerBiPublicClient application, you must first register it with Azure AD by creating a new Azure AD application. Once you have created the Azure AD application, you must record the application ID and the redirected URI so you can update the appSetting values in *App.config*.

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<appSettings>

<!-- Add client-id and redirect-uri url for this application here -->

<add key="application-id" value="ADD\_APPLICATION\_ID\_HERE" />

<add key="redirect-uri" value="ADD\_REDIRECT\_URI\_HERE" />

<!-- add Azure AD account name and password for account with Power BI Pro license -->

<!-- if you leave these settings blank, user will be presented with interactive login -->

<add key="aad-account-name" value="ADD\_USER\_NAME\_HERE" />

<add key="aad-account-password" value="ADD\_USER\_PASSWORD\_HERE" />

</appSettings>

</configuration>

You will note that the **Demos** folder contains a PowerShell script named *RegisterPowerBiPublicClient.ps1*. This script has been written to create the Azure AD application that us used by the PowerBiPibcClient project. Here is the script content.

# authenticate with Azure AD

$authResult = Connect-AzureAD

# get user account ID for logged in user

$user = Get-AzureADUser -ObjectId $authResult.Account.Id

# add variables to configure new applicaiton name and redirect URI

$appDisplayName = "Power BI Public Client"

$redirectUri = "http://localhost/app1234"

# create a new Azure AD application

$aadApplication = New-AzureADApplication `

-DisplayName $appDisplayName `

-PublicClient $true `

-AvailableToOtherTenants $false `

-ReplyUrls @($redirectUri)

# retrieve application Id

$appId = $aadApplication.AppId

# create service principal for application

$serviceServicePrincipal = New-AzureADServicePrincipal -AppId $appId

# assign current user as application owner

Add-AzureADApplicationOwner -ObjectId $aadApplication.ObjectId -RefObjectId $user.ObjectId

# generate text file that contains application Id and redirect URI

$outputFile = "$PSScriptRoot\PowerBiPublicClient.txt"

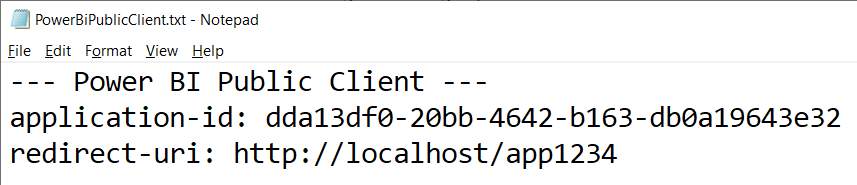
Out-File -FilePath $outputFile -InputObject "--- Power BI Public Client ---"

Out-File -FilePath $outputFile -Append -InputObject "application-id: $appId"

Out-File -FilePath $outputFile -Append -InputObject "redirect-uri: $replyUrl"

Notepad $outputFile

When you run the PowerShell script named *RegisterPowerBiPublicClient.ps1*, it will create a new Azure AD application as a public client application with a redirect URI of *http://localhost/app1234*. There’s also code at the bottom of the script that will launch *Notepad.exe* with a text file named *PowerBiPublicClient.txt* containing the application Id and redirect URI.



After running *RegisterPowerBiPublicClient.ps1*, you can copy and paste the values from *PowerBiPublicClient.txt* to update the *appSetting* values in *App.config* named *application-id* and *redirect-uri*. Save your changes to *App.config*.

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<appSettings>

<!-- Add client-id and redirect-uri url for this application here -->

<add key="application-id" value="dda13df0-20bb-4642-b163-db0a19643e32" />

<add key="redirect-uri" value="http://localhost/app1234" />

<!-- add Azure AD account name and password for account with Power BI Pro license -->

<!-- if you leave these settings blank, user will be presented with interactive login -->

<add key="aad-account-name" value="ADD\_USER\_NAME\_HERE" />

<add key="aad-account-password" value="ADD\_USER\_PASSWORD\_HERE" />

</appSettings>

</configuration>

At this point, you have created a new Azure AD application and you have also updated the PowerBiPublicClient project by adding the application Id and redirect URI values into App.config. Now we will walk through a little more C# code before testing the application.

Let’s begin by examining the C# source files named *PowerBiPermissionScopes.cs*. The files a class named *PowerBiPermissionScopes* with public static fields which hold string arrays. Each of these public static fields tracks a set of delegated permissions for the Power BI Service API.

namespace PowerBIPublicClient.Models {

class PowerBiPermissionScopes {

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

"https://analysis.windows.net/powerbi/api/.default"

};

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

"https://analysis.windows.net/powerbi/api/Group.Read.All"

};

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

"https://analysis.windows.net/powerbi/api/Group.Read.All",

"https://analysis.windows.net/powerbi/api/Dashboard.Read.All",

"https://analysis.windows.net/powerbi/api/Report.Read.All",

"https://analysis.windows.net/powerbi/api/Dataset.Read.All",

"https://analysis.windows.net/powerbi/api/Dataflow.Read.All"

};

// other scopes omitted for brevity

}

}

When it’s time to implement an authentication flow, you can access the public fields in the *PowerBiPermissionScopes* class by name to obtain the set of requested permissions that will be passed to the scope parameter.

string[] scopes = PowerBiPermissionScopes.ReadWorkspaceAssets;

Once you have examined the *PowerBiPermissionScopes* class, open up the other C# source file in the **Models** folder named *TokenManager.cs*. This source file contains the *TokenManager* class which provides several examples of implementing an authentication flow in a public client application using MSAL.

There are two constants named *urlPowerBiServiceApiRoot* and *tenantCommonAuthority*. The const named *urlPowerBiServiceApiRoot* references the root URL of the Power BI Service API for the Power BI public cloud which has a value of *https://api.powerbi.com/*. Note that that this root URL is different when developing against the US Government cloud or one of the Sovereign clouds for other countries such as Germany or China. The *tenantCommonAuthority* const will be used to provide a generic parameter for the target Azure AD Tenant.

namespace PowerBIPublicClient.Models {

class TokenManager {

private const string urlPowerBiServiceApiRoot = "https://api.powerbi.com/";

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

private static string applicationId = ConfigurationManager.AppSettings["application-id"];

private static string redirectUri = ConfigurationManager.AppSettings["redirect-uri"];

private static string userName = ConfigurationManager.AppSettings["aad-account-name"];

private static string userPassword = ConfigurationManager.AppSettings["aad-account-password"];

public static string GetAccessTokenInteractive(string[] scopes) {}

public static PowerBIClient GetPowerBiClientInteractive(string[] scopes) {}

public static string GetAccessTokenWithUserPassword(string[] scopes) {}

public static PowerBIClient GetPowerBiClientWithUserPassword(string[] scopes) {}

public static string GetAccessToken(string[] scopes) {}

public static PowerBIClient GetPowerBiClient(string[] scopes) {}

}

}

The TokenManager class contains private static fields that are initialized with values from the appSetting values stored in App.config. Therefore, these appSetting values that track an application ID and redirect URI are available to any method implementation inside the TokenManager class.

### Programming an Interactive Authentication Flow

Let’s begin by examining the *GetAccessTokenInteractive* method of the *TokenMaster* class. The method has been designed to accept a scopes **parameter** and return an access token as a string. This method follows the common MSAL pattern of creating an application builder object and using the builder to create an application object. In this case, a public application builder is used to create a new application which is initialized with the common organizations authority URL and a redirect URL. After creating the application by calling the **Build** method, you can begin an interactive authentication flow by calling the *AcquireTokenInteractive* method of the application object. If the call to *AcquireTokenInteractive*is successful, it returns an MSAL AuthenticationResult object which exposes the *AccessToken* as a string property.

public static string GetAccessTokenInteractive(string[] scopes) {

// create new public client application

var appPublic = PublicClientApplicationBuilder.Create(applicationId)

.WithAuthority(tenantCommonAuthority)

.WithRedirectUri(redirectUri)

.Build();

// execute interactive authentication flow

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

// return access token to caller

return authResult.AccessToken;

}

When writing code against the TokenManager class, you can acquire an access token by calling GetAccessTokenInteractive and passing a named permission set to the **scopes** argument.

string accessToken = TokenManager.GetAccessTokenInteractive(PowerBiPermissionScopes.ReadWorkspaceAssets);

When using the Power BI .NET SDK to call the Power BI Service API , you must begin by creating a PowerBIClient client object which is initialized using an access token and the root URL to the Power BI Service API.

// acquire access token

string accessToken = TokenManager.GetAccessTokenInteractive(PowerBiPermissionScopes.ReadWorkspaceAssets);

// add access token to new TokenCredentials instance

TokenCredentials tokenCredentials = new TokenCredentials(accessToken, "Bearer");

// create Uri object that points Power BI API root

Uri uriPowerBiServiceApiRoot = new Uri(urlPowerBiServiceApiRoot);

// create PowerBiClient object

PowerBIClient pbiClient = new PowerBIClient(uriPowerBiServiceApiRoot, tokenCredentials);

The *TokenManager* class exposed another method named *GetPowerBiClientInteractive* which creates an access token by calling GetAccessTokenInteractive and then returns an PowerBIClient instance back to the caller.

public static PowerBIClient GetPowerBiClientInteractive(string[] scopes) {

var tokenCredentials = new TokenCredentials(GetAccessTokenInteractive(scopes), "Bearer");

return new PowerBIClient(new Uri(urlPowerBiRestApiRoot), tokenCredentials);

}

Now run this client code.

public static void DisplayUserWorkspaces() {

var requestedScopes = PowerBiPermissionScopes.ReadUserWorkspaces;

PowerBIClient pbiClient = TokenManager.GetPowerBiClient(requestedScopes);

var workspaces = pbiClient.Groups.GetGroups().Value;

foreach (var workspace in workspaces) {

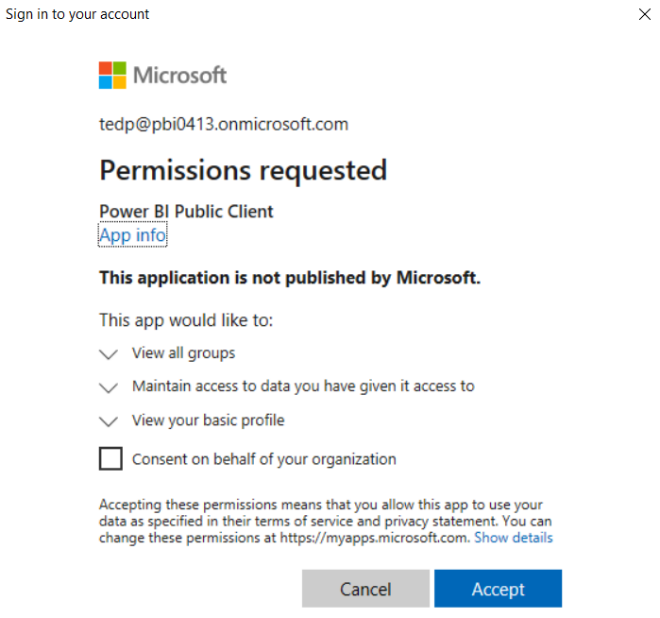
Console.WriteLine($" - {workspace.Name} [{workspace.Id.ToString()}]");

}

Console.WriteLine();

}

You have just seen a complete C# program that acquires an access token from Azure AD and uses it to call the Power BI Service API to retrieve data about the reports in the user's personal workspace. Now that you have seen a complete walkthrough of the code, let's discuss what is really going on behind the scenes. ADAL provided an implementation of the authorization code flow using the following steps.



All is good. Now try running again. It should not prompt for consent.

Ask for more permissions.

public static void DisplayReportsPersonalWorkspace() {

var requestedScopes = PowerBiPermissionScopes.ReadWorkspaceAssets;

PowerBIClient pbiClient = TokenManager.GetPowerBiClient(requestedScopes);

var reports = pbiClient.Reports.GetReports().Value;

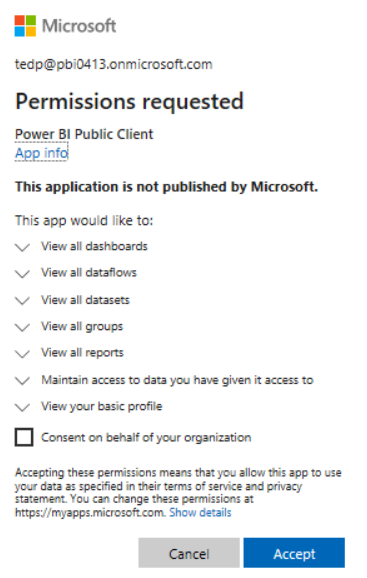
foreach (var report in reports) {

Console.WriteLine($" - {report.Name} [{report.Id.ToString()}]");

}

}

More text here



More text here.

What’s going on behind the scenes.

1. MSAL opens a browser and redirects the user to the Azure AD authorization endpoint begin the authorization code flow.
2. Azure AD interacts with the user using the standard login prompt where the user can enter login credentials.
3. The users completes the login process by entering credentials and consenting to delegated permissions.
4. Azure AD returns an authorization code back to your application.
5. MSAL calls to the Azure AD token endpoint passing the authorization code to obtain an access token.

While this is an example of the authorization code flow, it's a specializes version for a native client. Later in the section *Programming the Authorization Code Flow in a Web App*, you will learn about the authorization code flow in greater detail. But for now you can observe how much work ADAL does for you behind the scenes. All you were required to do was to call *AcquireTokenAsync*. ADAL does all the work to implement an authentication flow that involves interactive behavior and several roundtrips between your application and Azure AD.

### Programming User Password Credential Flow in a Native Application

In the previous example, the application leveraged MSAL functionality to provide interactive behavior which prompts the user to login when the application starts. During development, you might find it helpful to hardcode the user name and password into your code so it runs without prompting. You can accomplish by creating a *UserPasswordCredential* object that is initialized with an Azure AD user account login and a password. You then pass the *UserPasswordCredential* object when you call *AcquireTokenByUsernamePassword*.

private static string clientId = ConfigurationManager.AppSettings["client-id"];

private static string userName = ConfigurationManager.AppSettings["aad-account-name"];

private static string userPassword = ConfigurationManager.AppSettings["aad-account-password"];

public static string GetAccessTokenWithUserPassword(string[] scopes) {

// create new authentication context

var appPublic = PublicClientApplicationBuilder.Create(clientId)

.WithAuthority(tenantCommonAuthority)

.Build();

SecureString userPasswordSecure = new System.Security.SecureString();

foreach (char c in userPassword) {

userPasswordSecure.AppendChar(c);

}

AuthenticationResult authResult =

appPublic.AcquireTokenByUsernamePassword(scopes,

userName,

userPasswordSecure).ExecuteAsync().Result;

// return access token to caller

return authResult.AccessToken;

}

public static PowerBIClient GetPowerBiClientWithUserPassword(string[] scopes) {

var tokenCredentials = new TokenCredentials(GetAccessTokenWithUserPassword(scopes), "Bearer");

return new PowerBIClient(new Uri(urlPowerBiRestApiRoot), tokenCredentials);

}

When you run the program with *GetAccessTokenWithUserPassword*, you will find that the program runs without requiring any interaction on the part of the user. You might have also noticed that you are not required to pass a reply URL when you use the user password credential flow. The way the application interacts with Azure AD is also quite different because ADAL implements the user password credential flow by making a single call to the Azure AD token endpoint which involves passing the user name and password across the network. While the user password credential flow is easy to program, it is considered to be the least secure of the Azure AD authentication flows due to passing a password across the network.

Another issue to be aware of when using the user password credential flow is that it does not provide any ability to provide interactive behavior. If you attempt to acquire an access token using the user password credential flow with a user who has not yet consented to the application's delegated permissions, the call to *AcquireAccessToken* will fail. Remember that you can work around this problem by navigating the *Required permissions* blade for the application in the Azure portal and clicking the *Grant permissions* button.

While the user password credential flow is less secure that other authentication flows, you will find that in certain development scenarios you are required to use it. One noteworthy scenario in which developers have been required to authenticate using the user password credential flow has been when developing with Power BI embedding using third-party embedding and the app-own-data model. Let's examine why.

When the Power BI Service API was first introduced, it did not provide support for application permissions or app-only access tokens. Instead, it only support delegated permissions and user-specific access tokens. With these limitations, the Microsoft recommendation for implementing third-party embedding and the app-owns-data model included the following.

1. Create an Azure AD user account in the same tenant to serve as a master user account
2. Assign a Power BI Pro license to the master user account.
3. Configure the master user account as the administrator for any app workspace it needs to access
4. Authenticate the master user account and acquire access tokens using the user password credential flow

Here is the important takeaway. When you develop with third-party embedding using this security model, your application does not access the Power BI Service API under the identity of the current user nor under the identity of the application itself. Instead, your application accesses the Power BI Service API on behalf of the master user account and it relies on delegated permissions which must have already been granted. Plenty of developers have run into the issue where the user password credential flow fails because it cannot provide interactive behavior for the user to consent the required permissions.

There is good news for companies developing with third-party embedding and the app-owns-data model. Microsoft is introducing new support that allows you to call into the Power BI Service API using app-only access tokens. This means your custom application can use the client credentials flow to establish an app-only identity that doesn't involve any user account which provides two big benefits. First, your application can now rely on application permissions instead of delegated permissions. Second, it eliminates the problem of provisioning and licensing a master user account. We will examine authenticating for third-party embedding in depth in the section titled *Programming the Client Credentials Flow in a Web App*.

### Adding MSAL Support for Token Caching

Ssssss

static class TokenCacheHelper {

private static readonly string CacheFilePath = Assembly.GetExecutingAssembly().Location + ".msalcache.bin3";

private static readonly object FileLock = new object();

public static void EnableSerialization(ITokenCache tokenCache) {

tokenCache.SetBeforeAccess(BeforeAccessNotification);

tokenCache.SetAfterAccess(AfterAccessNotification);

}

private static void BeforeAccessNotification(TokenCacheNotificationArgs args) {

lock (FileLock) {

// repopulate token cache from persisted store

args.TokenCache.DeserializeMsalV3(File.Exists(CacheFilePath) ? File.ReadAllBytes(CacheFilePath) : null);

}

}

private static void AfterAccessNotification(TokenCacheNotificationArgs args) {

// if the access operation resulted in a cache update

if (args.HasStateChanged) {

lock (FileLock) {

// write token cache changes to persistent store

File.WriteAllBytes(CacheFilePath, args.TokenCache.SerializeMsalV3());

}

}

}

}

Ssssss

var appPublic = PublicClientApplicationBuilder.Create(clientId)

.WithAuthority(tenantCommonAuthority)

.WithRedirectUri(redirectUri)

.Build();

TokenCacheHelper.EnableSerialization(appPublic.UserTokenCache);

Ssss

public static string GetAccessToken(string[] scopes) {

// create new authentication context

var appPublic = PublicClientApplicationBuilder.Create(applicationId)

.WithAuthority(tenantCommonAuthority)

.WithRedirectUri(redirectUri)

.Build();

// plug-in serialization support to enable token caching

TokenCacheHelper.EnableSerialization(appPublic.UserTokenCache);

AuthenticationResult authResult;

try {

// try to acquire token from token cache

authResult = appPublic.AcquireTokenSilent(scopes, userName).ExecuteAsync().Result;

}

catch {

try {

// try to acquire token with non-interactive User Password Credential Flow

SecureString userPasswordSecure = new System.Security.SecureString();

foreach (char c in userPassword) {

userPasswordSecure.AppendChar(c);

}

authResult = appPublic.AcquireTokenByUsernamePassword(scopes,

userName,

userPasswordSecure).ExecuteAsync().Result;

}

catch {

// try to acquire token with interactive flow as the last option

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

}

}

// return access token to caller

return authResult.AccessToken;

}

xx

## Developing Confidential Web Clients

### Programming the Client Credentials Flow for the App-Owns-Data Model

The *Client Credentials Grant Flow* is used in a web app to authenticate the application itself and to establish an application identity which has no associated user identity. This authentication flow is used when an application needs to take advantage of application permissions. However, the *Client Credentials Grant Flow* isn’t relevant to Power BI embedding because the Power BI Service API does not currently support any application permissions.

### Programming Implicit Flow in a Single Page Application (SPA)

Azure AD maintains an application manifest for every registered applications.



Figure 3.6: The Azure portal makes it possible to view and, if necessary, edit the application manifest.

### Programming Authorization Code Flow in an ASP.NET MVC Web Application

In an earlier section you saw that ADAL can provide an implementation of the authorization code flow in a native client. When you use this flow in a native client by calling *AcquireTokenAsync*, ADAL prompts the user with a dialog with an embedded browser to provide an interactive login experience. However, the ADAL implementation of the authorization code flow in a native client cuts a few corners and does not meet the requirements of OpenID connect. In order to implement the authorization code flow the right way, you must create an Azure AD application as a Web app / API instead of as a native client.

The OAuth 2.0 framework differentiates between confidential clients and public clients. A *confidential client* is an application that contain credentials such as a password or certificate file without the risk of exposing this sensitive data to a potential attacker. A *public client* is the opposite because it cannot protect sensitive data. A public client is used in scenarios where an application is running on a client device or running as an single page application within a browser where an attacker can see all the data used by the application. The key point here is that an application must be a confidential client to implement the authorization code flow in a secure manner.

Another import change is that the application must be running at an HTTPS endpoint that is registered as a reply URL. This adds an important security dimension because Azure AD will only return an access token when it sees that the application is running at a network endpoint that is registered as a reply URL. This cuts down the attack surface.

Here is the high-level overview of the authorization code flow.

1. The application redirects the user to the authorization endpoint.to start flow
2. User enter credentials and (if required) consents to required permissions
3. Azure AD send POST to application with authorization code.
4. Application passes authorization code and application secret to token endpoint to acquire an access token.

Key point

1. The application never sees the user's password.
2. The authentication flow validate both the user identity and the application identity.
3. Access token is acquired in server-to-server call so never passes through browser or client device.

You need more than just ADAL to implement the authorization code flow. If you are developing with ASP.NET MVC, the most common approach is to combine ADAL together with the OWEN framework and a set of OWEN middleware components provide by Microsoft.

What is OWEN? 1 paragraph.

What does OWEN add

1. It know how to redirect to authorization endpoint.
2. It provide listening mechanism to handle POST callback from Azure AD with authorization code.
3. After the end of the authentication process, OWEN middle populates the ASP.NET principal object
4. Allow you to use Authorization attribute

Here are the NuGet packages

1. Microsoft.Owin
2. Microsoft.Owin.Host.SystemWeb
3. Microsoft.Owin.Security
4. Microsoft.Owin.Security.Cookies
5. Microsoft.Owin.Security.OpenIdConnect

More

public partial class Startup {

private static string commonAuthority = " https://login.microsoftonline.com/common/";

private static string clientId = ConfigurationManager.AppSettings["client-id"];

private static string replyUrl = ConfigurationManager.AppSettings["reply-url"];

public void ConfigureAuth(IAppBuilder app) {

app.SetDefaultSignInAsAuthenticationType(CookieAuthenticationDefaults.AuthenticationType);

app.UseCookieAuthentication(new CookieAuthenticationOptions());

app.UseOpenIdConnectAuthentication(

new OpenIdConnectAuthenticationOptions {

ClientId = clientId,

Authority = commonAuthority,

TokenValidationParameters = new TokenValidationParameters { ValidateIssuer = false },

PostLogoutRedirectUri = replyUrl,

Notifications = new OpenIdConnectAuthenticationNotifications() {

AuthorizationCodeReceived = (context) => {

// code to authenticate and acquire access token

}

});

}

}

And now you add a controller class named AccountControl.

more

using System.Web;

using System.Web.Mvc;

using Microsoft.Owin.Security.Cookies;

using Microsoft.Owin.Security.OpenIdConnect;

using Microsoft.Owin.Security;

namespace DailyReporterPersonal.Controllers {

public class AccountController : Controller {

public void SignIn() {

if (!Request.IsAuthenticated) {

HttpContext.GetOwinContext().Authentication.Challenge(

new AuthenticationProperties { RedirectUri = "/" },

OpenIdConnectAuthenticationDefaults.AuthenticationType);

}

}

public void SignOut() {

string callbackUrl = Url.Action("SignOutCallback", "Account",

routeValues: null, protocol: Request.Url.Scheme);

HttpContext.GetOwinContext().Authentication.SignOut(

new AuthenticationProperties { RedirectUri = callbackUrl },

OpenIdConnectAuthenticationDefaults.AuthenticationType,

CookieAuthenticationDefaults.AuthenticationType);

}

public ActionResult SignOutCallback() {

if (Request.IsAuthenticated) {

return RedirectToAction("Index", "Home");

}

return View();

}

}

}

When you first acquire an access token suing ADAL, this library provides built-in code which inserts the access token along with a refresh token into a cache. After that, you can call ADAL methods such as AcquireAccessTokenSilent to retrieve an access token from the cache. If there isn't a valid access token in the cache, ADAL will use the refresh token to acquire a new access token from Azure AD. All this work of caching and refreshing expired access tokens takes place behind the scenes and is transparent to your code.

There is good news here. While you now understand what refresh tokens are and how they work, this is something you only need to understand in theory, but not in practice. When you begin to program with the Azure Active Directory Library (ADAL), you will happily discover that this library abstracts away any need for a developer to write any code that directly deals with refresh tokens. In fact, any code that uses ADAL and is working directly with refresh tokens is likely not using the library as it was intended.

The next two authentication flows are used in a Web app to authenticate the user and to establish user identity. *Authorization Code Grant Flow* is more secure because it requires application to provide a client secret during the authentication process just after requiring the user to provide a secret password. *Implicit Grant Flow* is used by client-side Web applications such as single page applications (SPAs) which run entirely within the browser and cannot keep any hidden secrets. The implicit grant flow authentication is a bit less secure because it does not include a client secret and the access token is passed directly back to the client code running in the browser.

When you create an Azure AD application as a Web app, you can configure it with secret credentials to achieve stronger levels of authentication. In most cases, you will also configure an Azure AD application which as a Web app with one or more Reply URLs. Reply URLs add an extra security dimension because Azure AD can verify that the application is running within a pre-configured DNS domain on the Internet. This can really help to decrease the surface area that is exposed to attackers.

Azure AD also makes it possible to create an Azure AD application as a Native app instead of as a Web app. Native apps are used for specific scenarios such as a .NET application running on the laptop computer or an iOS app running on an iPhone. An important aspect of a Native app is that it is considered to be a *public client*. Unlike a web app which can keep track of server-side secrets, native apps cannot keep secrets such as client credentials. Therefore, Native apps can only authenticate with a user name and password. This means that a native app cannot establish application identity nor can it take advantage of application permissions.

So why am I going into all this detail about native apps? As it turns out, it’s important to 3rd party embedding where you must create the Azure AD application for your custom application as a native app. I will explain why this requirement exists later in this post. For now, I just want you to keep in mind that Native app is more restricted and less secure than a web app in several ways.

### Configuring Delegated Permissions

An essential aspect of creating the Azure AD application for a custom application is configuring the required permissions. Configuring an Azure AD application with required permissions is what makes it possible for your application to call Azure AD-secured resources such as the Power BI Service API and the Microsoft Graph API.

Each resource that is secured by Azure AD defines its own custom set of permissions. This means that the set of permissions for the Power BI Service API will be different from the set of permissions for the Microsoft Graph API. For example, the Power BI Service API defines permissions such as *Dashboard.Read.All*, *Report.ReadWrite.All* and *Content.Create*. The Microsoft Graph API defines different permissions such as *Calendars.Read*, *Contacts.ReadWrite* and *Files.ReadWrite*.

Let's say you want to configure permissions for your new Azure AD application by in the Azure portal. Figure 3.7 shows the *Add API access* blade and the *Enable Access* blade in the Azure portal which make it possible to configure Power BI Service permissions for you application. As you can see in the *Enable Access* blade, there are two different types of permissions which include delegated permissions and application permissions. It's important that you understand the differences between these two different types of permissions.

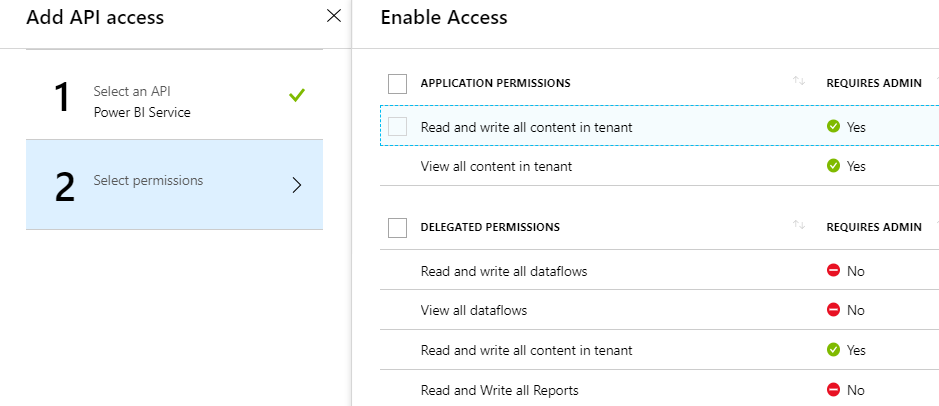


Figure 3.7: The Power BI Service API provides application permissions and delegated permissions.

*Delegated permissions* are used to call into an API with delegated access on behalf of a specific user. Delegated permissions are based on the principle that users can grant an application a subset of their own permissions. Delegated permissions are more restrictive than application permissions because they can never grant a level of permissions greater than the permissions of the current user.

Application permissions are used when your application makes calls to a resource with an app-only identity. An important observation is that application permissions can be far more powerful than delegated permissions. That's because application permissions are never restricted by the permissions of any particular user. Let’s look at an example of delegated permissions and application permissions using the Power BI Service API.

The Power BI Service API provides a delegated permission named *View all reports*. If your application is granted that permission, you can access all the reports that the current user is allowed to view. However, you will not be able to access any report to which the current user does not have access.

Now let's compare this delegated permission to an application permission. The Power BI Service API provides an application permission named *View all content in tenant*. Obviously, this application permission is far more powerful because it allows your application to access any Power BI content in all workspaces across the current Azure AD tenant.

Remember that the type of authentication flow you choose to implement in a custom application determines the type of permissions you can use. In order to take advantage of application permissions, you must authenticate the application without any user identity using the Client Credentials flow which will generate an app-only access token. The other three types of authentication flows will generate access tokens that contain a user identity in addition to the app identity. When an access token contains the user identity, your code will always rely on delegated permissions instead of application permissions.

As you can see from Figure 3.7, some delegated permissions have their *REQUIRES ADMIN* property set to true. This means that a user requires Power BI administrative permissions in order to grant those permissions to your application. It also means that a user requires Power BI administrative permissions just to log into the application. Therefore, it is important to use these *REQUIRES ADMIN* permissions sparingly because they prevent any user without administrative permissions from logging in or using the application in any way.

An important aspect of using delegated permissions has to do with obtaining user consent. The central idea is that a user needs to grant delegated permissions to an application by consenting before that application can make calls on behalf of that user. The act of the user consenting to your application is what actually grants the delegated permissions you’re your application requires.

Consider a simple example that illustrates how user consent works. Imagine you are developing a custom application using first party embedding application where users must authenticate using their Azure AD user accounts. Azure AD provides a **common consent framework** which provides built-in interactive behavior when a user logs into an application with delegated permissions for the first time. After each user successfully authenticates for the first time, Azure AD will prompt the user with an interactive *Permissions requested* dialog like the one shown in Figure 3.8.

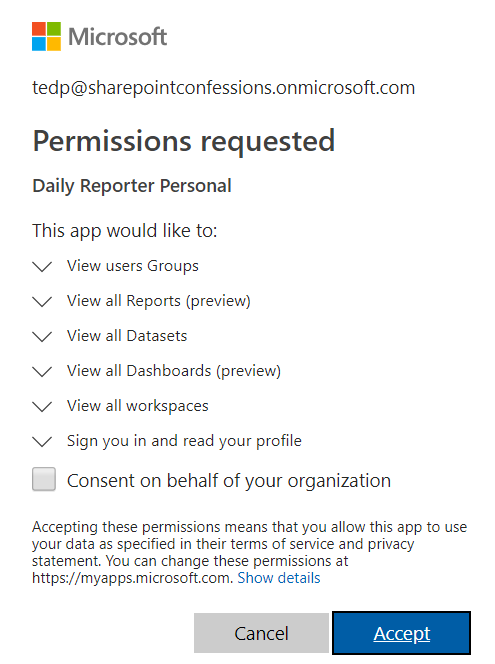


Figure 3.8: Azure AD provides a Common Consent framework which allows user to grant delegated permissions.

The *Permissions requested* dialog shown in Figure 3.8 lists all the delegated permissions required by the application. If the user clicks the *Accept* button, it will effectively grant the application the delegated permissions it requires to execute calls to Azure AD-secured resources on behalf of the current user. Once a user clicks the *Accept* button, Azure AD remembers that this user has consented and it does not need to interact with the user in future authentication requests. Azure AD is able to track which users have already consented and which users still need to provide their consent when they first log into the application.

The *Permissions requested* dialog shown in Figure 3.8 displays a checkbox with the caption *Consent on behalf of your organization*. This option is made available to administrators who have the ability to consent for all users in the organization at once making it unnecessary for individuals users to go through the consent process themselves. The *Required permissions* blade in the Azure portal as sown in Figure 3.9 provides the *Grant permissions* button which accomplishes the same goal. When you click the *Grant permissions* button, it automatically grants all delegated permissions to your application for all users at once.

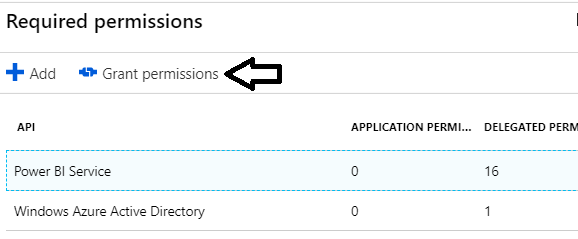


Figure 3.9: You can click the Grant permissions button in the Azure portal to consent for all users at once.

The one last technical detail to note about user consent with the Azure AD v1.0 endpoint. That is the Azure AD v1.0 endpoint does not support dynamically updating the list of permission grants over time. Instead, when a user consents to an application's required permissions, the permission list for that user is created as a static list that cannot be updated.

Consider a scenario in which you have configured an Azure AD application with four requested permissions and you have also deployed the application into production. Each user that logs into the application and consents to the requested permissions will have a permissions list with these four granted permissions.

Now imagine the business requirements for the application change forcing you to add two more required permission to the application. While you can update the Azure AD application by adding these two new permissions, there is no elegant way to propagate these new permissions into the existing list of permission grants for user who have already consented. The only way to accomplish this goal is to delete the granted permission list for all users so you can begin the consent process for each user with a fresh start.