Authenticating with Azure AD

In the Developer Quick Start, you learned to use the Power BI Embedding Onboarding Experience to provide a quick and easy way to set up a Power BI embedding development environment. Behind the scenes, the Onboarding Experience wizard called into Azure AD to create and configure a new Azure AD application on your behalf. While the Onboarding Experience is a convenient way to get started, it will only take you so far. If you want to develop with Power BI embedding for real-world scenarios, you must learn how authentication and authorization works in Azure AD at a fundamental level.

Your motivation for learning to work with Azure AD stems from the Power BI embedding requirement to call the Power BI Service API. To successfully call the Power BI Service API, you must first acquired an access token from Azure AD. Now, here is where things can get complicated because the manner in which you acquire access tokens varies significantly depending upon the scenario for which you are developing. Therefore, this section will begin with a primer on how authentication and authorization works in Azure AD from the ground up.

OAuth 2.0 and OpenID Connect

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

It's not really as confusing as it sounds. You don't have to think about *OAuth 2.0* and *OpenID Connect* separately. Instead, these two open standards combine to create a single protocol with one set of rules. When you hear developers talk about OAuth with Azure AD, it's usually implied that they are talking about *OAuth 2.0* combined with *OpenID Connect*.

The authorization framework of OAuth 2.0 defines four roles in the authorization process including the client, resource owner, authorization server and resources 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 *resources* (aka *resource servers*) represent secured endpoints on the Internet which you need to access such as the Microsoft Graph API and the Power BI Service API.

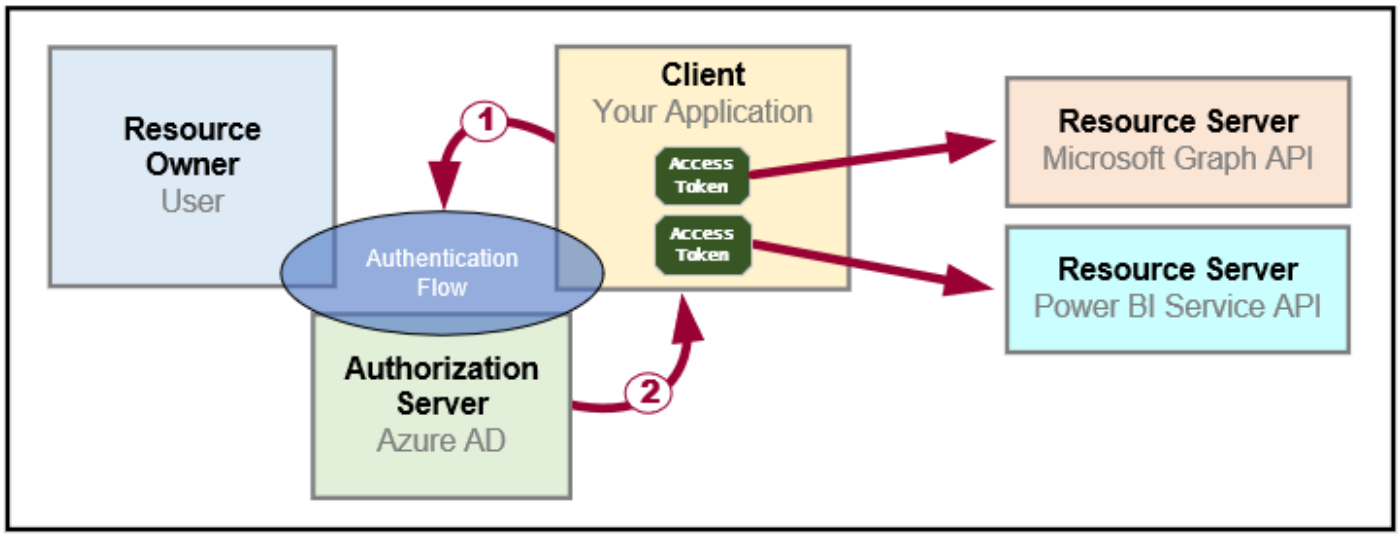

Figure 3.1: OAuth 2.0 provides a framework for distributing access tokens to client applications.

At the heart of OAuth 2.0 lies the process of transferring an access token from the authorization server to the client in a safe and secure manner. This process is known as an *authentication flow*. The client implements an authentication flow to retrieve access tokens from the authorization server. You must choose the type of authentication flow for each application you build depending on upon the details of your development scenario. 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
- **Authorization Code Flow**: used to authenticate the user in web applications
- **Implicit Flow**: used to authenticate the user in an SPA (i.e. single page application with only client-side code)
- **Client Credentials Flow**: used to authenticate the application with an app-only identity

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

A key principle in OAuth 2.0 is that of *delegated access*. This is different than impersonation because your application will not call to a resource using the identity and permissions of the user. Instead, your application gets its own separate identity which is granted a subset of the user's permissions. When using delegated access, your application calls to a resource *on behalf of the user* instead of calling *as the user*.

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, an ID the authorization server and an ID for the target resource server. In many cases, the access token also includes a user-specific ID for 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. Since app-only authentication does not involve establishing user identity, the resulting access token will not contain any ID for a user.

Keep in mind that every access token is generated to provide access to one specific resource. As an example, consider a scenario in which you are developing an application that needs to call both the Power BI Service API and the Microsoft Graph API. You will need acquire a separate access token for each API because they are recognized as separate resources.

When you implement an authentication flow to acquire an access token, you pass a *resource identifier* to the authorization server to indicate the target resource. A resource identifier is really just a unique string but it's formatted as a URI. The resource identifier for the Power BI Service API is *https://analysis.windows.net/powerbi/api*. The resource identifier for the Microsoft Graph API is *https://graph.microsoft.com*.

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 contains a *duration which* defines the lifetime for which the access token is valid. Access tokens are given a relatively short lifetime to decrease the surface area for attackers. For example, Azure AD generates access tokens with a duration of 65 minutes. Once an access token expires, any attempt to use it with result in an access denied error.

The OAuth 2.0 framework provides *refresh tokens* to deal with the problem of expired access tokens. During certain types of authentication flows, the authorization server passes a refresh token to the client 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 (e.g. 90 days). Once the original access token expires, the client can use the refresh token as its credentials when it calls Azure AD to acquire a new access token. Given the long lifetime of a refresh token, it is common for the client to persist refresh tokens so they can be used across user sessions. The use of refresh tokens makes it possible to reduce the number of times a user is prompted with an interactive login and required to enter a user name and password.

As noted earlier, the OAuth 2.0 framework has a few shortcomings with respect to user authentication and identity. These shortcomings are addressed by OpenID Connect which adds a third type of security token known as an **ID token**. An ID token contains additional data on the authenticated user and makes it possible for the client to validate the user's identity.

There is one last fundamental detail you need to know about how OAuth 2.0 works. The OAuth 2.0 framework requires you to register the client application with the authorization server before it can be used. When an application is registered, it must be assigned an ID, a friendly name and a set of permissions as shown in Figure 3.2. Depending upon the type of application you are creating, you might also need to configure the registered application with reply URLs and credentials.

Authorization Server
Azure AD

Registered Applications

Name	App ID	Permissions	Reply URL	Credentials
App1	guid1	...	none	none
App2	guid2	...	...	secret key
App3	guid3	...	...	X.509 Certificate

Figure 3.2 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 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.

Now that you have learned the fundamentals of OAuth 2.0 and OpenID Connect, it's time to put all this theory to work and begin discussing what needs to be done in Azure AD to register and use a custom application. It is important to know that Azure AD currently supports two different implementations of OAuth 2.0 and OpenID Connect. These two different implementation go by the names of the v1.0 endpoint and the v2.0 endpoint.

The Azure AD v1.0 endpoint has been generally available for over 5 years and it is heavily used in production applications. You can register and configure Azure AD applications for the v1.0 endpoint using the Azure portal or a PowerShell script. The v2.0 endpoint and its user experience for registering applications is still in a preview program. Furthermore, the v2.0 endpoint only supports a subset of the features available through the v1.0 endpoint. Therefore, the section will focus on the Azure AD v1.0 endpoint and then briefly discuss the changes you can expect in the future if you decide to move to the v2.0 endpoint.

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 using a PowerShell script or by using 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. If you navigate to the **Azure Active Directory** link in the Azure portal and then click the **App registrations**, you will see a view that allows you to view existing applications in the current Azure AD tenant as shown in figure 3.3.

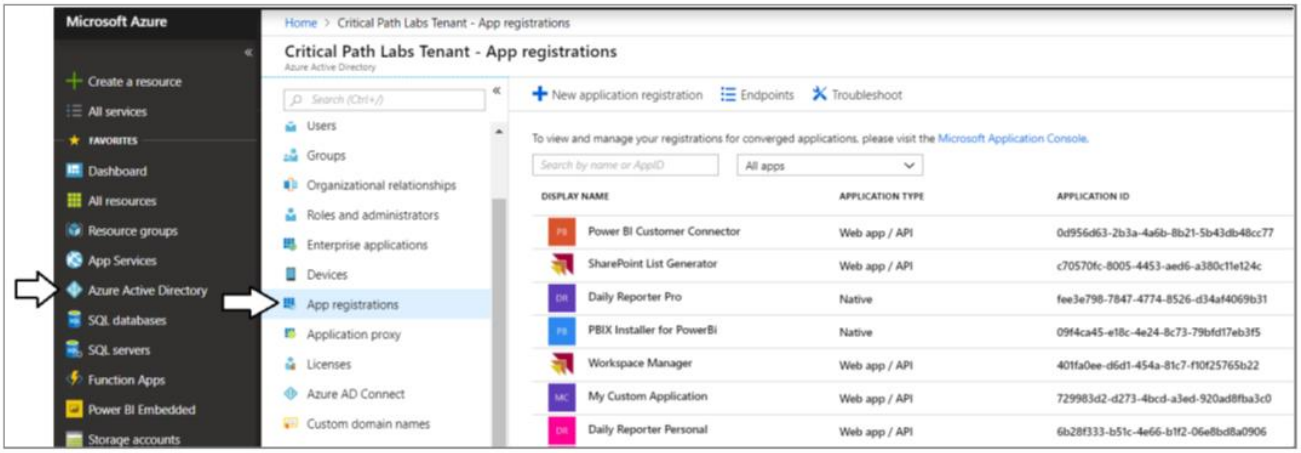

Figure 3.3: The Azure portal makes it possible to view and manage

The *App registrations* view in the Azure portal shown in Figure 3.3 provides a **New application registration** button which you can click to navigate to the **Create** blade where you can create new a new Azure AD application by hand. The screenshot shown in Figure 3.4 demonstrates entering the data required to create a new Azure AD application.

When creating a new Azure AD application, you must select an *Application type* of either *Web app / API* or *Native*. As you will learn throughout this chapter, some development scenarios call for you to create a Native application while other scenarios call for you to register your application as a Web app / API.

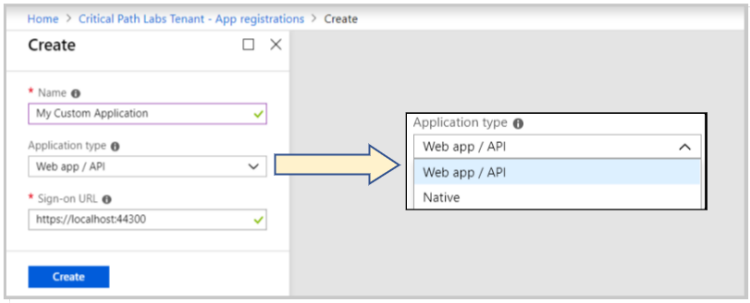

Figure 3.4: When creating an Azure AD application, you must select an Application Type of Native or Web app / API.

When you click the **Create** button to create a new application, Azure AD will generate a new GUID to serve as the Application ID as shown in Figure 3.4. The Application ID is similar to the user principle name for a Azure AD user account in that it references an identity that can be authenticated. Your application will be required to pass its Application ID to Azure AD to identify itself when it implements an authentication flow to acquire an access token.

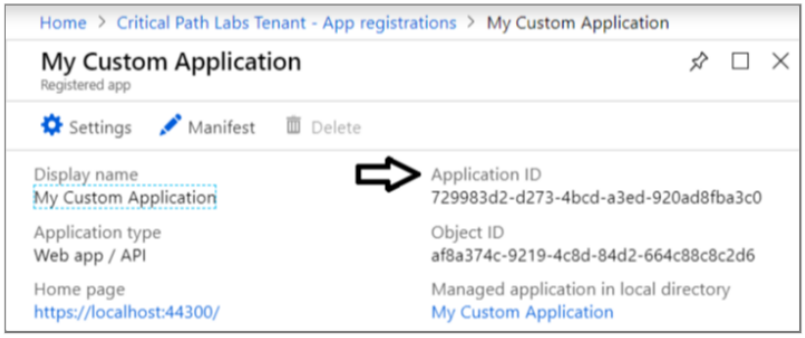

Figure 3.5: Whenever you create a new application, Azure AD will assign it a unique Application ID.

There is an unfortunate terminology issue and potential confusion surrounding the ID for an Azure AD application. In the Azure portal, the ID is referred to as an **Application ID**. However, many other developers, libraries and SDKs refer this ID as 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 that goes by the names of Bob and Robert.

Once you have initially created a new Azure AD application in the Azure portal, there is usually additional configuration that needs to be completed before you can actually use the application. If you click the **Setting** button as shown in Figure 3.6, you can navigate to the blades that make it possible to configure a new Azure AD application with reply URLs, permissions and security keys which act as credentials. You will see several examples of this throughout this section as you begin to implement each of the four authentication flows.

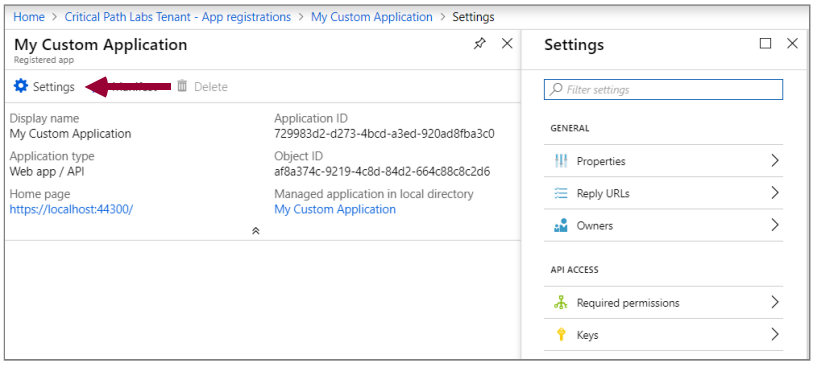

Figure 3.6: In the Azure portal, you can configure application settings such as Reply URLs, Permissions and Keys.

Configuring and Granting 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.

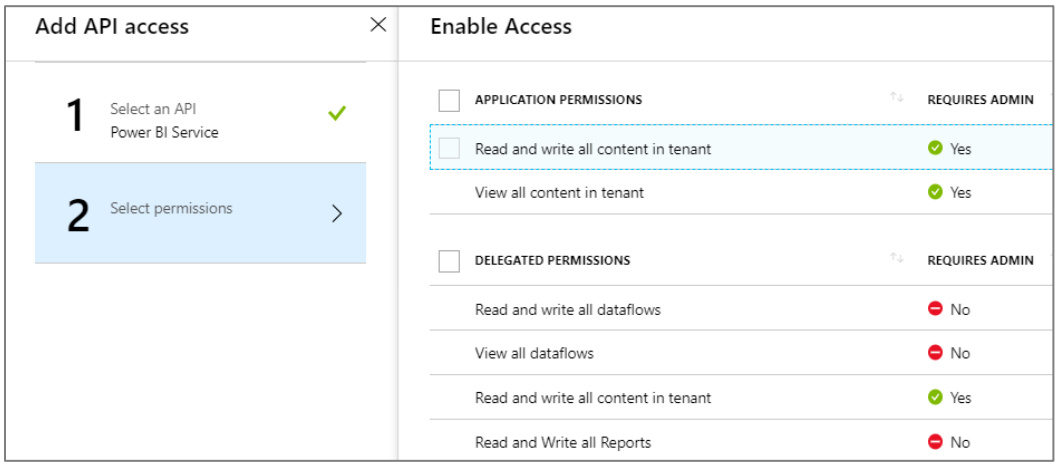

Figure 3.7: The Power BI Service API provides application permissions and delegated permissions.

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.

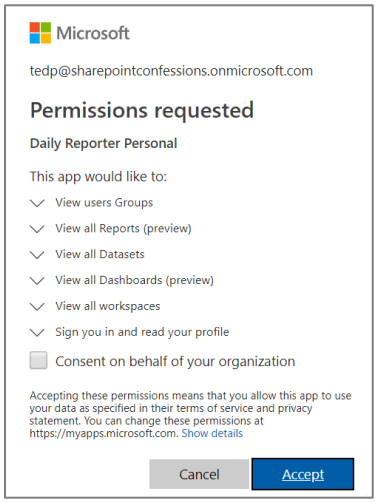

Figure 3.8: Azure AD provides a Common Consent framework which allows user to grant delegated permissions.

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.

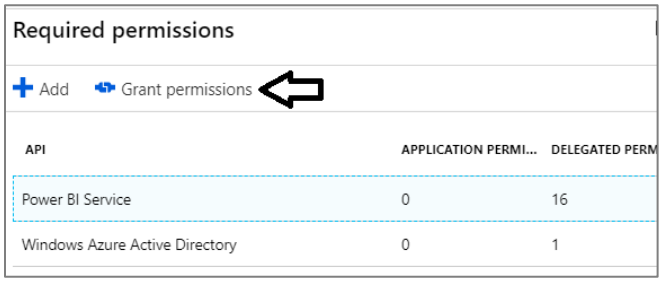

Figure 3.9: You can click the Grant permissions button in the Azure portal to consent for all users at once.

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.

Creating Azure AD Applications using PowerShell

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. This 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 also hardcode a user name and password into your PowerShell script to avoid having to enter credentials when you are running your scripts.

```
$userName = "myuseraccount@myorg.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 native 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 usually 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 Native client versus a Web app / API. You can create a Native 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 Web app / API.

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 reply URLs. When you create a new Azure AD application as a *Web app / API*, you must provide a reply URL that tells Azure AD where your application is running on the Internet. For example, the reply URL for a production application could be *https://myAzureWebApp.azurewebsites.net*. The reply URL 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 reply URL. You can configure an application with more than one reply URL in scenarios where it makes sense.

In the case of a native application, your application might require a reply URL. However, the reply URL for a native application does not have to be a real endpoint on the Internet. Instead, the reply URL for a native application just needs to be a string value formatted as a URI such as *https://localhost/app1234*.

When a native application authenticates using an interactive login, it must pass a reply URL to Azure AD that matches one of the reply URLs that have been configured for the application. Azure AD will return an access denied error if you pass a reply URL that does not match one of the reply URLs that has been registered with the application. Azure AD is also notoriously strict about returning access denied errors in cases where the reply URL matching fails due to case sensitivity or a missing backslash.

Understanding Service Principals in Azure AD

When you begin to create and work with Azure AD applications, it's important to understand the relationship between the Azure AD application object and another important Azure AD object known as the *service principal*. The service principal object acts as the identity for your application within a specific tenant. This begs the question "why can't the application object be used provide an identity for the application?" To answer this question requires a bit of background information.

Remember that a multitenant application is accessible to users across Azure AD tenants. In other words, a multitenant application can execute within the context of many different tenants. However, an Azure AD application requires 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 a single application ID, each tenant gets its own service principle with a unique object ID. The service principal object has the responsibility of tracking user consent and which delegated permissions have been granted to the application.

While the additional complexity of service principals was added to Azure AD to manage multitenant applications, it is something you still have to deal with when working with a single-tenant application. If you create an Azure AD application in PowerShell without creating a service principal, Azure AD will create the service principal on demand the first time the application is accessed by a user. However, it's a good practice to explicitly create the local service principal after an Azure AD application in your PowerShell scripts.

Note that you cannot pass an application ID when creating a new Azure AD application. Instead, Azure AD will always generate a new GUID for the application ID. 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.

```
# 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")

# create service principal for application
$appId = $aadApplication.AppId
$serviceServicePrincipal = New-AzureADServicePrincipal -AppId $appId

# assign current user as application owner
Add-AzureADApplicationOwner -ObjectId $aadApplication.ObjectId -RefObjectId $user.ObjectId
```

Note that this PowerShell script also performs one other common task. It assigns ownership of the application to the logged on user. When you create a new Azure AD application in the Azure portal or by using PowerShell, Azure AD will not assign a default owner. Instead, you must explicitly assign yourself or other users as the application owner.

Once you have created the service principal, you can begin to configure the application's requested permissions. To accomplish this, you can create a *RequiredResourceAccess* object and then set its *ResourceAppId* property to the application ID for the service you want to access. If your application requires permissions to call the Power BI Service API, you can pass its well-known application ID which is *00000009-0000-0000-c000-000000000000*.

```
$requiredAccess = New-Object -TypeName "Microsoft.Open.AzureAD.Model.RequiredResourceAccess"
$requiredAccess.ResourceAppId = "00000009-0000-0000-c000-000000000000"
```

The way in which you add a specific permission is by creating a *ResourceAccess* object which needs to be initialized with the GUID that identifies the specific permissions and the Scope parameters which indicated that the permission is a delegated permission as opposed to an application permission. Here is a simple example of creating a *ResourceAccess* object for the *Report.Read.All* permission which is a delegated permission of the Power BI Service API.

```
$permission1 = New-Object -TypeName "Microsoft.Open.AzureAD.Model.ResourceAccess" `
                          -ArgumentList "4ae1bf56-f562-4747-b7bc-2fa0874ed46f","Scope"
```

Once you have created the *ResourceAccess* objects for all the delegated permissions you need, you can assign them to the *RequiredResourceAccess* object and then assign the *RequiredResourceAccess* object to target application using the following PowerShell code.

```
# configure delegated permisssions for the Power BI Service API
$requiredAccess = New-Object -TypeName "Microsoft.Open.AzureAD.Model.RequiredResourceAccess"
$requiredAccess.ResourceAppId = "00000009-0000-0000-c000-000000000000"

# create first delegated permission - Report.Read.All
$permission1 = New-Object -TypeName "Microsoft.Open.AzureAD.Model.ResourceAccess" `
                          -ArgumentList "4ae1bf56-f562-4747-b7bc-2fa0874ed46f","Scope"

# create second delegated permission - Dashboards.Read.All
$permission2 = New-Object -TypeName "Microsoft.Open.AzureAD.Model.ResourceAccess" `
                          -ArgumentList "2448370f-f988-42cd-909c-6528efd67c1a","Scope"

# add permissions to ResourceAccess list
$requiredAccess.ResourceAccess = $permission1, $permission2

# add permissions by updating application with RequiredResourceAccess object
Set-AzureADApplication -ObjectId $aadApplication.ObjectId -RequiredResourceAccess $requiredAccess
```

As you look at the proceeding PowerShell script, you first thought it likely "so where do I find all these GUIDs that I use to identify specific delegated permissions?" You can answer this question by writing a simple PowerShell script that enumerates through the *Oauth2Permissions* collection property of the service principal object. Here is a simple example.

```
Connect-AzureAD

$powerBiServiceAppId = "00000009-0000-0000-c000-000000000000"
$powerBiService = Get-AzureADServicePrincipal | Where-Object {$_.AppId -eq $powerBiServiceAppId}
$powerBiService.Oauth2Permissions | Sort-Object Type, Value | Format-Table Type, Value, Id
```

When you run this PowerShell script, its output is shown as a table in the following listing. You can use this table to look up the IDs for any of the delegated permissions from the Power BI Service API you need when configuring an application.

```
Type  Value                    Id
----  -----                    --
Admin Tenant.Read.All          01944dba-21df-426f-bb8c-796488be96ad
Admin Tenant.ReadWrite.All     d594897b-76e7-4b2b-984b-b4adff35e109
User  Capacity.Read.All        76e2ebd5-0dfb-4a5b-93c7-ed89e0362834
User  Capacity.ReadWrite.All   4eabc3d1-b762-40ff-9da5-0e18fdf11230
User  Content.Create           f3076109-ca66-412a-be10-d4ee1be95d47
User  Dashboard.Read.All       2448370f-f988-42cd-909c-6528efd67c1a
User  Dashboard.ReadWrite.All  b271f05e-8329-4b97-baa4-91cf15b99cf1
User  Data.Alter_Any           ecc85717-98b0-4465-af6d-1cbba6f9c961
User  Datapool.Read.All        f9759906-80a4-4f4a-b010-24b832bc6a30
User  Datapool.ReadWrite.All   ddd37690-e119-40c5-a821-3746ea6125c4
User  Dataset.Read.All         7f33e027-4039-419b-938e-2f8ca153e68e
User  Dataset.ReadWrite.All    322b68b2-0804-416e-86a5-d772c567b6e6
User  Group.Read               a65a6bd9-0978-46d6-a261-36b3e6fdd32e
User  Group.Read.All           47df08d3-85e6-4bd3-8c77-680fbe28162e
User  Metadata.View_Any        ecf4e395-4315-4efa-ba57-a253fe0438b4
User  Report.Read.All          4ae1bf56-f562-4747-b7bc-2fa0874ed46f
User  Report.ReadWrite.All     7504609f-c495-4c64-8542-686125a5a36f
User  Workspace.Read.All       b2f1b2fa-f35c-407c-979c-a858a808ba85
User  Workspace.ReadWrite.All  445002fb-a6f2-4dc1-a81e-4254a111cd29
```

Active Directory Authentication Library

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 between your application and Azure AD. 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 doesn't mean you should.

Microsoft provides the *Azure Active Directory Library (ADAL)* to assist developers meet the requirements of implementing authentication flows for the Azure AD v1.0 endpoint. There is one version of ADAL for .NET developers (*ADAL.NET*) which can be used to implement authentication flows with managed languages such as C#. There is a second version of ADAL for JavaScript (ADAL.JS) used to implement implicit authentication flows in single page applications (SPAs) created with JavaScript frameworks such as React.js and AngularJS.

ADAL adds value to the development process by abstracting away many of the low-level details required to implement an authentication flow with Azure AD. When you're programming with ADAL, you don't have to worry about sending HTTP requests to Azure AD or parsing the HTTP response to extract the access code. ADAL does that for you. If this isn't enough for you, ADAL provide even more value in certain scenarios by caching access tokens and refresh tokens.

To add ADAL.NET to a Visual Studio project, install the NuGet package *Microsoft.IdentityModel.Clients.ActiveDirectory*. This NuGet package adds the ADAL.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 ADAL.NET along with a few other valuable developer resources which is accessible through the following URL.

```
https://github.com/AzureAD/azure-activedirectory-library-for-dotnet
```

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

https://github.com/AzureAD/azure-activedirectory-library-for-js

There is one aspect of working with ADAL.JS that can be confusing and frustrating if you are not using the AngularJS framework. The problem is that the standard *adal.js* library has been packaged together with a complimentary library named *adal.angular.js* which contains a custom AngularJS service. The *adal.angular.js* library is great addition which adds a lot of value when you are developing client-side applications with AngularJS. But if you creating an SPA using React.js, it's confusing because Microsoft does not provide a package that will allow you to add *adal.js* to your project without also adding the unneeded library named *adal.angular.js*.

Now that you have learned about creating Azure AD applications and adding ADAL to a Visual Studio project, it's finally time to start writing some code and learning how to implement an Azure AD authentication flow. By the end of this chapter, you will learn how to implement each of the four primary authentication flow types including user password credential flow, authorization code flow, Implicit flow and client credentials flow.

Calling the Power BI Service API from a Native Application

Let's begin by creating a new native application and going through all the steps required to call the Power BI Service API. Developing a native application is a good starting point when first learning how to authenticate with Azure AD because it does not require as much complexity as an application registered as a Web app \ API. In our first example, we will be creating a C# console application in Visual Studio which calls into the Power BI Service API. The first step is to create an new Azure AD application and configure it with the delegated permissions you require to call the Power BI Service API. You can accomplish this step by running the following PowerShell script.

```
# connect to Azure AD
$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")

# create service principal for application
$appId = $aadApplication.AppId
$serviceServicePrincipal = New-AzureADServicePrincipal -AppId $appId

# assign current user as application owner
Add-AzureADApplicationOwner -ObjectId $aadApplication.ObjectId -RefObjectId $user.ObjectId

# configure delegated permisssions for the Power BI Service API
$requiredAccess = New-Object -TypeName "Microsoft.Open.AzureAD.Model.RequiredResourceAccess"
$requiredAccess.ResourceAppId = "00000009-0000-0000-c000-000000000000"

# create first delegated permission - Report.Read.All
$permission1 = New-Object -TypeName "Microsoft.Open.AzureAD.Model.ResourceAccess" `
                          -ArgumentList "4ae1bf56-f562-4747-b7bc-2fa0874ed46f","Scope"

# create second delegated permission - Dashboards.Read.All
$permission2 = New-Object -TypeName "Microsoft.Open.AzureAD.Model.ResourceAccess" `
                          -ArgumentList "2448370f-f988-42cd-909c-6528efd67c1a","Scope"

# add permissions to ResourceAccess list
$requiredAccess.ResourceAccess = $permission1, $permission2

# add permissions by updating application with RequiredResourceAccess object
Set-AzureADApplication -ObjectId $aadApplication.ObjectId -RequiredResourceAccess $requiredAccess
```

When you run this PowerShell script, it will create a new native application in Azure AD with required Power BI Service API permissions and a reply URL of *https://localhost/app1234*. As the execution of the PowerShell script creates the new application, Azure AD will automatically generate a new GUID for the application ID. After you have created the Azure AD application, your next steps are to create a new C# console application in Visual Studio and to install the NuGet package

for ADAL.NET named Microsoft.IdentityModel.Clients.ActiveDirectory as shown in Figure 3.10. You should also install a second NuGet package named *Newtonsoft.Json* which will be used convert JSON returned the Power BI Service API into strongly-typed objects making it easier to access JSON-based content returned from the Power BI Service API when you are programming in C#.

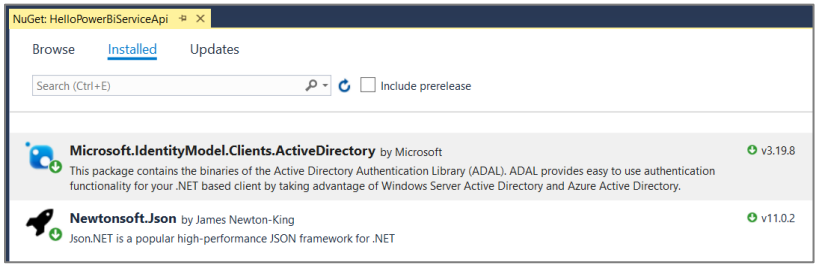

Figure 3.10: ADAL.NET is added by installing the NuGet package named Microsoft.IdentityModel.Clients.ActiveDirectory.

Now it's time to write a complete C# program that calls the Power BI Service API. First, the program must implement an authentication flow to call to Azure AD and obtain an access token for the Power BI Service API. Second, the application must transmit the access token along with any HTTP request sent to the Power BI Service API. Let's start by examining the entire program at once and, after that, we'll walk through smaller sections of this code.

```
using System;
using System.Net;
using System.Net.Http;
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using Newtonsoft.Json;
using HelloPowerBiServiceApi.Models;

namespace HelloPowerBiServiceApi {

  class Program {

    const string aadAuthorizationEndpoint = "https://login.windows.net/common/oauth2/authorize";
    const string resourceUriPowerBi = "https://analysis.windows.net/powerbi/api";
    const string clientId = "c07dc205-fe75-45dd-93d8-48aa30cac269";
    static readonly Uri redirectUri = new Uri("https://localhost/app1234");

    static string GetAccessToken() {
      var authContext = new AuthenticationContext(aadAuthorizationEndpoint);
      var promptBehavior = new PlatformParameters(PromptBehavior.Auto);
      AuthenticationResult result =
        authContext.AcquireTokenAsync(resourceUriPowerBi, clientId, redirectUri, promptBehavior).Result;
      return result.AccessToken;
    }

    static string ExecuteGetRequest(string restUrl) {
      HttpClient client = new HttpClient();
      HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, restUrl);
      request.Headers.Add("Authorization", "Bearer " + GetAccessToken());
      request.Headers.Add("Accept", "application/json;odata.metadata=minimal");
      HttpResponseMessage response = client.SendAsync(request).Result;
      if (response.StatusCode != HttpStatusCode.OK) {
        throw new ApplicationException("Error occured calling the Power BI Servide API");
      }
      return response.Content.ReadAsStringAsync().Result;
    }

    static void Main() {
      var json = ExecuteGetRequest("https://api.powerbi.com/v1.0/myorg/reports/");
      ReportCollection reports = JsonConvert.DeserializeObject<ReportCollection>(json);
      foreach (Report report in reports.value) {
        Console.WriteLine(report.name);
      }
    }

  }
}
```

Let's begin with the constant values in the application. First, there is the constant named *aadAuthorizationEndpoint* which tracks the URL which is used to begin the authentication flow.

```
const string aadAuthorizationEndpoint = "https://login.microsoftonline.com/common";
```

When you use ADAL to implement an authentication flow, the user will be redirected to the authorization endpoint in the browser which will begin an interactive login experience. Note that this example uses the common authorization endpoint. Using the common authorization endpoint is required when you are developing multitenant applications. It's also convenient when working single-tenant applications.

For scenarios where you are developing applications that only authenticate with one Azure AD tenant, it is also possible to configure the authorization endpoint using the tenant id.

```
https://login.microsoftonline.com/37bf5ca4-68cb-4f6a-b915-efd9d1dcb35a
```

Next, there is a constant named *resourceUriPowerBi* which is the resource identifier for the Power BI Service API. This value must be passed to Azure AD in order to create access tokens which allow you to call the Power BI Service API.

```
const string resourceUriPowerBi = "https://analysis.windows.net/powerbi/api";
```

The application also contains a constant named *clientid* and a read-only variable named *redirectUri*. These values must match the values for the Azure AD application you have created. These values are important because they must be passed to Azure AD during an authentication flow in order to acquire an access token.

```
const string clientId = "c07dc205-fe75-45dd-93d8-48aa30cac269";
static readonly Uri redirectUri = new Uri("https://localhost/app1234");
```

Now that you understand the purpose of each program constant, it's time to walk through the code inside the *GetAccessToken* function which uses ADAL to implement a interactive authentication flow. The code begins by creating an *AuthenticationContext* object which is initialized using the authorization endpoint.

```
var authContext = new AuthenticationContext(aadAuthorizationEndpoint);
```

Next, the code calls *AcquireTokenAsync* passing the resource identifier for the Power BI Service API, the client id, the redirect URI and a parameter to control the interactive prompt behavior. This is the call that begins the authentication flow.

```
AuthenticationResult authResult =
  authContext.AcquireTokenAsync(resourceUriPowerBi, clientId, redirectUri, promptBehavior).Result;
```

When you call *AcquireTokenAsync* method in a desktop application such as a C# console application, ADAL is able to provide a browser-based login experience by prompting the user with a dialog containing an embedded instance of Internet Explorer. This interactive experience allows a user to enter login credentials and to consent to delegated permissions just as if the user were logging into a browser-based application. You can create a PlatformParameters object which allows you to control whether the user is always prompted for login credentials or whether the program can use cached credentials from a previous login.

```
var promptBehavior = new PlatformParameters(PromptBehavior.Auto);
```

Once the user completes the interactive log in experience, the interactive dialog is dismissed and the return value from the *AcquireTokenAsync* method makes it possible to retrieve the access token. While the *AcquireTokenAsync* method exhibits asynchronous behavior, you can assign the *Result* property from the *AcquireTokenAsync* return value to a *AuthenticationResult* variable to simulate calling a synchronous methods to simplify your code. The *AuthenticationResult* object has several useful properties but the one we are interested in here is the AccessToken property.

```
static string GetAccessToken() {
  var authContext = new AuthenticationContext(aadAuthorizationEndpoint);
  var promptBehavior = new PlatformParameters(PromptBehavior.Auto);
  AuthenticationResult authResult =
    authContext.AcquireTokenAsync(resourceUriPowerBi, clientId, redirectUri, promptBehavior).Result;
  return authResult.AccessToken;
}
```

Now let's move on to the *ExecuteGetRequest* function which uses an *HttpClient* object to execute an HTTP GET operation. The main point to see here is that this function adds the *Authorization* header to each requests and sets the value for this header to a string that combines the word "Bearer" together with a space and an access token returned from the *GetAccessToken* function. As long as the call to *client.SendAsync* returns a successful HTTP status code, the *ExecuteGetRequest* function returns the content from the HTTP GET operation as a string.

```
static string ExecuteGetRequest(string restUrl) {
  HttpClient client = new HttpClient();
  HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, restUrl);
  request.Headers.Add("Authorization", "Bearer " + GetAccessToken());
  request.Headers.Add("Accept", "application/json;odata.metadata=minimal");
  HttpResponseMessage response = client.SendAsync(request).Result;
  if (response.StatusCode != HttpStatusCode.OK) {
    throw new ApplicationException("Error occured calling the Power BI Servide API");
  }
  return response.Content.ReadAsStringAsync().Result;
}
```

When can now move on to the last function named *Main*. When the *Main* function begins to execute, it calls *ExecuteGetRequest* and passes the REST URL required by the Power BI Service API to retrieve the reports in the current users personal workspace.

```
string restUrl = "https://api.powerbi.com/v1.0/myorg/reports/";
var json = ExecuteGetRequest(restUrl);
```

The call to *ExecuteGetRequest* triggers a call to *GetAccessToken* which begins the authentication flow. The user should be prompted to log in by entering a user name and password. If delegated permissions have not yet been granted, the user will be prompted with the consent dialog. Once the user has completed the interactive login, the call to GetAccessToken returns an access token back to the *ExecuteGetRequest* function which then passes the access token in the *Authorization* header when it calls to the Power BI Service. Figure 3.11 shows an example of using the popular Fiddler utility to inspect a call to the Power BI Service and see the access token that's being transmitted in the *Authorization* header.

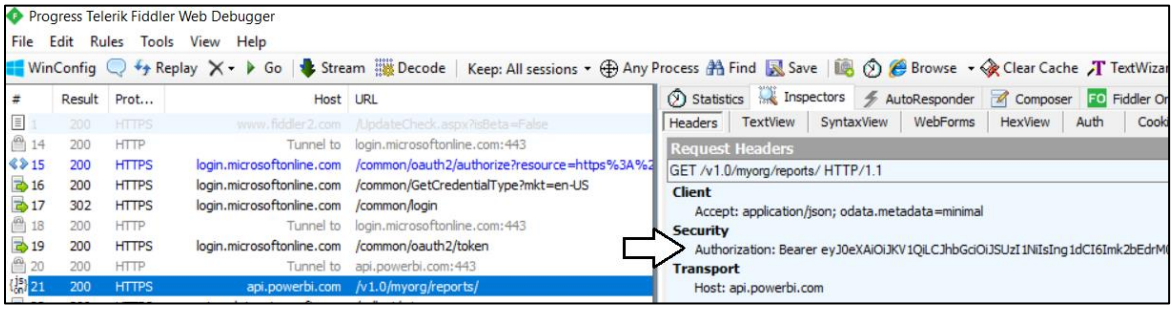

Figure 3.11: All calls to the Power BI Service API must pass an access token in the Authorization header

At this point, the call to *ExecuteGetRequest* returns back to *Main* with a string value containing the JSON returned from the Power Service API with the data for the reports in the user's personal workspace. The Power BI Service usually returns JSON results in a standard ODATA format as shown in Figure 3.12.


```
{
  "@odata.context": "http://wabi-us-west2-redirect.analysis.windows.net/v1.0/myorg/$metadata#reports",
  "value": [
    {
      "id": "11c84cab-5ce7-4495-88cd-2f454ed691f7",
      "name": "Wingtip Sales Analysis",
      "webUrl": "https://app.powerbi.com/reports/11c84cab-5ce7-4495-88cd-2f454ed691f7",
      "embedUrl": "https://app.powerbi.com/reportEmbed?reportId=11c84cab-5ce7-4495-88cd-2f454ed691f7&config=eyJjb
      "isOwnedByMe": true,
      "datasetId": "1e3a238b-b6ee-419b-a83c-fac131ed9cf8"
    },
    {
      "id": "811b0dfd-598b-4954-ab05-bba300361d95",
      "name": "Northwind Retro",
      "webUrl": "https://app.powerbi.com/reports/811b0dfd-598b-4954-ab05-bba300361d95",
      "embedUrl": "https://app.powerbi.com/reportEmbed?reportId=811b0dfd-598b-4954-ab05-bba300361d95&config=eyJjb
      "isOwnedByMe": true,
      "datasetId": "743c481a-101f-4694-b15b-5de3e3c82212"
    }
  ]
}
```


Figure 3.12: This is an example of the JSON format returned by the Power BI Service API.

The final step now is to convert the JSON string value into strongly-typed objects to simply the code. This can be accomplished by defining the following two classes in C# and using them in a call to *JsonConvert.DeserializeObject*.

```
public class Report {
  public string id { get; set; }
  public string name { get; set; }
  public string webUrl { get; set; }
  public string embedUrl { get; set; }
  public bool isOwnedByMe { get; set; }
  public string datasetId { get; set; }
}

public class ReportCollection {
  public List<Report> value { get; set; }
}
```

The call to *JsonConvert.DeserializeObject* returns an object with a *value* property containing a collection of *Report* objects that you can enumerate using a C# foreach loop. This makes it possible to display the names of each report to the console window.

```
static void Main() {
  var json = ExecuteGetRequest("https://api.powerbi.com/v1.0/myorg/reports/");
  ReportCollection reports = JsonConvert.DeserializeObject<ReportCollection>(json);
  foreach (Report report in reports.value) {
    Console.WriteLine(report.name);
  }
}
```

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.

1. ADAL 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. ADAL 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 ADAL 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 rewriting the *GeAccessToken* method shown earlier. This technique involves 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 *AcquireTokenAsync*.

```
static string GetAccessToken() {

  var authContext = new AuthenticationContext(aadAuthorizationEndpoint);

  var userPasswordCredential = new UserPasswordCredential("user1@myorg.onMicrosoft.com", "pass@word1");

  AuthenticationResult authResult =
      authContext.AcquireTokenAsync(resourceUriPowerBi, clientId, userPasswordCredential).Result;

  return authResult.AccessToken;
}
```

When you run the program with this new implementation of *GetAccessToken*, 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*.

Programming the Authorization Code Flow in a Web App

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.

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

Azure AD maintains an application manifest for every registered applications.

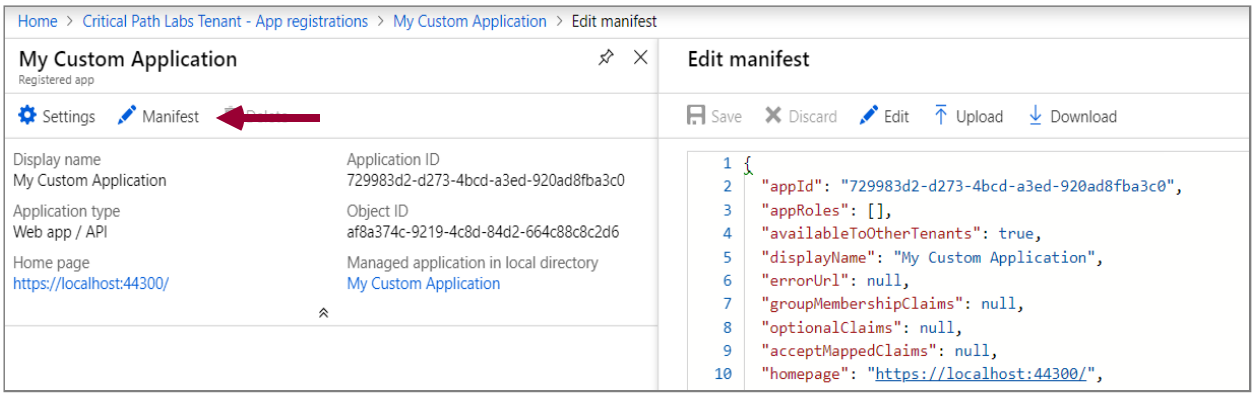

Figure 3.6: The Azure portal makes it possible to view and, if necessary, edit the application manifest.

Programming the Client Credentials Flow in a Web App

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.