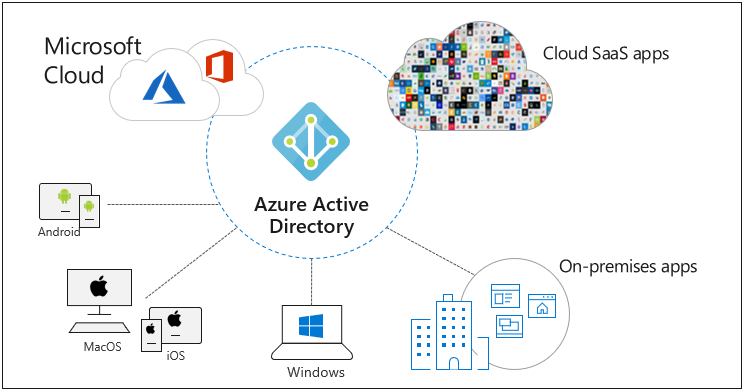
**Agenda:** **Authentication and Authorization using Azure Active Directory**

* Brief about Azure AD
* Application Types Scenarios
* Multi-Tenant vs Single-Tenant
* Application and Service Principal
* Programming using Active Directory Authentication Library (ADAL)
* Abount Microsoft Identity Platform
* About Microsoft Authentication Library (MSAL)
* Authentication flows
* Programming Authentication using MSAL.NET
* Microsoft Graph API
* Integrating ASP.NET MVC Applications with Azure AD
* Integrating ASP.NET Web API Applications with Azure AD
* Claims Based Authentication
* Role Based Authentication
* Implementing Azure AD B2C Collaboration
* Social Identity Provider Authentication

**Azure Active Directory**

Microsoft Azure Active Directory (Azure AD) is Microsoft’s cloud-based **identity and access management** service. Azure AD creates and manages credentials that help enterprise users sign in and access both internal and external resources that are offered by your company or third-party companies more securely.

Azure AD's geographically distributed architecture combines extensive monitoring, automated rerouting, failover, and recovery capabilities, which deliver company-wide availability and performance to customers.



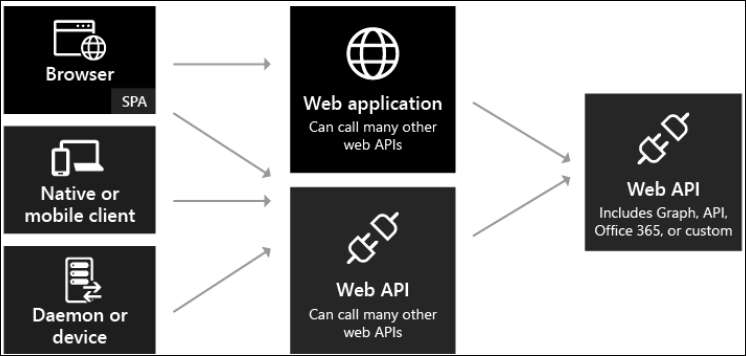
**You can use Azure AD to:**

* Provide an identity management solution.
* Manage users and groups.
* Role based Access Control (RBAC).
* Enable federation between organizations.
* Identify irregular sign-in activity.
* Configure SSO to cloud-based SaaS applications like Office365, Salesforce.com, DropBox etc…
* Configure access to the on-premise applications.
* Configure multi-factor authentication (MFA).
* Extend existing on-premises Active Directory implementations to Azure AD.

**Application Types Scenarios**

**These are the five primary application scenarios supported by Azure AD:**

1. **Web browser to web application**: A user needs to sign in to a web application that is secured by Azure AD.
2. **Single-page application (SPA):** A user needs to sign in to a single-page application that is secured by Azure AD.
3. **Web application to web API:** A web application needs to get resources from a web API secured by Azure AD.
4. **Native application to web API:** A native application that runs on a phone, tablet, or PC needs to authenticate a user to get resources from a web API that is secured by Azure AD.
5. **Daemon or server application to web API**: A daemon application or a server application with no web user interface needs to get resources from a web API secured by Azure AD.



#### **Single-tenant vs multi-tenant apps**

There are two categories of applications that can be developed and integrated with Azure AD:

* **Single tenant application** - A single tenant application is intended for use in one organization. These are typically line-of-business (LoB) applications written by an enterprise developer. A single tenant application only needs to be accessed by **users in one directory**, and as a result, it only needs to be provisioned in one directory. These applications are typically registered by a developer in the organization.

EndPoint:

* + <https://login.microsoftonline.com/sandeepsonideccansoft.onmicrosoft.com>
  + <https://login.microsoftonline.com/ef404960-95a9-49fb-be86-72acd7a3bc27>
* **Multi-tenant application -** A multi-tenant application is intended for use in many organizations, not just one organization. These are typically software-as-a-service (SaaS) applications written by an independent software vendor (ISV). Multi-tenant applications need to be **provisioned in each directory where they will be used**, which requires user or administrator consent to register them. This consent process starts when an application has been registered in the directory and is given access to the Graph API or perhaps another web API. When a user or administrator from a different organization signs up to use the application, they are presented with a dialog that displays the permissions the application requires. The user or administrator can then consent to the application, which gives the application access to the stated data, and finally registers the application in their directory.
  + EndPoint: [https://login.microsoftonline.com](https://login.microsoftonline.com/sandeepsonideccansoft.onmicrosoft.com)**/common**

**About Azure AD Application and Service Principal**

When you register an Azure AD application in the Azure portal, two objects are created in your Azure AD tenant:

1. An application object
2. A service principal object

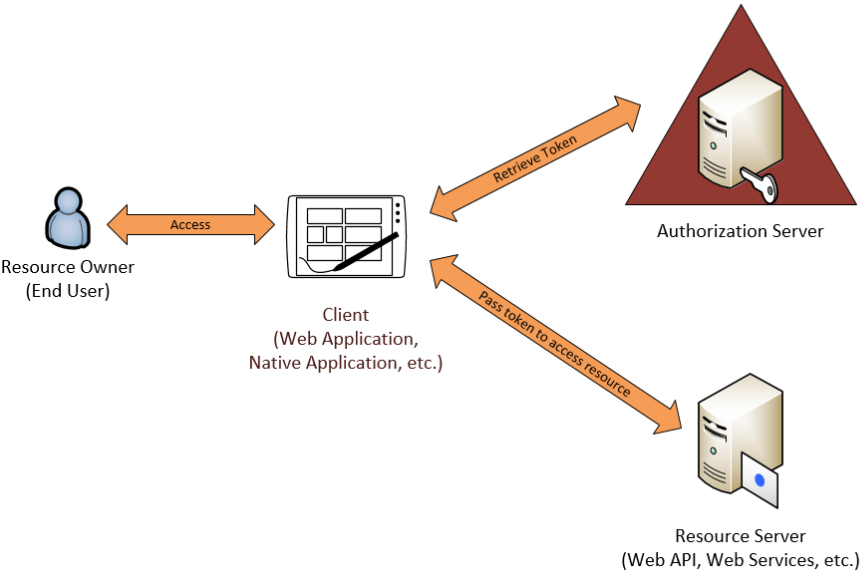
**Azure AD application** is defined by its one and only **application object**, which resides in the Azure AD tenant where the application was registered, known as the application's “home” tenant.

To access resources that are secured by an Azure AD tenant, the entity that requires access must be represented by a **security principal**. This is true for both users (**user principal**) and applications (**service principal**). The security principal defines the **access policy** and **permissions** for the user or application in the Azure AD tenant. This enables core features such as **authentication** of the user or application during sign-in and **authorization** during resource access.

Consider the application object as the global representation of your application for use across all tenants, and the service principal as the local representation for use in a specific tenant. An application object therefore has a 1:1 relationship with the software application, and a 1:many relationships with its corresponding service principal object(s).

**Actors involved in OAuth 2:**

1. **User / Resource Owner:** The resource owner is the **end user** who is giving access to some portion of his/her account.
2. **Authorization Server**: The server where the client application is registered and returns the access token after it gets consent from the resource owner for accessing protected resources hosted by a resource server.
3. **Resource Server:** The Web API or Web Service server which hosts the secured users protected resources and are protected by OAuth2. The resource server validates the access-token and serves the protected resources. Eg: Photo Sharing site, online bank service or any other service where **users private stuff** is kept.
4. **Third party Client Application:** The application that is attempting to get access to the user's account or a resource form Resource Server. It can be website, desktop or mobile application or a set-top box or anything connected to the web. Eg: Photo Printing Application/Website.



**Azure AD Application Registration**

The application registration might include, depending on the type:

* **Application ID URI:** The identifier for an application. This value is sent to Azure AD during authentication to indicate which application the caller wants a token for.
* **Reply URL and redirect URI** : For a web API or web application, the Reply URL is the location where Azure AD will send the authentication response, including a token if authentication was successful. For a native application, the Redirect URI is a unique identifier to which Azure AD will redirect the user-agent in an OAuth 2.0 request.
* **Application ID:** The ID for an application, which is generated by Azure AD when the application is registered. When requesting an authorization code or token, the Application ID and Key are sent to Azure AD during authentication.
* **Key/Secret:** The key that’s sent along with an Application ID when authenticating to Azure AD to call a web API

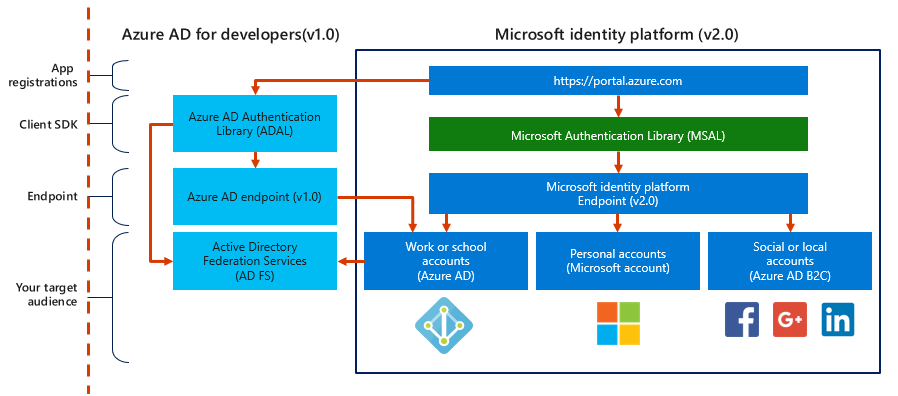
**About Microsoft Identity Platform v2.0**

Microsoft identity platform is an evolution of the Azure Active Directory (Azure AD) developer platform.

It allows developers to build applications that sign in **all Microsoft Identities** and get tokens to call Microsoft APIs, such as Microsoft Graph, or APIs that developers have built.

**The Microsoft identity platform consists of:**

* **OAuth 2.0 and OpenID Connect standard-compliant authentication service** that enables developers to authenticate any Microsoft identity, including:
  + Work or school accounts (provisioned through Azure AD)
  + Personal Microsoft accounts (such as Skype, Xbox, and Outlook.com)
  + Social or local accounts (via Azure AD B2C)
* **Open-source libraries:** Microsoft Authentication Libraries (MSAL) and support for other standards-compliant libraries.
* **Application management portal:** A registration and configuration experience built in the Azure portal, along with all your other Azure management capabilities.



**Microsoft Authentication Library (MSAL):**

* The library to streamline working with Microsoft identity platform from code:
  + Obtains and manages tokens.
  + Caches tokens by using a configurable cache.
  + Refreshes tokens automatically when they expire.
  + Supports asynchronous invocation.
* Available on multiple platforms such as:
  + .NET
  + JavaScript
  + Android
  + iOS
  + Java

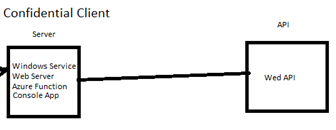
**Public client and Confidential client applications**

Microsoft Authentication Library (MSAL) defines two types of clients:





1. **Confidential client applications** are apps that run on servers (Web Apps, Web API apps, or even service/daemon apps). A web app is the most common confidential client. The client ID is exposed through the web browser, but the secret is passed only in the back channel and never directly exposed.



Uses the MSAL **ConfidentialClientApplication** class.

string redirectUri = "https://myapp.azurewebsites.net";

IConfidentialClientApplication app = ConfidentialClientApplicationBuilder

.Create(clientId)

 .WithAuthority(AzureCloudInstance.AzurePublic, \_tenantId)

 .WithClientSecret(**clientSecret**)

.WithRedirectUri(redirectUri)

 .Build();

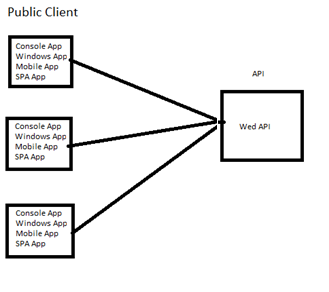
AzureCloudInstance.AzurePublic =

<https://login.microsoftonline.com/ef404960-95a9-49fb-be86-72acd7a3bc27>

OR

<https://login.microsoftonline.com/common>

1. **Public client applications** are apps that run on devices or desktop computers or in a web browser SPA. They're not trusted to safely keep application secrets, so they only access Web APIs on behalf of the user. (They support only public client flows.) Public clients can't hold configuration-time secrets, so they don't have client secrets. Uses the MSAL **PublicClientApplication** class.

****

var app = PublicClientApplicationBuilder

 .Create(\_clientId)

 .WithAuthority(AzureCloudInstance.AzurePublic, \_tenantId)

 .WithRedirectUri("http://localhost")

 .Build();

**Authentication flows:**

* **Authorization code:** Native and web apps securely obtain tokens in the name of the user.
* **Interactive**: User authenticates by using a web browser. Mobile and desktops applications call Microsoft Graph in the name of a user.
* **Client credentials**: Service applications run without user interaction.
* **On-behalf-of**: Application authenticates on behalf of a user.
* **Implicit:** Used in browser-based applications.
* **Device code**: Enables sign-in to a device by using another device that has a browser.
* **Integrated Windows**: Windows computers silently acquire an access token when they are domain joined.
* **Username/password**: The application signs in a user by using their username and password.

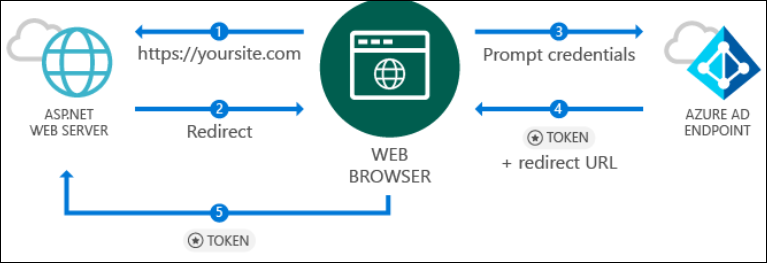
string[] scopes = { "user.read" };

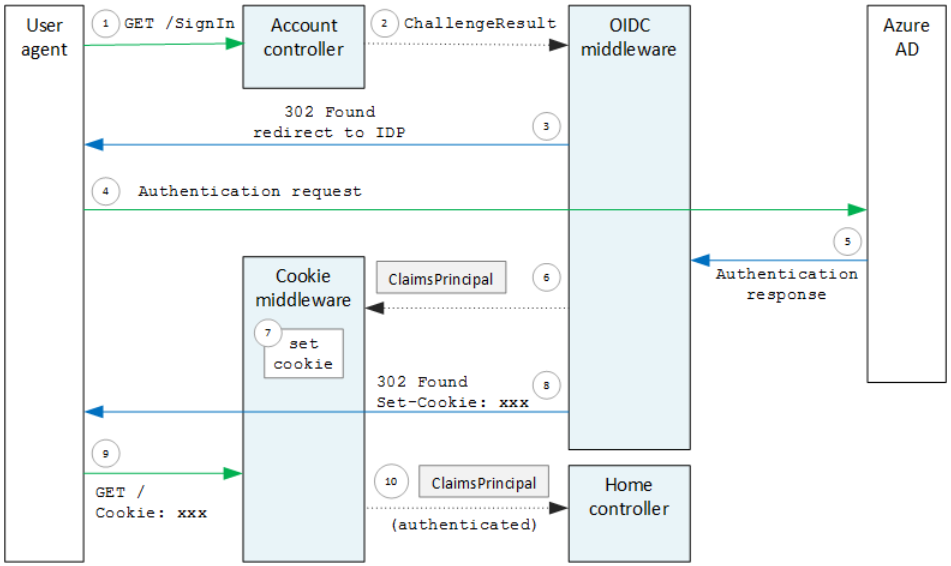
AuthenticationResult result = await app.AcquireTokenInteractive(scopes).ExecuteAsync();

**Very Important**: <https://docs.microsoft.com/en-us/azure/active-directory/develop/msal-authentication-flows>

**Integrating Web Applications with Azure Active Directory**

Organizations that develop their own line-of-business (LOB) applications can protect access to those applications by using Azure AD. Developers can enable their own custom applications to use Azure AD, and obtain the same features that are available in the Azure AD gallery applications.





**Web App Sign In & Sign Out with Azure AD:**

In ASP.NET web apps, you can accomplish this using Microsoft's implementation of the **community-driven OWIN middleware** included in .NET Framework 4.8.

**Registering an AD App using Azure Portal**

1. Azure Portal 🡪 Azure Active Directory 🡪 App registrations 🡪 **+ New registration**
2. Provide Name="MySecuredWebApp", Redirect URI (optional) **Select a platform = Web**, Sign-on URL: [http://localhost:~~44336~~/signin-oidc](http://localhost:44336/signin-oidc) (Run the web application and replace the 44336 port with what ever is assigned for your application)
3. Azure Portal 🡪 Azure Active Directory 🡪 App registrations 🡪 Select MySecuredWebApp 🡪 Authentication 🡪 Check **ID tokens** (used for implicit and hybrid flows)
4. **Copy Application ID**
5. Create a New ASP.NET MVC Web Application
   1. File 🡪 New 🡪 Project 🡪 ASP.NET Web Application 🡪 Name=DssDemoWebApp 🡪 OK
   2. Select a template = MVC, Click on **Authentication=None**
   3. Create
6. Add a NuGet Package: **Microsoft.Identity.Web**
7. Add the following to **Program.cs** (6.0)

builder.Services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)

.**AddMicrosoftIdentityWebApp**(builder.Configuration.GetSection("**AzureAd**"));

**And**

**app.UseAuthentication();**

app.UseAuthorization();

1. **Add the below to AppSettings.json**

{

  "AzureAd": {

    "Instance": "https://login.microsoftonline.com/",

    "Domain": "sandeepsonideccansoft.onmicrosoft.com",

    "TenantId": "82d8af3b-d3f9-465c-b724-0fb186cc28c7",

    "ClientId": "da1ebcac-01ec-4053-8078-93b057545eb6",

    "CallbackPath": "/signin-oidc"

  }

}

1. **Remove the attribute** [Authorize] from HomeController in Controllers\HomeController.cs
2. Open the Controllers\HomeController.cs file. You can access the user's claims in your controllers via the ClaimsPrincipal.Current security principal object.

[Authorize]

public IActionResult About()

{

ViewBag.Username = User.Claims.FirstOrDefault(claim => claim.Type == "name")?.Value;

ViewBag.TenantId = User.Claims.FirstOrDefault(claim => claim.Type == "http://schemas.microsoft.com/identity/claims/tenantid")?.Value;

return View();

}

1. Edit Views/Home/About.cshtml

<link href="@Url.Content("~/Content/bootstrap.min.css")" rel="stylesheet" type="text/css" />

<h3>Main Claims:</h3>

<table class="table table-striped table-bordered table-hover">

<tr><td>Username</td><td>@ViewBag.Username</td></tr>

<tr><td>TenantId</td><td>@ViewBag.TenantId</td></tr>

</table>

<br />

<h3>All Claims:</h3>

<table class="table table-striped table-bordered table-hover table-condensed">

@foreach (var claim in ((System.Security.Claims.ClaimsIdentity) User.Identity).Claims)

{

<tr><td>@claim.Type</td><td>@claim.Value</td></tr>

}

</table>

<br />

<br />

@Html.ActionLink("Sign out", "SignOut", "Account", null, new { @class = "btn btn-primary" })

1. **Finally, build and run your app.**

If you haven't already, now is the time to create a new user in your tenant with a \*.onmicrosoft.com domain. Sign in with that user, and notice how the user's identity is reflected in the top navigation bar. Sign out, and sign back in as another user in your tenant. If you're feeling particularly ambitious, register and run another instance of this application (with its own clientId), and watch see single-sign on in action.

**Configure sign-in options – Multitenant vs Singletenant**

**Single Tenant Option:**

If you want your application to accept sign-ins only from accounts that belong to a specific Azure AD instance (including *guest accounts* of that instance)

Tenant parameter in *appsettings.json* should be as below

{

  "AzureAd": {

    "Instance": "https://login.microsoftonline.com/",

    "Domain": "sandeepsonideccansoft.onmicrosoft.com",

    "TenantId": "82d8af3b-d3f9-465c-b724-0fb186cc28c7",

    "ClientId": "da1ebcac-01ec-4053-8078-93b057545eb6",

    "CallbackPath": "/signin-oidc"

  }

}

**Multi Tenant Option: Configure your application to allow sign-ins of work and school accounts from any company or organization (multi-tenant)**

1. Go back to Microsoft Azure portal - **App registrations** and locate the application you registered.
2. Select **Authentication** 🡪 select **Supported account types** = **Accounts in any organizational directory (Any Azure AD directory – Multitenant).**
3. Select **Save**.
4. Edit **Appsettings.json**

  "AzureAd": {

    "Instance": "https://login.microsoftonline.com/ ",

    "ClientId": "29d2c719-e33d-4ca4-a2eb-cd21459e701b",

"TenantId": "common",

    "CallbackPath": "/signin-oidc"

  }

**Optional: To restrict access to only users of few tenants**

1. In Program.cs 🡪 Move all existing code to Program class, Main method.
2. Add the following methods below to Main method

private string **ValidateSpecificIssuers**(string issuer, SecurityToken securityToken,

TokenValidationParameters validationParameters)

{

var validIssuers = **GetAcceptedTenantIds**()

.Select(tid => $"https://login.microsoftonline.com/{tid}/v2.0");

if (validIssuers.Contains(issuer))

return issuer;

throw new SecurityTokenInvalidIssuerException("The sign-in user's account does not belong to one of the tenants that this Web App accepts users from.");

}

private string[] **GetAcceptedTenantIds**()

{

return new[]

{

"82d8af3b-d3f9-465c-b724-0fb186cc28c7",

"2de8d54d-5576-4bf6-b419-6065cb1e700e"

};

}

1. Edit **Program.cs** and add the below service configuration

services.Configure<OpenIdConnectOptions>(OpenIdConnectDefaults.AuthenticationScheme, options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

IssuerValidator = **ValidateSpecificIssuers**,

ValidateIssuer = true

};

});

**To allow only Authorized Users to Login to the Application**

1. Azure Active Directory 🡪 **Enterprise Application** 🡪 All Applications 🡪 change filter: Application type = All Applications 🡪 Select DssDemoApp 🡪 **Properties**
2. Properties 🡪 **User assignment required** = Yes 🡪 Save
3. Now add users to **Users and Groups** and only these users will be able to login to the application.

**Java Example:**

<https://docs.microsoft.com/en-us/azure/active-directory/develop/quickstart-v2-java-webapp>

<https://docs.microsoft.com/en-us/azure/developer/java/spring-framework/configure-spring-boot-starter-java-app-with-azure-active-directory>

<https://docs.microsoft.com/en-us/azure/developer/java/spring-framework/spring-boot-starter-for-azure-active-directory-developer-guide>

**Calling a Web API from a Daemon Application (server to server call) – Authentication Flow: Client Credentials**

To Demonstrate how the Application can use its own Identity (not its loggedin users) to access the WebAPI

Diagram

Description automatically generated

**Application roles** are exposed by web APIs called by daemon applications (that calls your web API on their own behalf).

**Diagram

Description automatically generated**

**Step1: Azure Portal: Create an Azure AD App for WebAPI.**

1. Azure Portal 🡪 Azure Active Directory 🡪 App Registration 🡪 **+ New Registration**, Name=MyWebApi
2. MyWebAPI 🡪 Overview 🡪 Add an Application ID URI 🡪 Set 🡪 **Application ID URI**=https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI 🡪 Save
3. Note Application (ClientID), Tenant ID.
4. MyWebAPI 🡪 Owners 🡪 + Add owner 🡪 Search and Select for your Id 🡪 Select **(Only if AD App is created using VS along with API App)**
5. MyWebAPI **🡪 App roles** 🡪 + Create app role 🡪 Display Name="**access\_as\_application**", Select **Application**, Value="access\_as\_application", Description="any string", Apply

This will edit Manifest as below:

MyWebAPI 🡪 **Manifest** (Edit a below)

"appRoles": [

**{**

        "allowedMemberTypes": [

            "Application"

        ],

        "description": "Accesses the MyWebAPI as an application.",

        "displayName": "access\_as\_application",

        "id": "fad303c2-d9e0-4c8e-8113-85964ec372fc",

        "isEnabled": true,

        "lang": null,

        "origin": "Application",

        "value": "access\_as\_application"

    }

]

**Step2: Build WebAPI Application**

1. Visual Studio: Create a Web API Application
2. Edit appsettings.json

"AzureAd": {

"Instance": "https://login.microsoftonline.com/",

"Domain": "sandeepsonideccansoft.onmicrosoft.com",

"TenantId": "82d8af3b-d3f9-465c-b724-0fb186cc28c7",

"ClientId": "b34aafc5-2d79-46f7-afbb-e02045766c53",

"Audience": "https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI"

}

1. Add NUGET Package: **Microsoft.Identity.Web**
2. Edit Program.cs

using Microsoft.Identity.Web;

builder.Services.AddMicrosoftIdentityWebApiAuthentication(builder.Configuration, "AzureAd");

**app.UseAuthentication();**

app.UseAuthorization();

1. Use **[Authorize]** attribute whereever required. (Either for API Controller or it’s action methods)

**Step3: Azure Portal: Create an Azure AD App for Console Application.**

1. Azure Portal 🡪 Azure Active Directory 🡪 App Registration 🡪 + New Registration, Name=**MyConApp**
2. API permission 🡪 + Add a permission 🡪
   1. + Add a permission 🡪 Microsoft Graph 🡪 **Application permission** 🡪 Expand Users 🡪 Check **User.ReadAll** 🡪 Add permissions
   2. + Add a permission 🡪 My APIs Tab 🡪 Select MyWebAPI, select Application permission and check **access\_as\_aplication** 🡪 Add permissions.
   3. Click **Grand admin consent for the (AD) Directory.**

**Step 4: Develop the Console Application to use Secure API**

Add NuGet Package reference **Microsoft.Identity.Client**

using System;

using System.Net.Http;

using System.Net.Http.Headers;

using System.Threading.Tasks;

using Microsoft.Identity.Client;

class Program

{

private const string \_clientId = "<client id of AD App for ConApp";

private const string \_secret = ". . . ";

private const string \_tenantId = "82d8af3b-d3f9-465c-b724-0fb186cc28c7";

public static async Task Main(string[] args)

{

IConfidentialClientApplication app = **ConfidentialClientApplicationBuilder**

.Create(\_clientId)

.WithAuthority(AzureCloudInstance.AzurePublic, \_tenantId)

.WithClientSecret(\_secret)

.Build();

//Invoding Microsoft Graph API

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

AuthenticationResult result = await app.**AcquireTokenForClient**(scopes).ExecuteAsync();

Console.WriteLine($"Token:\t{result.AccessToken}");

string endpoint = "https://graph.microsoft.com/v1.0/users";

var client = new HttpClient();

var authHeader = new AuthenticationHeaderValue("Bearer", result.**AccessToken**);

client.DefaultRequestHeaders.Authorization = authHeader;

var response = await client.GetAsync(endpoint);

string json = await response.Content.ReadAsStringAsync();

Console.WriteLine(json);

//Invoding Custom Web API

var client1 = new HttpClient();

string[] scopes1 = { "https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI/.default" };

var result1 = await app.AcquireTokenForClient(scopes1).ExecuteAsync();

client1 = new HttpClient();

var authHeader1 = new AuthenticationHeaderValue("Bearer", result1.AccessToken);

client1.DefaultRequestHeaders.Authorization = authHeader1;

string endpoint1 = "https://localhost:5200/weatherforecast"; //URL of WebAPI

var response1 = await client1.GetAsync(endpoint1);

string json1 = await response1.Content.ReadAsStringAsync();

Console.WriteLine(json1);

}

}

**Java Example:**

<https://docs.microsoft.com/en-us/azure/active-directory/develop/quickstart-v2-java-daemon>

**Invoking a Secure API from Client using Users Identity - Authentication Flow: Interactive**

**To Demonstrate how the Application can use its logged-in users identity to access the WebAPI**

**Step1: Create a New AD Application for Web API**

1. Select **App registrations**, and then select **New registration**
   1. Name = MyWebAPI
   2. Supported account types = Accounts in this organizational directory only (Personal Directory only - Single tenant)
   3. Redirect URIs = <Leave it as blank> (Web APIs don't need to register a redirect URI because no user is interactively signed in)
2. **Overview** 🡪 Application ID URL = <https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI>
3. **Expose an API** 🡪 + Add a Scope 🡪
   1. for **Scope name** use **user\_impersonation**
   2. Ensure the **Admins and users** option is selected for **Who can consent**
   3. in **Admin consent display name** type Access My Web API as a Admin
   4. in **Admin consent description** type Accesses the My Web API as a Admin
   5. in **User consent display name** type Access My Web API as a user
   6. in **User consent description** type Accesses the My Web API as a user
   7. Keep **State** as **Enabled**
   8. Select **Add scope**

**Step2: Build WebAPI Application**

1. Visual Studio: Create a Web API Application
   1. Write a method which returns **User.Claims.First(claim => claim.Type == "name").Value**
2. Edit appsettings.json

"AzureAd": {

"Instance": "https://login.microsoftonline.com/",

"Domain": "sandeepsonideccansoft.onmicrosoft.com",

"TenantId": "82d8af3b-d3f9-465c-b724-0fb186cc28c7",

"ClientId": "b34aafc5-2d79-46f7-afbb-e02045766c53",

"Audience": <https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI>

}

1. Edit Configure.cs

public void ConfigureServices(IServiceCollection services)

{

**services.AddMicrosoftIdentityWebApiAuthentication(Configuration, "AzureAd");**

**}**

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

**app.UseAuthentication();**

app.UseAuthorization();

}

1. Use **[Authorize]** attribute whereever required.
2. Add the following to WebAPI Method to check if the request has the scope **user\_impersonation** or not

[HttpGet]

**[Authorize]**

public IEnumerable<WeatherForecast> Get()

{

//User.Claims.First(claim => claim.Type == "name").Value

var apiClaim = User.Claims.Where(c => c.Type == "http://schemas.microsoft.com/identity/claims/scope" && c.Value.Contains("user\_impersonation")).FirstOrDefault();

if (apiClaim == null)

{

throw new ApplicationException("Unauthorized-The Scope claim does not contain 'user\_impersonation' or scope claim not found");

}

var rng = new Random();

return Enumerable.Range(1, 5).Select(index => new WeatherForecast

{

Date = DateTime.Now.AddDays(index),

TemperatureC = rng.Next(-20, 55),

Summary = Summaries[rng.Next(Summaries.Length)]

})

.ToArray();

}

**Step3: Create an Azure AD Application for Client Application (Mobile App or Console App)**

1. Azure Portal 🡪 Active Directory 🡪 App Registrations 🡪 New registration 🡪 Name=MyConApp
2. Name = MyDemoApp, Supported Account Types = Accounts in this organizational directory only
3. Redirect URL: Public **client/native** (mobile & desktop), <http://localhost>
4. **API Permissions** 🡪 Add a permission 🡪 My APIs 🡪 Select MyWebAPI 🡪 Check **user\_impersonation** 🡪 Add permission.

**Step4: .NET Core Console Application**

Add reference to NuGet package **Microsoft.Identity.Client**

using System;

using System.Net.Http;

using System.Net.Http.Headers;

using System.Threading.Tasks;

using Microsoft.Identity.Client;

class Program

{

private const string \_clientId = "2226001b-e403-4f74-b9b6-76133a83d990";

private const string \_tenantId = "82d8af3b-d3f9-465c-b724-0fb186cc28c7";

public static async Task Main(string[] args)

{

IPublicClientApplication app = **PublicClientApplicationBuilder**

.Create(\_clientId)

.WithAuthority(AzureCloudInstance.AzurePublic, \_tenantId)

.WithRedirectUri("http://localhost")

.Build();

//Invoking Graph API

string[] scopes = { "user.read" }; //{ "https://graph.microsoft.com/user.read" };

AuthenticationResult result = await app.**AcquireTokenInteractive**(scopes).ExecuteAsync();

Console.WriteLine($"Token:\t{result.AccessToken}");

string endpoint = "https://graph.microsoft.com/v1.0/me";

var client = new HttpClient();

var authHeader = new AuthenticationHeaderValue("Bearer", result.**AccessToken**);

client.DefaultRequestHeaders.Authorization = authHeader;

var response = await client.GetAsync(endpoint);

string json = await response.Content.ReadAsStringAsync();

Console.WriteLine(json);

//Custom API

string[] scopes1 = { "https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI/user\_impersonation" };

AuthenticationResult result1 = await app.**AcquireTokenInteractive**(scopes1).ExecuteAsync();

var client1 = new HttpClient();

var authHeader1 = new AuthenticationHeaderValue("Bearer", result1.AccessToken);

client1.DefaultRequestHeaders.Authorization = authHeader1;

string endpoint1 = "https://localhost:44384/weatherforecast"; //URL of WebAPI

var response1 = await client1.GetAsync(endpoint1);

string json1 = await response1.Content.ReadAsStringAsync();

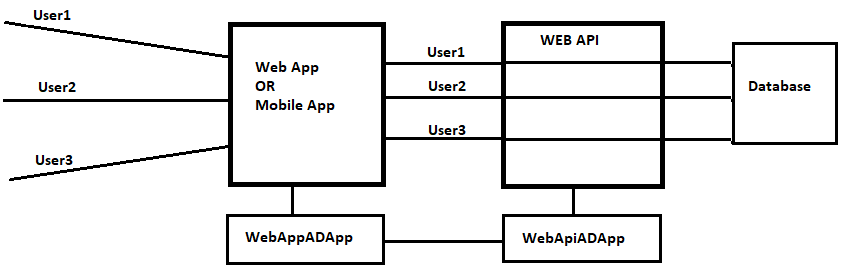
Console.WriteLine(json1);

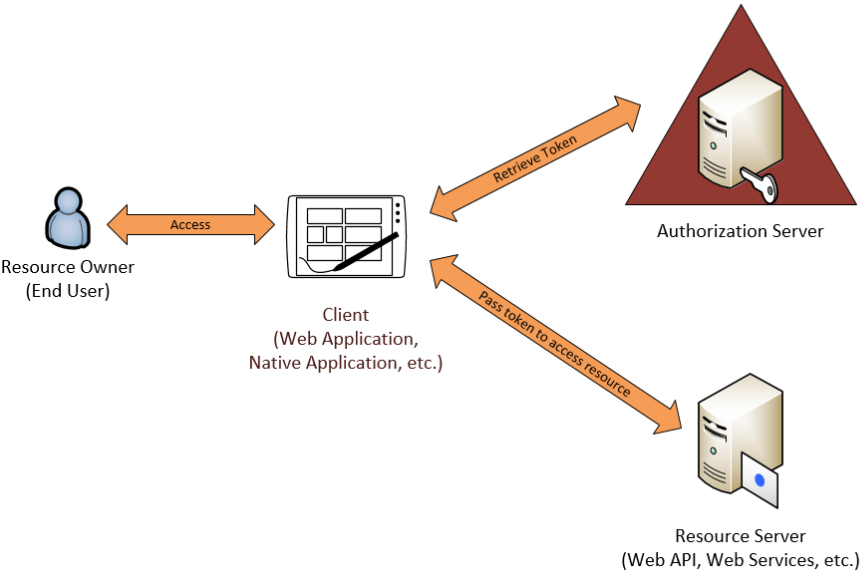
}

}

Note: The **AcquireTokenSilent** will return the token it already has in cache if it is still valid or get a new one using refresh token or cookies in case implicit id\_token. You can only make this call however if you are sure that you already have an access token or use has already been authenticated by a previous non-silent acquire token call.

**Accessing Secure Web API from Web App using Logged-In Users Identity**





**Build WebAPI and WebApp and Test without security**

1. Create New ASP.NET Core Web Application (**Web API Template**). The template auto generates WeatherForecast API Controller and we will use the same for our demo.
2. Create New ASP.NET Core Web Application **(MVC Contoller Template**)
3. Add the following to **appSettings.json of WebApp** (Assuming that 44342 is the SSL port of Web API)

"APIUrl": "https://localhost:44342/"

1. Add to the project **WeatherForecast** class (Copy from Web API project and change the namespace)
2. **Edit HomeController, Index method**

HttpClient httpClient = new HttpClient();

string url = \_config["ApiUrl"] + "weatherforecast";

var request = new HttpRequestMessage()

{

    RequestUri = new Uri(url)

};

IEnumerable<WeatherForecast> cols = null;

using (var response = httpClient.SendAsync(request).Result)

{

    cols = JsonConvert.DeserializeObject<IEnumerable<WeatherForecast>>(response.Content.ReadAsStringAsync().Result);

};

return View(cols);

1. Right Click on Index and Add a New View – Razor View
   1. View Name = Index
   2. Template = List
   3. Model class = WeatherForecast
   4. Add
2. **Run both WebAPI and WebApp and test the application.**

**Create a New AD Application for Web API**

1. Select **App registrations**, and then select **New registration**
   1. Name = MyWebAPI
   2. Supported account types = Accounts in this organizational directory only (Personal Directory only - Single tenant)
   3. Redirect URI = <Leave it as blank>
2. **Authentication** 🡪 Under Implicit Grant **check** = ID tokens
3. **Overview** 🡪 Application ID URI = <https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI>
4. **Expose an API** 🡪 + Add a Scope 🡪
   1. for **Scope name** use **user\_impersonation**
   2. Ensure the **Admins and users** option is selected for **Who can consent**
   3. in **Admin consent display name** type Access My Web API as a Admin
   4. in **Admin consent description** type Accesses the My Web API as a Admin
   5. in **User consent display name** type Access My Web API as a user
   6. in **User consent description** type Accesses the My Web API as a user
   7. Keep **State** as **Enabled**
   8. Select **Add scope**

**Secure the WebApp Application**

**Create a New AD Application for Web App**

1. Select **App registrations**, and then select **New registration**
   1. Name = MyWebApp
   2. Supported account types = Accounts in this organizational directory only (Personal Directory only - Single tenant)
   3. Redirect URI = https://localhost:**44331**/signin-oidc
2. **Authentication** 🡪
   1. Logout URL = https://localhost:44331/signout-oidc
   2. Under Implicit Grant **check** = ID tokens
3. **API Permissions** 🡪 Add a permission 🡪 My APIs 🡪 Select MyWebAPI 🡪 Check **user\_impersonation** 🡪 Add permission.
4. Certificate & **Secrets** 🡪 Create a copy Client Secret

**Update the WebAPI Project**

1. Add reference to NuGet Package - **Microsoft.Identity.Web**
2. Update appsettings.json

"AzureAd": {

  "Instance": "https://login.microsoftonline.com/",

  "Domain": "sandeepsonideccansoft.onmicrosoft.com",

  "TenantId": "82d8af3b-d3f9-465c-b724-0fb186cc28c7",

  "ClientId": "2f2e4e4e-aa57-4e07-83bd-a9853b55eb67"

  "Audience": "https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI"

}

1. To Starup.cs 🡪 ConfigureServices method add the following line

services.**AddMicrosoftIdentityWebApiAuthentication**(Configuration, "AzureAd");

1. Add the following to Configure method

app.UseAuthentication();

app.UseAuthorization();

1. A common requirement for web APIs is to validate the "scopes" present in the token to ensure that the user has consented to the permissions required to access the WebAPI.

[HttpGet]

[Authorize]

public IEnumerable<WeatherForecast> Get()

{

    var apiClaim = User.Claims.Where(c => c.Type == "http://schemas.microsoft.com/identity/claims/scope" && c.Value.Contains("user\_impersonation")).FirstOrDefault();

    if (apiClaim == null)

    {

        throw new ApplicationException("Unauthorized-The Scope claim does not contain 'user\_impersonation' or scope claim not found");

    }

    var rng = new Random();

    return Enumerable.Range(1, 5).Select(index => new WeatherForecast

    {

        Date = DateTime.Now.AddDays(index),

        TemperatureC = rng.Next(-20, 55),

        Summary = Summaries[rng.Next(Summaries.Length)]

    })

    .ToArray();

}

**Update the WebApp Project**

1. Add reference to NuGet packages
2. Microsoft.Identity.Web
3. Microsoft.Identity.Web.UI
4. Edit **appsettings**.**json**

"AzureAd": {

  "Instance": "https://login.microsoftonline.com/",

  "Domain": "sandeepsonideccansoft.onmicrosoft.com",

  "TenantId": "82d8af3b-d3f9-465c-b724-0fb186cc28c7",

  "ClientId": "06cfc5e9-b404-4bc1-90de-217a98a188b7",

  "CallbackPath": "/signin-oidc",

  "SignedOutCallbackPath": "/signout-oidc",

  "ClientSecret": "d2-62sab3\_T86yz0wWCl7XvDbWf2DH~-r\_"

}

1. Edit ConfigureServices in Startup.cs

services.AddMicrosoftIdentityWebAppAuthentication(Configuration, "AzureAd")

          .EnableTokenAcquisitionToCallDownstreamApi(new string[] { "https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI/user\_impersonation" })

          .AddInMemoryTokenCaches();

services.AddControllersWithViews().**AddMicrosoftIdentityUI();**

1. Add the following to Configure method

app.UseAuthentication();

app.UseAuthorization();

1. Edit HomeController as below

public class HomeController : Controller

{

    IConfiguration \_config;

    readonly ITokenAcquisition \_tokenAcquisition;

    public HomeController(ILogger<HomeController> logger, IConfiguration config,  ITokenAcquisition tokenAcquisition)

    {

        \_config = config;

        \_tokenAcquisition = tokenAcquisition;

    }

    public IActionResult Index()

    {

        return View();

    }

    [AuthorizeForScopes(Scopes = new[] { "https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI/user\_impersonation" })]

    public async Task<IActionResult> WeatherForecast()

    {

        HttpClient httpClient = new HttpClient();

        string[] scopes = new string[] { "https://sandeepsonideccansoft.onmicrosoft.com/MyWebAPI/user\_impersonation" };

        string token = await \_tokenAcquisition.GetAccessTokenForUserAsync(scopes);

        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);

        string url = \_config["ApiUrl"] + "weatherforecast";

        var request = new HttpRequestMessage()

        {

            RequestUri = new Uri(url)

        };

        IEnumerable<WeatherForecast> cols = null;

        using (var response = httpClient.SendAsync(request).Result)

        {

            cols = JsonConvert.DeserializeObject<IEnumerable<WeatherForecast>>(response.Content.ReadAsStringAsync().Result);

        };

        return View(cols);

    }

}

**(Optional) Pre-authorize your client application**

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

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

### Role-based authorization

Role-based authorization is an authorization approach in which user permissions are managed and enforced by an application based **on user roles**. If a user has a role that is required to perform an action, access is granted; otherwise, access is denied. When an identity is created, it may belong to one or more roles. For example, Holly may belong to the Administrator and User roles, whereas Adam may belong only to the User role. How these roles are created and managed depends on the backing store of the authorization process.

**Implementing Roles using Azure AD App Roles**

1. The SaaS (Security as a Service) provider defines the application roles by adding them to the application manifest.
2. After a customer signs up, an admin for the customer's AD directory assigns users to the roles.
3. When a user signs in, the user's assigned roles are sent as claims.

**Advantages of this approach:**

* Simple programming model.
* Roles are specific to the application. The role claims for one application are not sent to another application.
* If the customer removes the application from their AD tenant, the roles go away.
* The application doesn't need any extra Active Directory permissions, other than reading the user's profile.

**Drawbacks:**

* Customers without Azure AD Premium cannot assign security groups to roles. For these customers, all user assignments must be done by an AD administrator.
* If you have a backend web API, which is separate from the web app, then role assignments for the web app don't apply to the web API.

**Steps To Implement**

Azure AD 🡪 **App registrations** 🡪 Select the app 🡪 **Manifest** 🡪 Edit the Manifest (Search “appRoles” and edit)

"appRoles": [

{

"allowedMemberTypes": [

"User"

],

"description": "This is Demo Role1",

"displayName": "DemoRole1",

"id": "1b4f816e-5eaf-48b9-8613-7923830595ty",

"isEnabled": true,

"value": "DemoRole1"

},

{

"allowedMemberTypes": [

"User"

],

"description": "This is Demo Role2",

"displayName": "DemoRole2",

"id": "c20e145e-5459-4a6c-a074-b942bbd4cab1",

"isEnabled": true,

"value": "DemoRole2"

}

],

Note: The value property appears in the role claim. The id property is the unique identifier for the defined role. Always generate a new GUID value for id.

**Assign users**.

When a new customer signs up, the application is registered in the customer's AD tenant. At this point, an AD admin for that tenant can assign users to roles.

Azure AD **🡪 Enterprise Application** 🡪 Select the Application 🡪 **Users and groups** 🡪 Add user 🡪 Select User and **Select Role**

#### **Role-Based authorization in ASP.NET**

**Get role claims**. When a user signs in, the application receives the user's assigned role(s) in a claim with type http://schemas.microsoft.com/ws/2008/06/identity/claims/role **OR** ClaimsTypes.Role

Roles are exposed to the developer through the **IsInRole** method on the **ClaimsPrincipal** class. Role-based Instead, write code that checks whether a particular claim value is present:

bool isHavingDemoRole1 = User.**IsInRole**("DemoRole1").ToString()

OR

bool isHavingDemoRole1 = ((ClaimsPrincipal)User).**HasClaim**(ClaimTypes.Role, "DemoRole1")

**Authorization checks are declarative**—the developer embeds them within their code, against a controller or an action within a controller, specifying roles that the current user must be a member of to access the requested resource.

For example, the following code limits access to any actions on the DemoController to users who are members of the **DemoRole1** role:

[Authorize(**Roles** = "DemoRole1")]

public class DemoController : Controller { }

You can specify multiple roles as a comma separated list:

[Authorize(**Roles** = "DemoRole1, DemoRole2")]

public class DemoController : Controller { }

This controller would be accessible only by users who are members of the **DemoRole1** role or the **DemoRole2** role.

If you apply multiple attributes, an accessing user must be a member of all the roles specified. The following sample requires that a user be a member of both the **DemoRole1** and **DemoRole2** roles:

[Authorize(**Roles** = "**DemoRole1**")]

[Authorize(**Roles** = "**DemoRole2**")]

public class ControlPanelController : Controller { }

You can further limit access by applying additional role authorization attributes at the action level:

**[Authorize(Roles = "DemoRole1, DemoRole2")]**

public class ControlPanelController : Controller

{

public ActionResult SetTime()

{ }

**[Authorize(Roles = "DemoRole2")]**

public ActionResult ShutDown()

{ }

}

In the previous code snippet, members of either the **DemoRole1** role or the **DemoRole2** role can access the controller and the SetTime action, but only members of the **DemoRole2** role can access the ShutDown action.

You can also lock down a controller but allow anonymous, unauthenticated access to individual actions:

[Authorize]

public class ControlPanelController : Controller

{

public ActionResult SetTime()

{ }

**[AllowAnonymous]**

public ActionResult Login()

{ }

}

**Policy Syntax**

In ASP.NET Core Role requirements can also be expressed using the **Policy** syntax, where a developer registers a policy at startup as part of the authorization service configuration. This normally occurs in ConfigureServices() in your Startup.cs file:

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.AddAuthorization(options => {

options.AddPolicy("RequireAdministratorRole", policy => policy.RequireRole("Administrator"));

});

}

Policies are applied using the Policy property on the AuthorizeAttribute attribute:

[Authorize(Policy = "RequireAdministratorRole")]

public IActionResult Shutdown()

{

return View();

}

If you want to specify multiple allowed roles in a requirement, you can specify them as parameters to the RequireRole method:

options.AddPolicy("ElevatedRights", policy => policy.RequireRole("Administrator", "PowerUser", "BackupAdministrator"));

This example authorizes users who belong to the **Administrator**, **PowerUser**, or **BackupAdministrator** roles.

## Roles using an application role manager

With this approach, application roles are not stored in Azure AD at all. Instead, the application stores the role assignments for each user in its own DB — for example, using the **RoleManager** class in **ASP.NET Identity**.

**Advantages:**

* The app has full control over the roles and user assignments.

**Drawbacks:**

* More complex, harder to maintain.
* Cannot use AD security groups to manage role assignments.
* Stores user information in the application database, where it can get out of sync with the tenant's AD directory, as users are added or removed.

**Claims Based Authorization**

**What is Claims?**

* When an identity is created, it may be assigned one or more claims issued by a trusted party. A claim is a name/value pair that represents what the subject is and not what the subject can do. For example, you may have a driver's license issued by a local driving license authority. Your driver's license has your date of birth on it. In this case, the claim name would be DateOfBirth, the claim value would be your date of birth — for example, June 8, 1970 — and the issuer would be the driving license authority. An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.
* **Claims-based authorization** is an approach where the authorization decision to grant or deny access is based on arbitrary logic that uses data available in claims to make the decision. Claims-based authorization, at its simplest, checks the value of a claim and allows access to a resource based on that value. For example, if you want access to a night club, the authorization process might be: The door security officer evaluates the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.
* In a relying party application, authorization determines what resources an authenticated identity is allowed to access and what operations it is allowed to perform on those resources. Improper or weak authorization leads to information disclosure and data tampering.
* Claim-based authorization checks are declarative—the developer embeds them within their code, against a controller or an action within a controller, specifying claims that the current user must possess and optionally the value the claim must hold to access the requested resource. Claims requirements are policy based; the developer must build and register a policy expressing the claims requirements.

#### **Claims-based authorization in Microsoft ASP.NET**

To get the list of claims in the authenticated request:

foreach (var claim in ((System.Security.Claims.ClaimsIdentity)User.Identity).Claims)

{

ViewBag.Message += claim.Type + ":" + claim.Value + "<br>";

}

bool hasRoleClaim = ((ClaimsPrincipal)User).**HasClaim**(c => c.Type == ClaimTypes.Role);

OR

bool isHavingDemoRole1 = ((ClaimsPrincipal)User).**HasClaim**(ClaimTypes.Role, "DemoRole1")

**Using Policy in ASP.NET Core**

The simplest type of claim policy looks for the presence of a claim and doesn't check the value.

First, you need to build and register the policy. This takes place as part of the authorization service configuration, which normally takes place in **ConfigureServices()** in your **Startup.cs** file:

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.**AddAuthorization**(options => {

options.**AddPolicy**("issuer", policy => policy.RequireClaim("iss"));

});

}

In this case, the **issuer** policy checks for the presence of an **iss** claim on the current identity. You then apply the policy using the **Policy** property on the

**AuthorizeAttribute** attribute to specify the policy name:

[Authorize(Policy = "issuer")]

public IActionResult **VacationBalance**() { return View(); }

If you have a controller that's protected by the **AuthorizeAttribute** attribute but want to allow anonymous access to particular actions, you apply the **AllowAnonymousAttribute** attribute:

[Authorize(Policy = "EmployeeOnly")]

public class VacationController : Controller

{

public ActionResult VacationBalance()

{ }

**[AllowAnonymous]**

public ActionResult VacationPolicy()

{ }

}

Most claims come with a value. You can specify a list of allowed values when creating the policy. The following example succeeds only for employees whose employee number is 1, 2, 3, 4 or 5:

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.AddAuthorization(options => {

options.AddPolicy("ValidIssuers", policy => policy.RequireClaim("iss",

"https://sts.windows.net/1d3819df-19f4-4814-9599-bf22f14fcdbe/",

"https://sts.windows.net/1d3819df-19f4-sdfds-9599-bf22f14fcdbe/"));

});

}

[Authorize(Policy = "ValidIssuers")]

public void Foo()

{ }