

Authentication, token validation and authorization

Last updated by | Jegadish Purushothaman | Mar 25, 2021 at 2:45 PM MDT

POH Authentication, Token Validation and Authorization

This document describes Platform of Health (POH) authentication, token validation (JWT type access token issued by Azure AD Enterprise) and authorization (granular level user permissions) for Web APIs. Azure AD supports Open ID Connect, OAuth 2.0 and SAML 2.0 protocols. POH mobile, web and API apps implement these protocols for authentication, token validation and authorization. POH web APIs are protected resources. The implementation would be using Microsoft .NET Standard 2.0 and ASP.NET Core 3.1 frameworks along with Microsoft Identity Platform.

Revision Summary

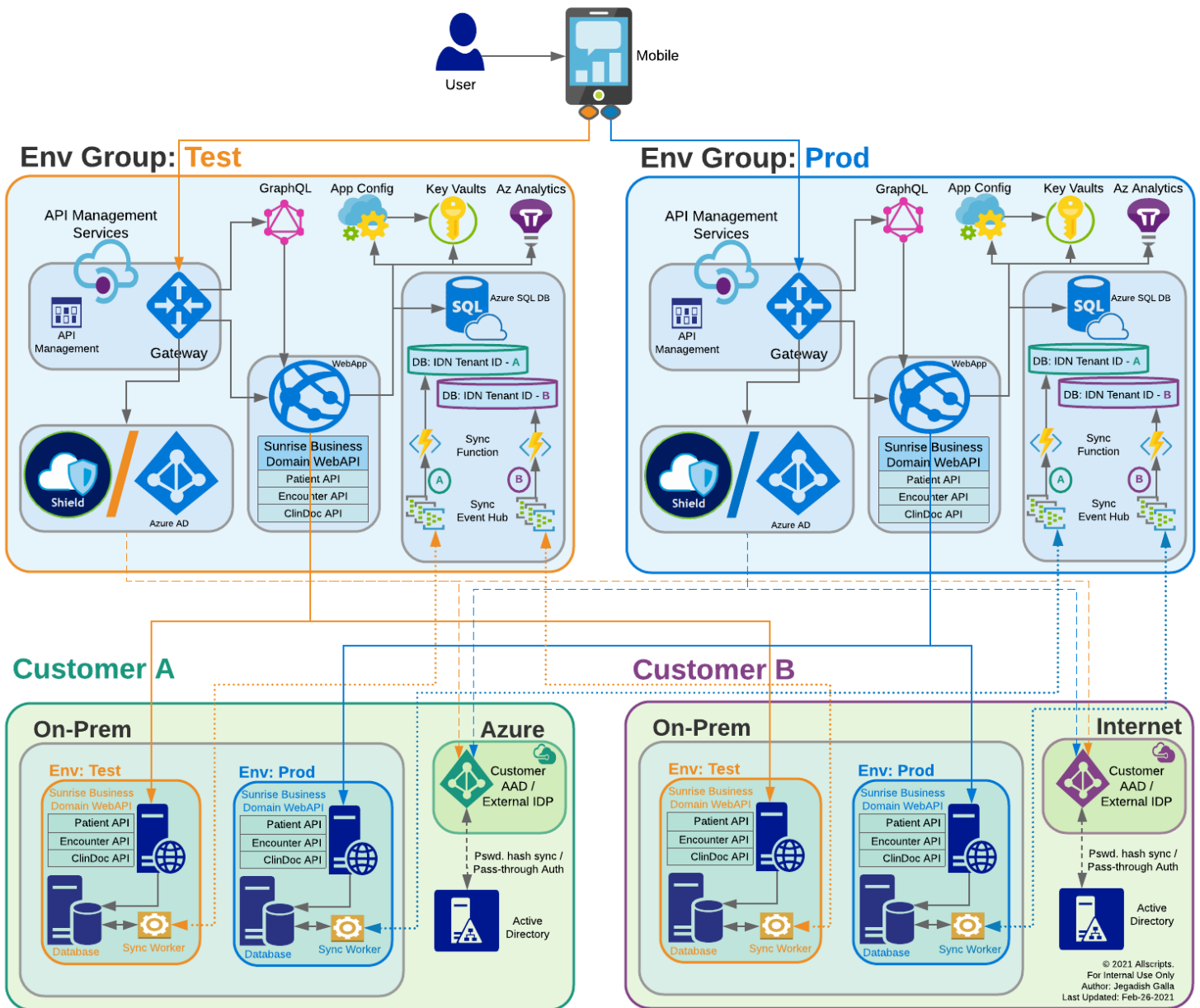
Date	Version	Summary of Changes
02/15/2021	0.1	Initial Draft. Platform of Health Authentication, Token Validation & Authorization library for apps deployed in Azure and changes in existing Sunrise apps deployed on-premise.
2/27/2021	0.2	Add section for Token Acquisition and notes about common Client ID for PoH Web APIs.
3/09/2021	0.3	Call out Env group selection and IDN Tenant Selection and remove references to B2C

Authentication

The mobility app is using Azure AD for authentication. The mobility app will use Microsoft Authentication Library (MSAL), a part of Microsoft Identity Platform, to get an access token and call POH Web APIs.

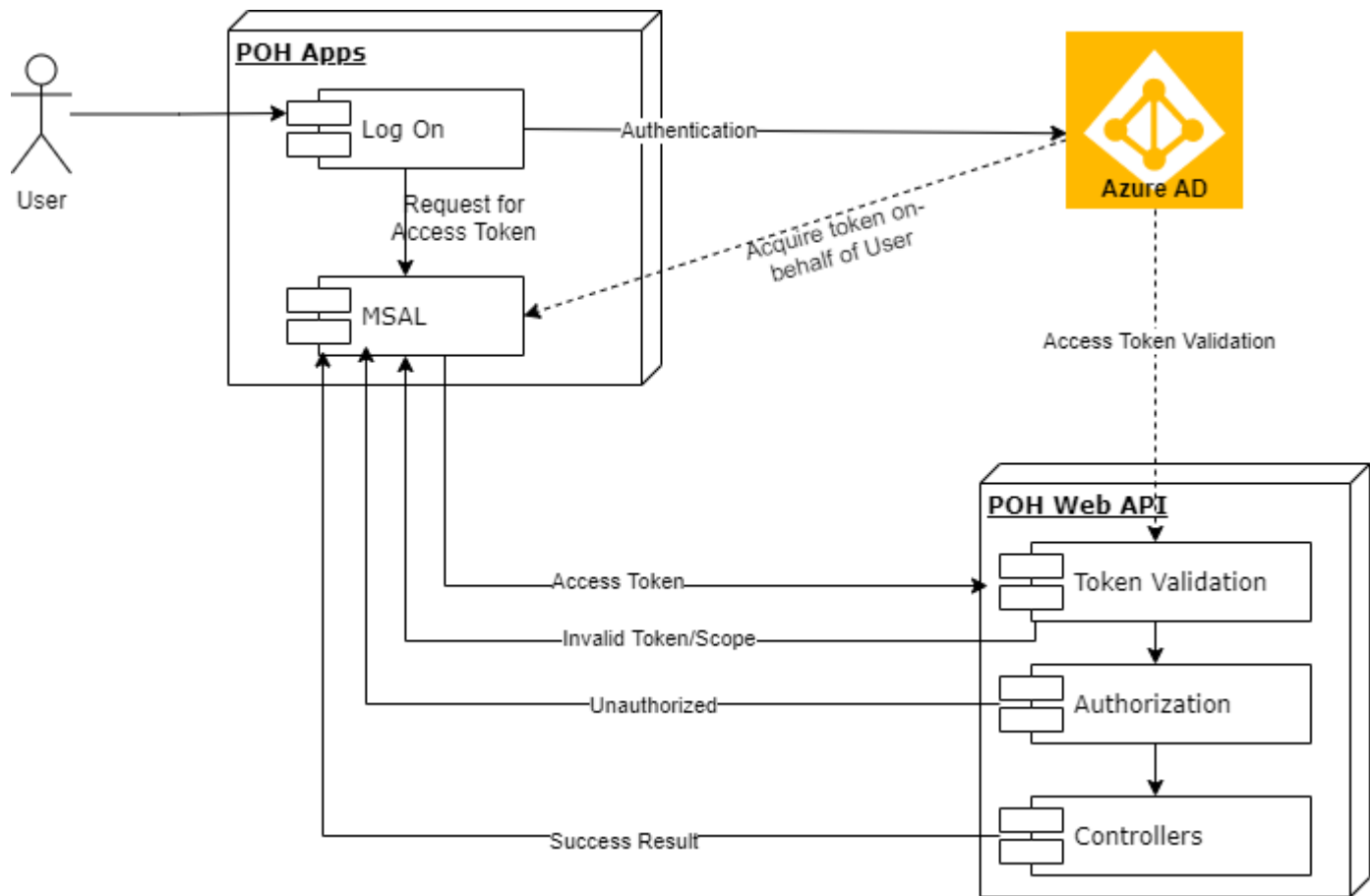
The authentication in existing Sunrise apps would continue without change with Enterprise/Helios Active Directory.

Sunrise POH Architecture



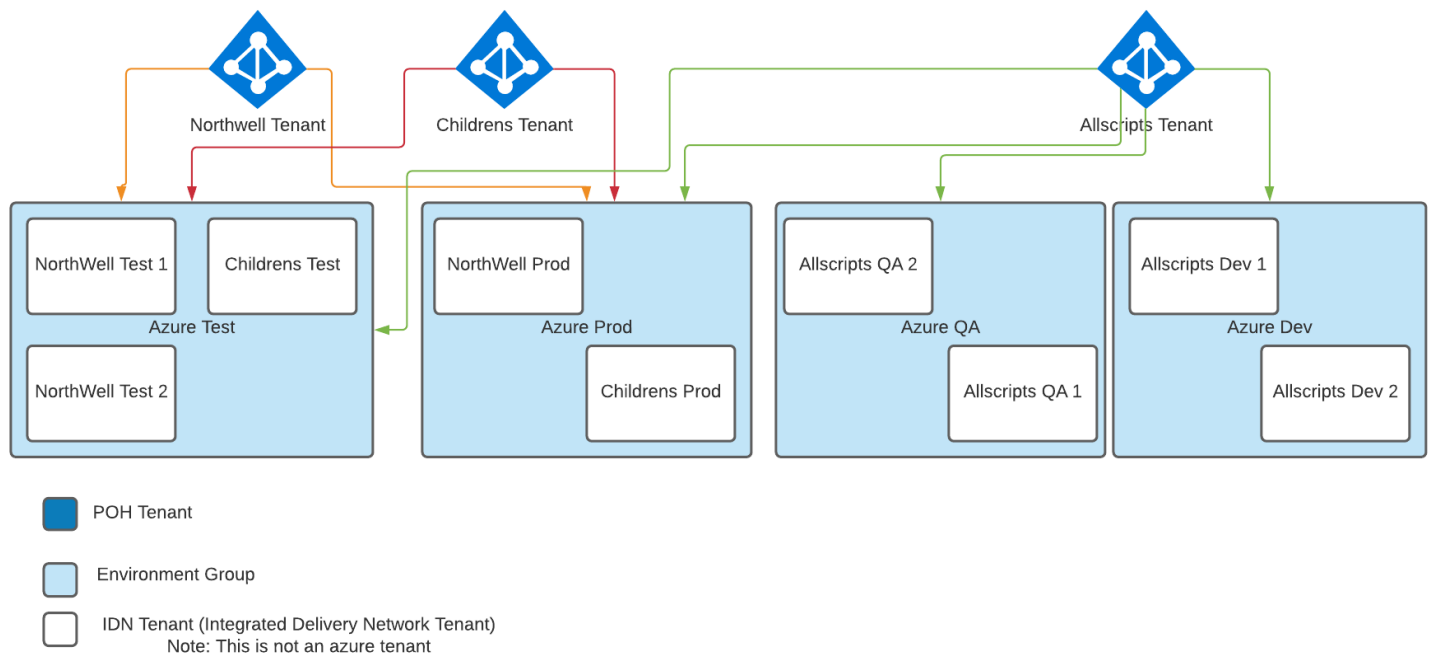
Component Design

The block diagram below shows the interactions amongst authentication, token validation and authorization:



Environment Selection Flow

An End user would typically have access to only one environment. But for testing and support, there will be cases where we would need user to logon to different environments. Typically a Prod environment group will be its own subscription and will have its AAD instance. A large customer like Northwell might require multiple test IDNTenants. And an Allscripts support staff would have access to all of the environments. The diagram below shows how multi-tenancy will look like for POH in such a scenario

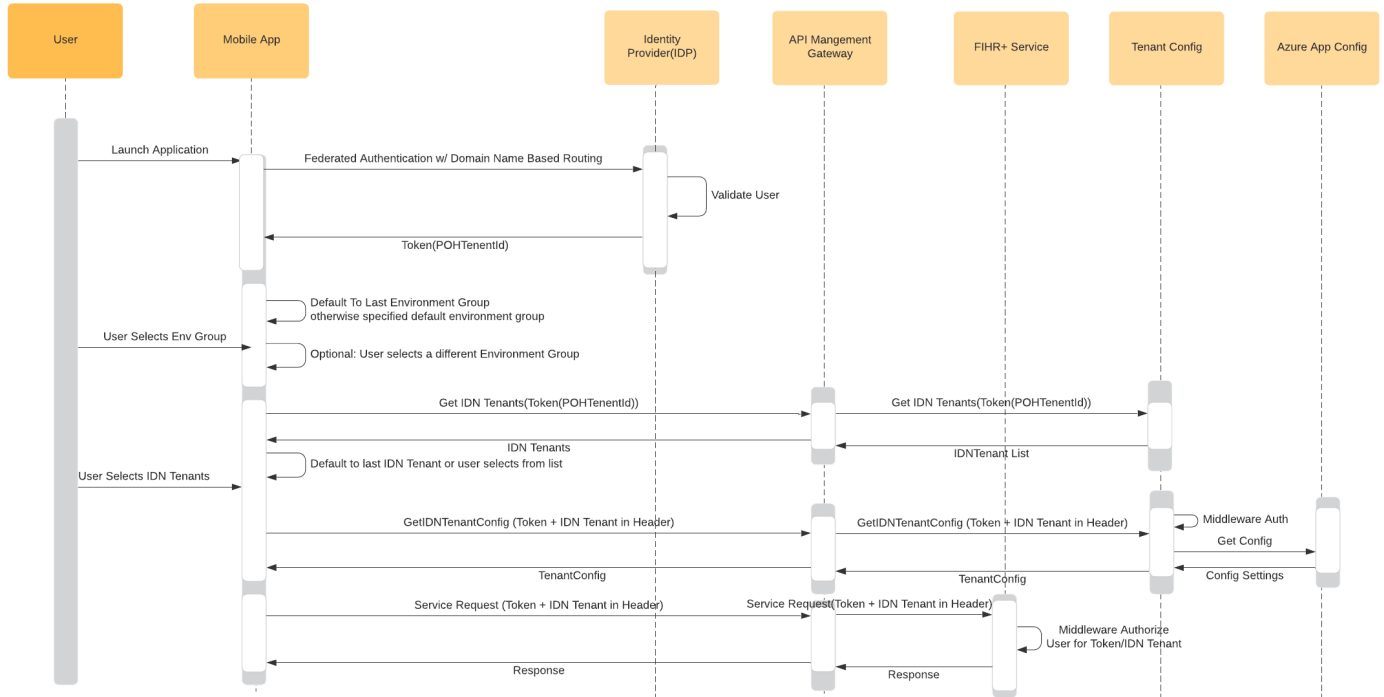


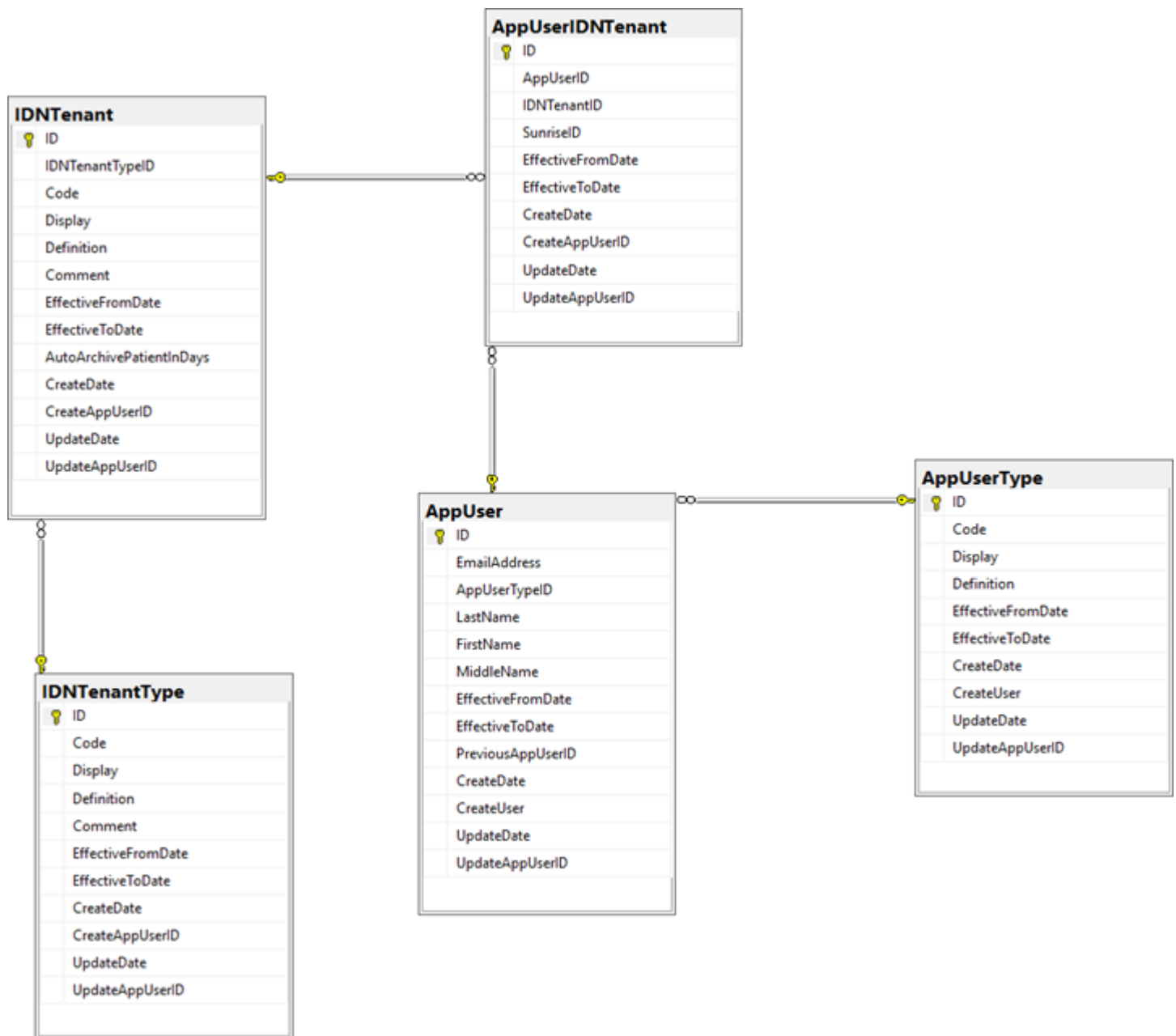
The following sequence diagram starts where a user will select the Environment group. This is optional for majority of the users as they will be hitting production environment only. So a typical user will directly go to logon screen and log in using Azure AD. Once the identity is validated, a Token is returned by the AAD for the user. The app then fetches all the valid IDNTenant that the user has access to. If it's only one IDNTenant, the app starts connecting to it. if a user has multiple IDNTenant access, we will ask user to select one. Once the user selects the IDN Tenant, all the service requests will have IDN Tenant as a Tenant parameter. We could also default to the last logged on environment so that the user does not have to select all IDN Tenant all the time

The proposed schema is shown in the ER diagram below the sequence diagram

Mobile Client Auth Flow

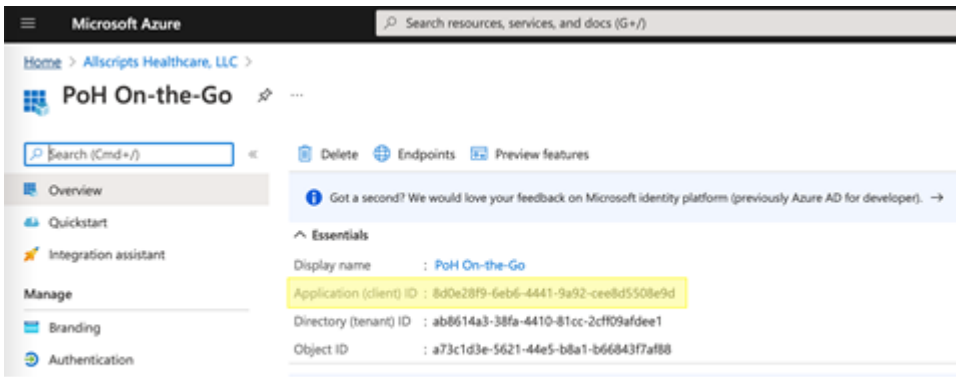
Keith Langreck | March 11, 2021





Token Acquisition

In order for the PoH On-the-Go native mobile client app needs to acquire an access token to be able to access PoH Web APIs on behalf of the logged in user, the mobile client app must first reach out to Azure AD to acquire an access token. The mobile client app must use the client id of the application registration in Azure AD for PoH On-the-Go native mobile client app.



When requesting an access token, you can only request an access token for a single resource at a time, so any scopes must be for the same resource. In this case, the resource is the PoH Web APIs so the scope will be registered with the PoH Web APIs.

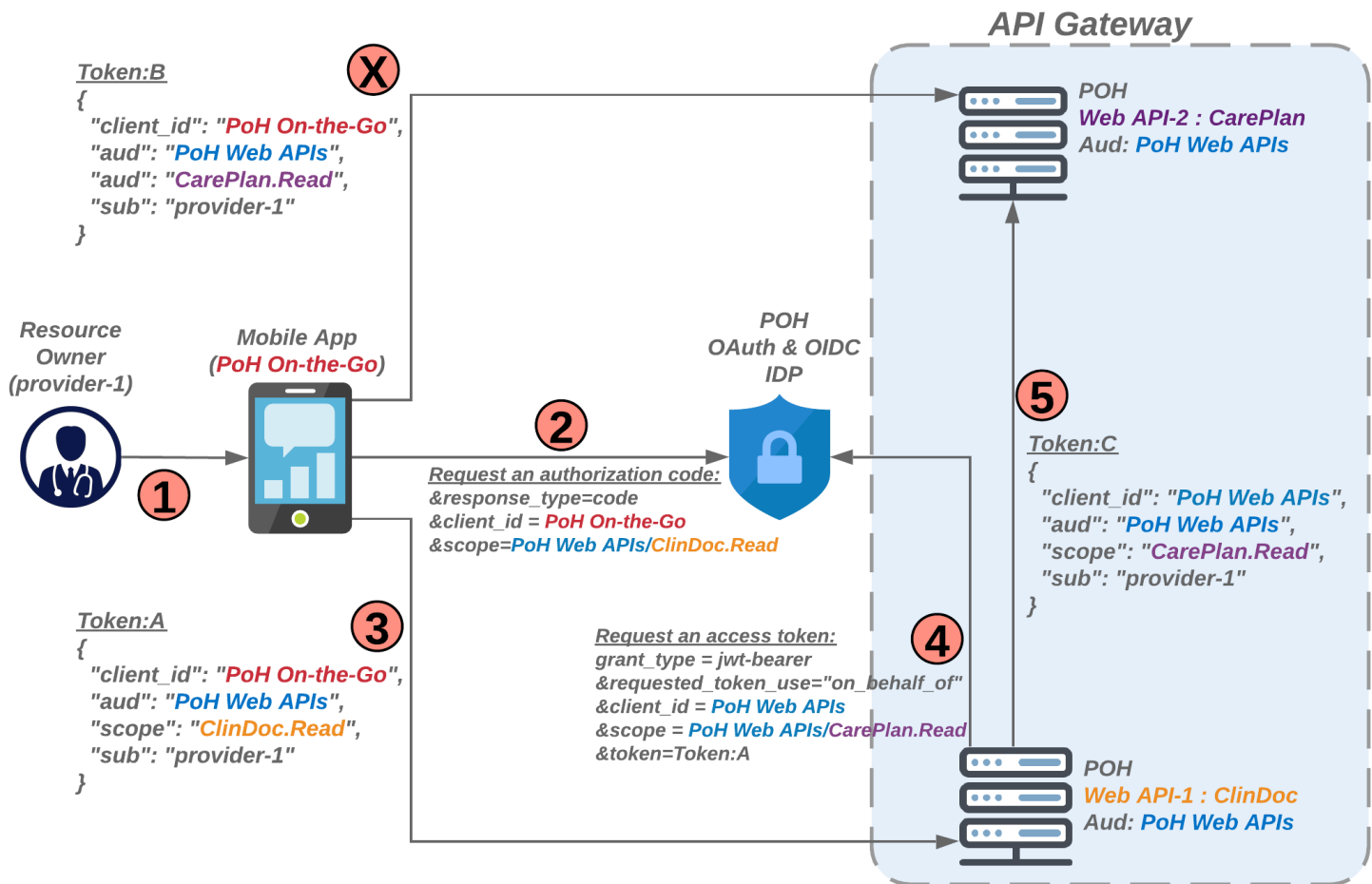
```
authResult = await App.PCA.AcquireTokenInteractive(App.Scopes)
    .WithParentActivityOrWindow(App.ParentWindow)
    .ExecuteAsync();
```



Token Exchange

Below diagram represents how token exchange between client application (Mobile) and Web API deployed in Azure and exposed through Azure API Gateway works. The diagram also shows how OAuth 2.0 On-Behalf-Of flow works (Token Exchange), when a Web API in turn calls a downstream Web API.

TokenExchange: OAuth 2.0 On-Behalf-Of flow (OBO)



Token Validation

POH Web APIs are protected resources. Client apps will acquire Access Token to call Web APIs on behalf of the logged in users. All the requests must be authenticated and should carry a valid Access Token issued by Azure AD to call POH Web APIs. The Access Token is a JSON Web Token (JWT) passed in "Authorization" header of the request as Bearer token.

To provide restricted access to POH Web APIs:

- Register POH Web API in Azure AD
- Note that all PoH Web APIs can share a common audience and so there is a single Application Registration for PoH Web APIs:

Microsoft Azure

Home > Allscripts Healthcare, LLC > PoH Web APIs

Search (Cmd+/)

Overview

Quickstart

Integration assistant

Manage

Branding

Authentication

Got a second? We would love your feedback on Microsoft identity platform

Essentials

Display name : PoH Web APIs

Application (client) ID : 4fa986d7-2121-4766-ab04-2b25177fd51a

Directory (tenant) ID : ab8614a3-38fa-4410-81cc-2cff09afdee1

Object ID : 37dcc283-cd67-495c-a0bb-925e553ef217

- Create appropriate Scopes under “Expose an API” option
- Configure API Permissions, Authentication in Azure portal.
- In Manifest option, set “accessTokenAcceptedVersion”: 2. This ensures the Web API can be called from multi-tenant authentication. By default, the value is null that means the API can be called from the same tenant.

Home > Azure AD B2C > FhirEncounter

FhirEncounter | Manifest

Search (Ctrl+/)

Save Discard Upload Download Got feedback?

Overview

Integration assistant

Manage

Branding

Authentication

Certificates & secrets

API permissions

Expose an API

Owners

Manifest

Support + Troubleshooting

Troubleshooting

The editor below allows you to update this application by directly modifying its JS

```

1 {
2   "id": "XXXXXXXXXX",
3   "acceptMappedClaims": null,
4   "accessTokenAcceptedVersion": 2,
5   "addIns": [],
6   "allowPublicClient": null,
7   "appId": "XXXXXXXXXX",
8   "appRoles": [],
9   "oauth2AllowUrlPathMatching": false,
10  "createdDateTime": "2021-02-09T10:31:19Z",
11  "disabledByMicrosoftStatus": null,
12  "groupMembershipClaims": null,
13  "identifierUris": [
14    "https://ridesharemdrxtenant1.onmicrosoft.com/f"
15  ],
16  "informationalUrls": {
17    "termsOfService": null,
18    "support": null,
19    "privacy": null
20  }
21 }
```

To validate all incoming requests by POH Web API:

- The Web APIs should validate all incoming requests to check whether the access token in “Authorization” header is valid.
- Configure Azure AD details of the registered Web API in appsettings.json file. The sensitive information can be stored in Azure Key Vault.
- A new middleware “POH.Infrastructure.Security.TokenValidation” implements “[Microsoft.Identity.Web](#)” to validate Access Token. Add this middleware in ConfigureServices() method of Startup.cs class of POH Web API before any controller method is called.

Example:

```
services.AddJwtTokenValidation(configuration).
```

This middleware provides abstraction to the token validation provided by [Microsoft.Identity.Web](#) library and the services pipeline is like:

```
services.AddMicrosoftIdentityWebApiAuthentication(configuration, "AzureAd")
    .EnableTokenAcquisitionToCallDownstreamApi()
    .AddDownstreamWebApi("SunriseTest01", configuration.GetSection("DownstreamAPI_01"))
    .AddInMemoryTokenCaches();
```

<https://github.com/AzureAD/microsoft-identity-web>

- Microsoft JwtBearer middleware

```
services.AddMicrosoftIdentityWebApiAuthentication(configuration, "AzureAd")
```

OR

```
services.AddAuthentication(AzureADDefaults.JwtBearerAuthenticationScheme)
    .AddMicrosoftIdentityWebApi(Configuration, "AzureAd");
```

are the same. One should not be confused with different extension methods.

- Add "Authorize" attribute to Web APIs controllers at class level or method level (as appropriate).
- In Configure() method of Startup.cs class add app.UseAuthentication(); and app.UseAuthorization(); before calling app.UseEndpoints().
- Exceptions/Errors: The token validation middleware will throw *Status Code 401 - "Unauthorized"* or *403 - "Forbidden"* if the token is invalid or user does not have permission to access that Web API. A list of REST API error code with description can be found:

<https://docs.microsoft.com/en-us/rest/api/storageservices/common-rest-api-error-codes>

- Run the token validation middleware first before executing the POH Authorization middleware.

NOTE: [Microsoft.Identity.Web](#) is .NET Core 3.1 compatible library.



POH.Infrastructure.Security.TokenValidation will be .NET Core 3.1 library. This cannot be referred in .NET Standard 2.0 or .NET Full framework applications.

One Vision Token Validation Requirements


The product implements the following industry standard security controls for identity and access management:

One Vision Requirements	Microsoft Identity TokenValidationParameters
Security Token Issuer Identity	Issuer: It was issued by a trusted security token service (STS)
Security Token Issuer Trust	Issuer: It was issued by a trusted security token service (STS)
Security Token Signature validity	Signature: It wasn't tampered with.
Security Token Certificate trust	Certificate: Certificate trust is not validated in Web APIs when the Web API is working as service. This parameter is validated when acquiring Access Token. Certificate trust would be validated for On-Behalf_Of workflow i.e. when a Web API calls a downstream Web API.
Security Token Certificate validity	Certificate: Certificate validity is not validated in Web APIs when the Web API is working as service. This parameter is validated when acquiring Access Token. Certificate trust would be validated for On-Behalf_Of workflow i.e. when a Web API calls a downstream Web API.
Security Token Timeframe/Expiration	Expiry: Its lifetime is in range.
Security Token Audience	Audience: The token is targeted for the web API.

Customizing Token Validation

The validators are associated with properties of the `TokenValidationParameters` class. The properties are initialized from the [ASP.NET](#)  and [ASP.NET](#)  Core configuration.

In most cases, you don't need to change the parameters. [Microsoft.Identity.Web](#) takes care of Issuer validation as well for multi-tenant apps.

If you want to write custom validation for the parameters where validation is not done by [Microsoft.Identity.Web](#) , you can do as below example code in Token Validation middleware.

```


services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddMicrosoftIdentityWebApi(Configuration);
services.Configure<JwtBearerOptions>(JwtBearerDefaults.AuthenticationScheme, options =>
{
    var existingOnTokenValidatedHandler = options.Events.OnTokenValidated;
    options.Events.OnTokenValidated = async context =>
    {
        await existingOnTokenValidatedHandler(context);
        // Your code to add extra configuration that will be executed after the current event implementation.
        options.TokenValidationParameters.ValidIssuers = new[] { /* list of valid issuers */ };
        options.TokenValidationParameters.ValidAudiences = new[] { /* list of valid audiences */ };
        ....
    }
});

```

Integrated Delivery Network (IDN) Tenant Selection

After the user selects the Environment Group (for development builds) and authenticates to Azure AD, the user will be shown a list of IDN tenants for which the user has rights to access. The IDN tenant selected will determine the database and on-prem resources that PoH will pull data from and send down to the native app.

Refer Microsoft documentation:

<https://docs.microsoft.com/en-us/azure/active-directory/develop/scenario-protected-web-api-app-configuration> 

Microsoft identity platform best practices and recommendations

Go through the below recommendations and best practices for Microsoft identity platform.


<https://docs.microsoft.com/en-us/azure/active-directory/develop/identity-platform-integration-checklist> 

References (Token Validation):

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

<https://github.com/AzureAD/microsoft-identity-web> 

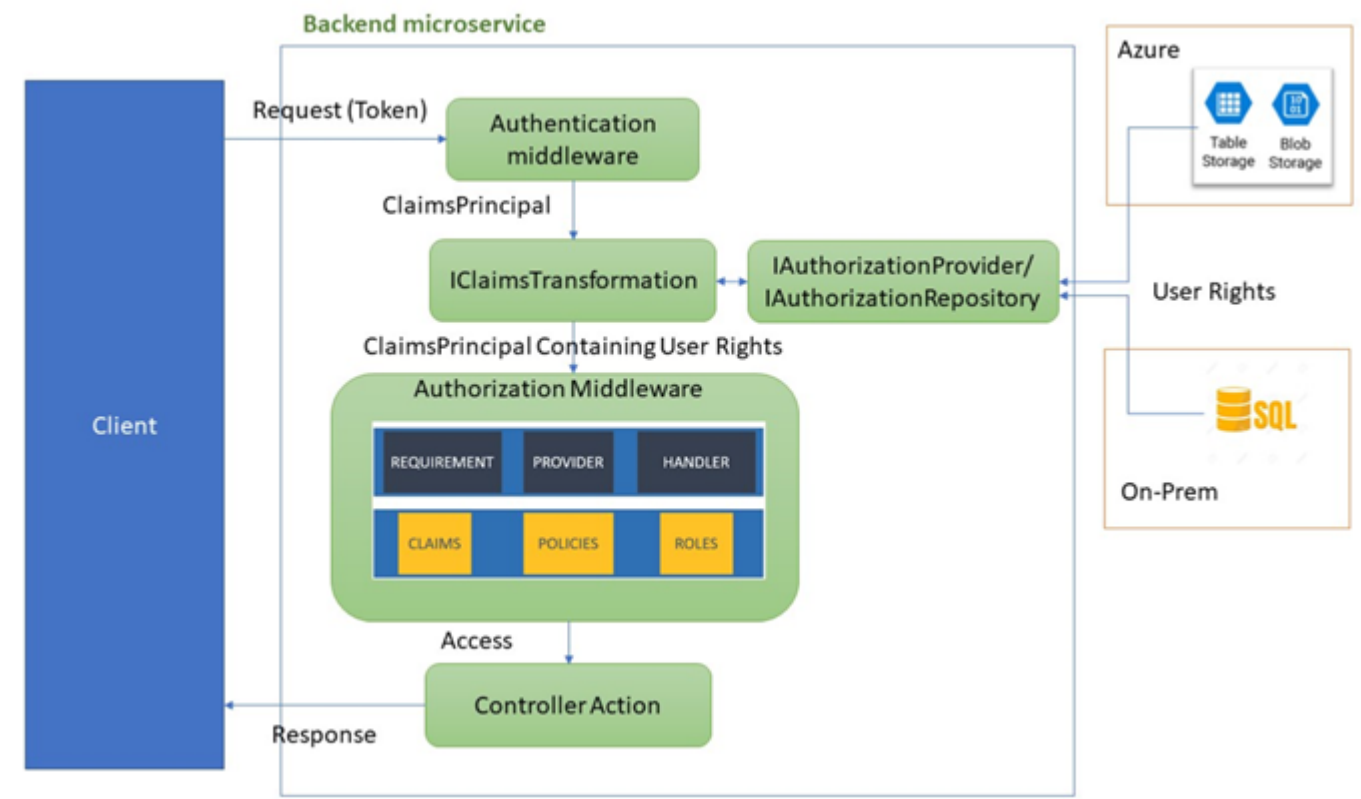
<https://docs.microsoft.com/en-us/azure/active-directory/develop/identity-platform-integration-checklist> 

<https://docs.microsoft.com/en-us/azure/active-directory/develop/scenario-protected-web-api-app-configuration> 

<https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet/wiki/ValidatingTokens> 

Authorization

POH Authorization middleware is industry standard, policy-based Authorization implementation using Microsoft ASP.NET Core Authorization library. This should be developed using Microsoft .NET Standard 2.0 framework so that it can be used in both .NET Standard and .NET Core projects.



Policy Based Authorization

A policy-based security model decouples authorization and application logic and provides a flexible, reusable, and extensible security model in .NET Core. The policy-based security model is centered on three main concepts. These include policies, requirements, and handlers. A policy is comprised of several requirements. A requirement, in turn, contains data parameters to validate the user's identity. A handler is used to determine if a user has access to a specific resource. The result is a richer, reusable, testable authorization structure.

Requirement

An authorization requirement is a collection of data parameters that a policy can use to evaluate the current user principal. A requirement implements `IAuthorizationRequirement`. If an authorization policy contains multiple authorization requirements, all requirements must pass for the policy evaluation to succeed. A requirement doesn't need to have data or properties.

Authorization (Requirement) Handler

The authorization handler evaluates the requirements against a provided `AuthorizationHandlerContext` to determine if access is allowed. A handler may inherit `AuthorizationHandler<TRequirement>`, where `TRequirement` is the requirement to be handled.

A requirement can have multiple handlers. Implement `IAuthorizationHandler` to implement more than one type of requirements.

Register Authorization Handler

Register all handlers in `POH.Infrastructure.Security.Authorization` middleware or `Startup.ConfigureServices` method.

Example code snippet:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });
    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
}
```

- A handler indicates success by calling `context.Succeed(IAuthorizationRequirement requirement)`, passing the requirement that has been successfully validated.
- A handler doesn't need to handle failures generally, as other handlers for the same requirement may succeed.
- To guarantee failure, even if other requirement handlers succeed, call `context.Fail`.

If a handler calls `context.Succeed` or `context.Fail`, all other handlers are still called. This allows requirements to produce side effects, such as logging, which takes place even if another handler has successfully validated or failed a requirement. When set to false, the `InvokeHandlersAfterFailure` property short-circuits the execution of handlers when `context.Fail` is called. `InvokeHandlersAfterFailure` defaults to true, meaning authentication handlers are invoked after failure.

NOTE: Authorization handlers are called even if authentication fails.

You can use a “func” to fulfill multiple policy having AND or OR condition. Example:
Example code snippet:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("BadgeEntry", policy =>
        policy.RequireAssertion(context =>
            context.User.HasClaim(c =>
                (c.Type == "BadgeId" ||
                 c.Type == "TemporaryBadgeId") &&
                 c.Issuer == "https://allscriptssecurity"))));
});
```

Policy Implementation

An authorization policy consists of one or more requirements. You can create a policy instance using the `AuthorizationPolicyBuilder` class. Then register policies by specifying policy names. This policy name can be applied in controllers or action methods by `AuthorizeAttribute` or `AuthorizeFilter`.

Claims Based Authorization via Policies

Claims based authorization provides a declarative way of checking access to resources. In this type of Authorization, you grant access to resource based on the value contained in the claim. A claim is given to a trusted party. A claim has nothing to do with what a subject can do. Refer the example.

Example code snippet:

1.

```
services.AddAuthorization(options =>
{
    options.AddPolicy("ShouldBeOnlyEmployee", policy =>
        policy.RequireClaim("EmployeeId"));
});
```

The code snippet given below shows how a simple claim policy is registered. The "ShouldBeOnlyEmployee" policy looks for the presence of the "EmployeeId" claim. The policy is added to the services collection using the "AddPolicy" method.

2.

```
services.AddAuthorization(options =>
{
    options.AddPolicy("AtLeast21", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(21)));
});
```

Apply this policy at the controller level on the "AuthorizeAttribute" attribute as shown below.

```
[Authorize(Policy = "ShouldBeOnlyEmployee")]
public IActionResult SomeMethod()
{
    // your code here
}
```

You can have policies with multiple claims as well. You should register them appropriately in the `ConfigureServices` method of the Startup class as shown in the code snippet given below.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(
        CompatibilityVersion.Version_2_1);
    services.AddAuthorization(options =>
    {
        options.AddPolicy("CustomSecurityPolicy", policy =>
            policy.RequireClaim("ShouldBeOnlyEmployee"));
        options.AddPolicy("CustomSecurityPolicy", policy =>
            policy.RequireClaim("IsAdmin", "true"));
    });
}
```

The primary service that determines if authorization is successful is `IAuthorizationService`

References:

<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies?view=aspnetcore-3.1> 

<https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authorization.iauthorizationservice?view=aspnetcore-3.1> 

<https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authorization.iauthorizationhandler?view=aspnetcore-3.1> 