IdentityModel Documentation

Dominick Baier and Brock Allen

Feb 05, 2020

IdentityModel

IdentityModel is a family of libraries for building OAuth 2.0 and OpenID Connect clients.

CHAPTER 1

IdentityModel

The base library for OIDC and OAuth 2.0 related protocol operations. It also provides useful constants and helper methods.

Currently we support .NET Standard 2.0 / .NET Framework > 4.6.1

- github https://github.com/IdentityModel/IdentityModel
- nuget https://www.nuget.org/packages/IdentityModel/
- CI builds https://github.com/orgs/IdentityModel/packages

The following libraries build on top of IdentityModel, and provide specific implementations for different applications:

CHAPTER 2

IdentityModel.AspNetCore

ASP.NET Core specific helper library for token management.

- github https://github.com/IdentityModel/IdentityModel.AspNetCore
- nuget https://www.nuget.org/packages/IdentityModel.AspNetCore/
- CI builds https://github.com/orgs/IdentityModel/packages

CHAPTER 3

IdentityModel.AspNetCore.OAuthIntrospection

OAuth 2.0 token introspection authentication handler for ASP.NET Core.

- github https://github.com/IdentityModel/IdentityModel.AspNetCore.OAuthIntrospection
- nuget https://www.nuget.org/packages/IdentityModel.AspNetCore.OAuthIntrospection/
- CI builds https://github.com/orgs/IdentityModel/packages

CHAPTER 4

IdentityModel.OidcClient

.NET based implementation of the **OAuth 2.0 for native apps** BCP. Certified by the OpenID Foundation.

- github https://github.com/IdentityModel/IdentityModel.OidcClient
- nuget https://www.nuget.org/packages/IdentityModel.OidClient
- CI builds https://github.com/orgs/IdentityModel/packages

CHAPTER 5

oidc-client.js

JavaScript based implementation of the **OAuth 2.0 for browser-based applications** BCP. Certified by the OpenID Foundation

- github https://github.com/IdentityModel/oidc-client-js
- npm https://www.npmjs.com/package/oidc-client

5.1 Overview

IdentityModel contains client libraries for many interactions with endpoints defined in OpenID Connect and OAuth 2.0. All of these libraries have a common design, let's examine the various layers using the client for the token endpoint.

5.1.1 Request and response objects

All protocol request are modelled as request objects and have a common base class called `ProtocolRequest` which has properties to set the endpoint address, client ID, client secret, client assertion, and the details of how client secrets are transmitted (e.g. authorization header vs POST body). `ProtocolRequest` derives from `HttpRequestMessage` and thus also allows setting custom headers etc.

The following code snippet creates a request for a client credentials grant type:

```
var request = new ClientCredentialsTokenRequest
{
    Address = "https://demo.identityserver.io/connect/token",
    ClientId = "client",
    ClientSecret = "secret"
});
```

While in theory you could now call `Prepare` (which internally sets the headers, body and adress) and send the request via a plain `HttpClient`, typically there are more parameters with special semantics and encoding required. That's why we provide extension methods to do the low level work.

Equally, a protocol response has a corresponding `ProtocolResponse` implementation that parses the status codes and response content. The following code snippet would parse the raw HTTP response from a token endpoint and turn it into a `TokenResponse` object:

```
var tokenResponse = await ProtocolResponse.FromHttpResponseAsync<TokenResponse>
↪(httpResponse);
```

Again these steps are automated using the extension methods. So let's have a look at an example next.

5.1.2 Extension methods

For each protocol interaction, an extension method for `HttpMessageInvoker` (that's the base class of `HttpClient`) exists. The extension methods expect a request object and return a response object.

It is your responsibility to setup and manage the lifetime of the `HttpClient`, e.g. manually:

```
var client = new HttpClient();

var response = await client.RequestClientCredentialsTokenAsync(new␣
↪ClientCredentialsTokenRequest
{
    Address = "https://demo.identityserver.io/connect/token",
    ClientId = "client",
    ClientSecret = "secret"
});
```

You might want to use other techniques to obtain an `HttpClient`, e.g. via the HTTP client factory:

```
var client = HttpClientFactory.CreateClient("my_named_token_client");

var response = await client.RequestClientCredentialsTokenAsync(new␣
↪ClientCredentialsTokenRequest
{
    Address = "https://demo.identityserver.io/connect/token",
    ClientId = "client",
    ClientSecret = "secret"
});
```

All other endpoint client follow the same design.

Note: Some client libraries also include a stateful client object (e.g. `TokenClient` and `IntrospectionClient`). See the corresponding section to find out more.

5.2 Discovery Endpoint

The client library for the OpenID Connect discovery endpoint is provided as an extension method for `HttpClient`. The `GetDiscoveryDocumentAsync` method returns a `DiscoveryResponse` object that has both strong and weak typed accessors for the various elements of the discovery document.

You should always check the `IsError` and `Error` properties before accessing the contents of the document.

Example:

```
var client = new HttpClient();

var disco = await client.GetDiscoveryDocumentAsync("https://demo.identityserver.io");
if (disco.IsError) throw new Exception(disco.Error);
```

Standard elements can be accessed by using properties:

```
var tokenEndpoint = disco.TokenEndpoint;
var keys = disco.KeySet.Keys;
```

Custom elements (or elements not covered by the standard properties) can be accessed like this:

```
// returns string or null
var stringValue = disco.TryGetString("some_string_element");

// return a nullable boolean
var boolValue = disco.TryGetBoolean("some_boolean_element");

// return array (maybe empty)
var arrayValue = disco.TryGetStringArray("some_array_element");

// returns JToken
var rawJson = disco.TryGetValue("some_element);
```

5.2.1 Discovery Policy

By default the discovery response is validated before it is returned to the client, validation includes:

- enforce that HTTPS is used (except for localhost addresses)
- enforce that the issuer matches the authority
- enforce that the protocol endpoints are on the same DNS name as the authority
- enforce the existence of a keyset

Policy violation errors will set the `ErrorType` property on the `DiscoveryResponse` to `PolicyViolation`.

All of the standard validation rules can be modified using the `DiscoveryPolicy` class, e.g. disabling the issuer name check:

```
var disco = await client.GetDiscoveryDocumentAsync(new DiscoveryDocumentRequest
{
    Address = "https://demo.identityserver.io",
    Policy =
    {
        ValidateIssuerName = false
    }
});
```

You can also customize validation strategy based on the authority with your own implementation of `IAuthorityValidationStrategy`. By default, comparison uses ordinal string comparison. To switch to Uri comparison:

```
var disco = await client.GetDiscoveryDocumentAsync(new DiscoveryDocumentRequest
{
    Address = "https://demo.identityserver.io",
```

(continues on next page)

(continued from previous page)

```
    Policy =
    {
        AuthorityValidationStrategy = new AuthorityUrlValidationStrategy()
    }
});
```

5.2.2 Caching the Discovery Document

You should periodically update your local copy of the discovery document, to be able to react to configuration changes on the server. This is especially important for playing nice with automatic key rotation.

The `DiscoveryCache` class can help you with that.

The following code will set-up the cache, retrieve the document the first time it is needed, and then cache it for 24 hours:

```
var cache = new DiscoveryCache("https://demo.identityserver.io");
```

You can then access the document like this:

```
var disco = await cache.GetAsync();
if (disco.IsError) throw new Exception(disco.Error);
```

You can specify the cache duration using the `CacheDuration` property and also specify a custom discovery policy by passing in a `DiscoveryPolicy` to the constructor.

Caching and HttpClient Instances

By default the discovery cache will create a new instance of `HttpClient` every time it needs to access the discovery endpoint. You can modify this behavior in two ways, either by passing in a pre-created instance into the constructor, or by providing a function that will return an `HttpClient` when needed.

The following code will setup the discovery cache in DI and will use the `HttpClientFactory` to create clients:

```
services.AddSingleton<IDiscoveryCache>(r =>
{
    var factory = r.GetRequiredService<IHttpClientFactory>();
    return new DiscoveryCache(Constants.Authority, () => factory.CreateClient());
});
```

5.3 Token Endpoint

The client library for the token endpoint (OAuth 2.0 and OpenID Connect) is provided as a set of extension methods for `HttpClient`. This allows creating and managing the lifetime of the `HttpClient` the way you prefer - e.g. statically or via a factory like the Microsoft `HttpClientFactory`.

5.3.1 Requesting a token

The main extension method is called `RequestTokenAsync` - it has direct support for standard parameters like client ID/secret (or assertion) and grant type, but it also allows setting arbitrary other parameters via a dictionary. All other extensions methods ultimately call this method internally:

```
var client = new HttpClient();

var response = await client.RequestTokenAsync(new TokenRequest
{
    Address = "https://demo.identityserver.io/connect/token",
    GrantType = "custom",

    ClientId = "client",
    ClientSecret = "secret",

    Parameters =
    {
        { "custom_parameter", "custom value"},
        { "scope", "api1" }
    }
});
```

The response is of type `TokenResponse` and has properties for the standard token response parameters like `access_token`, `expires_in` etc. You also have access to the the raw response as well as to a parsed JSON document (via the `Raw` and `Json` properties).

Before using the response, you should always check the `IsError` property to make sure the request was successful:

```
if (response.IsError) throw new Exception(response.Error);

var token = response.AccessToken;
var custom = response.Json.TryGetString("custom_parameter");
```

5.3.2 Requesting a token using the `client_credentials` Grant Type

The `RequestClientCredentialsToken` extension method has convenience properties for the `client_credentials` grant type:

```
var response = await client.RequestClientCredentialsTokenAsync(new␣
↪ClientCredentialsTokenRequest
{
    Address = "https://demo.identityserver.io/connect/token",

    ClientId = "client",
    ClientSecret = "secret",
    Scope = "api1"
});
```

5.3.3 Requesting a token using the `password` Grant Type

The `RequestPasswordToken` extension method has convenience properties for the `password` grant type:

```
var response = await client.RequestPasswordTokenAsync(new PasswordTokenRequest
{
    Address = "https://demo.identityserver.io/connect/token",

    ClientId = "client",
    ClientSecret = "secret",
    Scope = "api1",
```

(continues on next page)

(continued from previous page)

```
    UserName = "bob",
    Password = "bob"
});
```

5.3.4 Requesting a token using the `authorization_code` Grant Type

The `RequestAuthorizationCodeToken` extension method has convenience properties for the `authorization_code` grant type and PKCE:

```
var response = await client.RequestAuthorizationCodeTokenAsync(new␣
↪AuthorizationCodeTokenRequest
{
    Address = IdentityServerPipeline.TokenEndpoint,

    ClientId = "client",
    ClientSecret = "secret",

    Code = code,
    RedirectUri = "https://app.com/callback",

    // optional PKCE parameter
    CodeVerifier = "xyz"
});
```

5.3.5 Requesting a token using the `refresh_token` Grant Type

The `RequestRefreshToken` extension method has convenience properties for the `refresh_token` grant type:

```
var response = await _client.RequestRefreshTokenAsync(new RefreshTokenRequest
{
    Address = TokenEndpoint,

    ClientId = "client",
    ClientSecret = "secret",

    RefreshToken = "xyz"
});
```

5.3.6 Requesting a Device Token

The `RequestDeviceToken` extension method has convenience properties for the `urn:ietf:params:oauth:grant-type:device_code` grant type:

```
var response = await client.RequestDeviceTokenAsync(new DeviceTokenRequest
{
    Address = disco.TokenEndpoint,

    ClientId = "device",
    DeviceCode = authorizeResponse.DeviceCode
});
```

5.4 Token Introspection Endpoint

The client library for OAuth 2.0 token introspection is provided as an extension method for `HttpClient`.

The following code sends a reference token to an introspection endpoint:

```
var client = new HttpClient();

var response = await client.IntrospectTokenAsync(new TokenIntrospectionRequest
{
    Address = "https://demo.identityserver.io/connect/introspect",
    ClientId = "api1",
    ClientSecret = "secret",

    Token = accessToken
});
```

The response is of type `TokenIntrospectionResponse` and has properties for the standard response parameters. You also have access to the the raw response as well as to a parsed JSON document (via the `Raw` and `Json` properties).

Before using the response, you should always check the `IsError` property to make sure the request was successful:

```
if (response.IsError) throw new Exception(response.Error);

var isActive = response.IsActive;
var claims = response.Claims;
```

5.5 Token Revocation Endpoint

The client library for OAuth 2.0 token revocation is provided as an extension method for `HttpClient`.

The following code revokes an access token token at a revocation endpoint:

```
var client = new HttpClient();

var result = await client.RevokeTokenAsync(new TokenRevocationRequest
{
    Address = "https://demo.identityserver.io/connect/revocation",
    ClientId = "client",
    ClientSecret = "secret",

    Token = accessToken
});
```

The response is of type `TokenRevocationResponse` gives you access to the the raw response as well as to a parsed JSON document (via the `Raw` and `Json` properties).

Before using the response, you should always check the `IsError` property to make sure the request was successful:

```
if (response.IsError) throw new Exception(response.Error);
```

5.6 UserInfo Endpoint

The client library for the OpenID Connect UserInfo endpoint is provided as an extension method for `HttpClient`.

The following code sends an access token to the UserInfo endpoint:

```
var client = new HttpClient();

var response = await client.GetUserInfoAsync(new UserInfoRequest
{
    Address = disco.UserInfoEndpoint,
    Token = token
});
```

The response is of type `UserInfoResponse` and has properties for the standard response parameters. You also have access to the the raw response as well as to a parsed JSON document (via the `Raw` and `Json` properties).

Before using the response, you should always check the `IsError` property to make sure the request was successful:

```
if (response.IsError) throw new Exception(response.Error);

var claims = response.Claims;
```

5.7 Dynamic Client Registration

The client library for OpenID Connect Dynamic Client Registration is provided as an extension method for `HttpClient`.

The following code sends a registration request:

```
var client = new HttpClient();

var response = await client.RegisterClientAsync(new DynamicClientRegistrationRequest
{
    Address = Endpoint,
    Document = new DynamicClientRegistrationDocument
    {
        RedirectUris = { redirectUri },
        ApplicationType = "native"
    }
});
```

Note: The `DynamicClientRegistrationDocument` class has strongly typed properties for all standard registration parameters as defines by the specification. If you want to add custom parameters, it is recommended to derive from this class and add your own properties.

The response is of type `RegistrationResponse` and has properties for the standard response parameters. You also have access to the the raw response as well as to a parsed JSON document (via the `Raw` and `Json` properties).

Before using the response, you should always check the `IsError` property to make sure the request was successful:

```
if (response.IsError) throw new Exception(response.Error);
```

(continues on next page)

(continued from previous page)

```
var clientId = response.ClientId;
var secret = resopnse.ClientSecret;
```

5.8 Device Authorization Endpoint

The client library for the OAuth 2.0 device flow device authorization is provided as an extension method for `HttpClient`.

The following code sends a device authorization request:

```
var client = new HttpClient();

var response = await client.RequestDeviceAuthorizationAsync(new␣
↪DeviceAuthorizationRequest
{
    Address = "https://demo.identityserver.io/connect/device_authorize",
    ClientId = "device"
});
```

The response is of type `DeviceAuthorizationResponse` and has properties for the standard response parameters. You also have access to the the raw response as well as to a parsed JSON document (via the `Raw` and `Json` properties).

Before using the response, you should always check the `IsError` property to make sure the request was successful:

```
if (response.IsError) throw new Exception(response.Error);

var userCode = response.UserCode;
var deviceCode = response.DeviceCode;
var verificationUrl = response.VerificationUri;
var verificationUrlComplete = response.VerificationUriComplete;
```

5.9 Protocol and Claim Type Constants

When working with OAuth 2.0, OpenID Connect and claims, there are a lot of "magic strings" for claim types and protocoal values. IdentityModel provides a couple of constant strings classes to help with that.

5.9.1 OAuth 2.0 and OpenID Connect Protocol Values

The `OidcConstants` class has all the values for grant types, parameter names, error names etc.

5.9.2 JWT Claim Types

The `JwtClaimTypes` class has all standard claim types found in the OpenID Connect, JWT and OAuth 2.0 specs - many of them are also aggregated at IANA.

5.10 Creating Request URLs (e.g. for Authorize and EndSession endpoints)

The `RequestUrl` class is a helper for creating URLs with query string parameters, e.g.:

```
var ru = new RequestUrl("https://server/endpoint");

// produces https://server/endpoint?foo=foo&bar=bar
var url = ru.Create(new
    {
        foo: "foo",
        bar: "bar"
    });
```

As a parameter to the `Create` method you can either pass in an object, or a string dictionary. In both cases the properties/values will be serialized to key/value pairs.

Note: All values will be URL encoded.

5.10.1 Authorization Endpoint

For most cases, the OAuth 2.0 and OpenID Connect authorization endpoint expects a GET request with a number of query string parameters.

The `CreateAuthorizeUrl` extension method creats URLs for the authorize endpoint - it has support the most common parameters:

```
/// <summary>
/// Creates an authorize URL.
/// </summary>
/// <param name="request">The request.</param>
/// <param name="clientId">The client identifier.</param>
/// <param name="responseType">The response type.</param>
/// <param name="scope">The scope.</param>
/// <param name="redirectUri">The redirect URI.</param>
/// <param name="state">The state.</param>
/// <param name="nonce">The nonce.</param>
/// <param name="loginHint">The login hint.</param>
/// <param name="acrValues">The acr values.</param>
/// <param name="prompt">The prompt.</param>
/// <param name="responseMode">The response mode.</param>
/// <param name="codeChallenge">The code challenge.</param>
/// <param name="codeChallengeMethod">The code challenge method.</param>
/// <param name="display">The display option.</param>
/// <param name="maxAge">The max age.</param>
/// <param name="uiLocales">The ui locales.</param>
/// <param name="idTokenHint">The id_token hint.</param>
/// <param name="extra">Extra parameters.</param>
/// <returns></returns>
public static string CreateAuthorizeUrl(this RequestUrl request,
    string clientId,
    string responseType,
    string scope = null,
```

(continues on next page)

(continued from previous page)

```
    string redirectUri = null,
    string state = null,
    string nonce = null,
    string loginHint = null,
    string acrValues = null,
    string prompt = null,
    string responseMode = null,
    string codeChallenge = null,
    string codeChallengeMethod = null,
    string display = null,
    int? maxAge = null,
    string uiLocales = null,
    string idTokenHint = null,
    object extra = null)
{ ... }
```

Example:

```
var ru = new RequestUrl("https://demo.identityserver.io/connect/authorize");

var url = ru.CreateAuthorizeUrl(
    clientId: "client",
    responseType: "implicit",
    redirectUri: "https://app.com/callback",
    nonce: "xyz",
    scope: "openid");
```

Note: The `extra` parameter can either be a string dictionary or an arbitrary other type with properties. In both cases the values will be serialized as keys/values.

5.10.2 EndSession Endpoint

The `CreateEndSessionUrl` extensions methods supports the most common parameters:

```
/// <summary>
/// Creates a end_session URL.
/// </summary>
/// <param name="request">The request.</param>
/// <param name="idTokenHint">The id_token hint.</param>
/// <param name="postLogoutRedirectUri">The post logout redirect URI.</param>
/// <param name="state">The state.</param>
/// <param name="extra">The extra parameters.</param>
/// <returns></returns>
public static string CreateEndSessionUrl(this RequestUrl request,
    string idTokenHint = null,
    string postLogoutRedirectUri = null,
    string state = null,
    object extra = null)
{ ... }
```

Note: The `extra` parameter can either be a string dictionary or an arbitrary other type with properties. In both cases the values will be serialized as keys/values.

5.11 Fluent API for the X.509 Certificate Store

A common place to store X.509 certificates is the Windows X.509 certificate store. The raw APIs for the store are a bit arcane (and also slightly changed between .NET Framework and .NET Core).

The `X509` class is a simplified API to load certificates from the store. The following code loads a certificate by name from the personal machine store:

```
var cert = X509
              .LocalMachine
              .My
              .SubjectDistinguishedName
              .Find("CN=sts")
              .FirstOrDefault();
```

You can load certs from the machine or user store and from `My`, `AddressBook`, `TrustedPeople`, `CertificateAuthority` and `TrustedPublisher` respectively. You can search for subject name, thumbprint, issuer name or serial number.

5.12 Base64 URL Encoding

JWT tokens are serialized using Base64 URL encoding.

IdentityModel includes the `Base64Url` class to help with encoding/decoding:

```
var text = "hello";
var b64url = Base64Url.Encode(text);

text = Base64Url.Decode(b64url);
```

Note: ASP.NET Core has built-in support via WebEncoders.Base64UrlEncode and WebEncoders.Base64UrlDecode.

5.13 Epoch Time Conversion

JWT tokens use so called Epoch or Unix time to represent date/times.

IdentityModel contains extensions methods for `DateTime` to convert to/from Unix time:

```
var dt = DateTime.UtcNow;
var unix = dt.ToEpochTime();
```

Note: Starting with .NET Framework 4.6 and .NET Core 1.0 this functionality is built-in via DateTimeOffset.FromUnixTimeSeconds and DateTimeOffset.ToUnixTimeSeconds.

5.14 Time-Constant String Comparison

When comparing strings in a security context (e.g. comparing keys), you should try to avoid leaking timing information.

The `TimeConstantComparer` class can help with that:

```
var isEqual = TimeConstantComparer.IsEqual(key1, key2);
```

Note: Starting with .NET Core 2.1 this functionality is built in via CryptographicOperations.FixedTimeEquals

5.15 Overview

IdentityModel.AspNetCore is a helper library for ASP.NET Core web applications and service worker applications.

It helps with access token lifetime management for pure machine to machine communication and user-centric applications with refresh tokens.

5.16 Worker Applications

Workers use the client credentials grant type to request tokens from an OAuth 2.0 compatible token service.

You register the token service, client ID and secret in `ConfigureServices`, e.g.:

```
var host = Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) =>
    {
        services.AddAccessTokenManagement(options =>
        {
            options.Client.Clients.Add("identityserver", new␣
↪ClientCredentialsTokenRequest
            {
                Address = "https://demo.identityserver.io/connect/token",
                ClientId = "m2m.short",
                ClientSecret = "secret",
                Scope = "api" // optional
            });
        });
    });
```

You can register multiple clients for one or more token services if you like. Just make sure you give every client a unique name.

You can also customize the HTTP client that is used for requesting the tokens by calling the `ConfigureBackchannelHttpClient` extension method, e.g.:

```
services.AddAccessTokenManagement()
    .ConfigureBackchannelHttpClient()
        .AddTransientHttpErrorPolicy(policy => policy.WaitAndRetryAsync(new[]
        {
            TimeSpan.FromSeconds(1),
            TimeSpan.FromSeconds(2),
            TimeSpan.FromSeconds(3)
        }));
```

The above code wires up the `AccessTokenManagementService` and the `ClientAccessTokenCache` in the DI system. The service is the main entry point, and features a method called `GetClientAccessTokenAsync` (which you can also access via the HTTP context using `HttpContext.GetClientAccessTokenAsync`). This

method checks if a token for the client is cached, and if not requests one and caches it. The cache implementation can be replaced.

One piece of plumbing that automatically uses the token management service is the `ClientAccessTokenHandler`, which is a delegating handler to plug-in to `HttpClient`.

The easiest way to register an HTTP client that uses the token management is by calling `AddClientAccessTokenClient`:

```
services.AddClientAccessTokenClient("client", configureClient: client =>
{
    client.BaseAddress = new Uri("https://demo.identityserver.io/api/");
});
```

You can pass in the name of your HTTP client, the name of the token service configuration (you can omit this if you only have one token client) and additional customization. This returns the typical builder for the HTTP client factory to add aditional handlers.

It is also possible to add the handler to any HTTP client registration using the `AddClientAccessTokenHandler` extension method (which optionally also takes a token client name), e.g. a typed client:

```
services.AddHttpClient<MyClient>(client =>
{
    client.BaseAddress = new Uri("https://demo.identityserver.io/api/");
})
    .AddClientAccessTokenHandler();
```

5.16.1 Usage

You can use use one of the various ways to obtain an HTTP client with the handler set up, e.g. using the HTTP client factory:

```
public Worker(IHttpClientFactory factory)
{
    _client = factory.CreateClient("client");
}
```

..and then use that client to make API calls - all token management will be done under the covers:

```
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
        Console.WriteLine("\n\n");
        _logger.LogInformation("Worker running at: {time}", DateTimeOffset.Now);

        var response = await _client.GetStringAsync("test");
        _logger.LogInformation("API response: {response}", response);

        await Task.Delay(5000, stoppingToken);
    }
}
```

Full sample can be found in the samples.

5.17 Web Applications

In web applications you might either want to call APIs using the client identity or the user identity. The client identity scenario is exactly the same as the previous section that covered service workers.

For user centric scenarios, this library operates under a couple of assumptions by default:

- you are using the OpenID Connect to authenticate the user
- the OpenID Connect provider is also your token service for access tokens
- you are requesting access and refresh tokens and are using a flow that allows to refresh tokens (e.g. code flow)
- you use the `SaveTokens` option to store the access and refresh token in the authentication session

If all these pre-conditions are met, the token management plumbing will infer server endpoints, client ID and secret and other configuration settings from the OpenID Connect handler, and all you need to add is:

```
services.AddAccessTokenManagement();
```

To interact with the underlying services, this library adds two extension methods for `HttpContext`:

- `GetUserAccessTokenAsync` - retrieves current access token for user and refreshes it if it is expired (or expiring soon - can be configured)
- `RevokeUserRefreshTokenAsync` - revokes the refresh token when it is not needed anymore

Same as with the client access token, you can also wire up an HTTP client that automatically uses the token management library:

```
services.AddUserAccessTokenClient("user_client", client =>
{
    client.BaseAddress = new Uri("https://demo.identityserver.io/api/");
});
```

This registers an Http client with the factory, that you can use in your business code to make API calls. A more complete configuration could look like this:

```
services.AddAuthentication(options =>
{
    options.DefaultScheme = "cookie";
    options.DefaultChallengeScheme = "oidc";
})
    .AddCookie("cookie", options =>
    {
        options.Cookie.Name = "mvccode";

        options.Events.OnSigningOut = async e =>
        {
            // revoke refresh token on sign-out
            await e.HttpContext.RevokeUserRefreshTokenAsync();
        };
    })
    .AddOpenIdConnect("oidc", options =>
    {
        options.Authority = "https://demo.identityserver.io";

        options.ClientId = "interactive.confidential.short";
        options.ClientSecret = "secret";
```

(continues on next page)

(continued from previous page)

```
        // code flow + PKCE (PKCE is turned on by default)
        options.ResponseType = "code";

        options.Scope.Clear();
        options.Scope.Add("openid");
        options.Scope.Add("profile");
        options.Scope.Add("email");
        options.Scope.Add("offline_access");
        options.Scope.Add("api");

        // keeps id_token smaller
        options.GetClaimsFromUserInfoEndpoint = true;
        options.SaveTokens = true;

        options.TokenValidationParameters = new TokenValidationParameters
        {
            NameClaimType = "name",
            RoleClaimType = "role"
        };
    });

// adds user and client access token management
services.AddAccessTokenManagement(options =>
{
    // client config is inferred from OpenID Connect settings
    // if you want to specify scopes explicitly, do it here, otherwise the scope␣
↪parameter will not be sent
    options.Client.Scope = "api";
})
    .ConfigureBackchannelHttpClient()
        .AddTransientHttpErrorPolicy(policy => policy.WaitAndRetryAsync(new[]
        {
            TimeSpan.FromSeconds(1),
            TimeSpan.FromSeconds(2),
            TimeSpan.FromSeconds(3)
        }));

// registers HTTP client that uses the managed user access token
services.AddUserAccessTokenClient("user_client", client =>
{
    client.BaseAddress = new Uri("https://demo.identityserver.io/api/");
});

// registers HTTP client that uses the managed client access token
services.AddClientAccessTokenClient("client", configureClient: client =>
{
    client.BaseAddress = new Uri("https://demo.identityserver.io/api/");
});
```

Full sample can be found in the samples.

5.18 Extensibility

The main extensibility points are around token storage (users) and token caching (clients).

5.18.1 Client access tokens

Client access tokens are cached in memory by default. The default cache implementation uses the `IDistributedCache` abstraction in ASP.NET Core.

You can either

- replace the standard distributed cache with something else
- replace the `IClientAccessTokenCache` implementation in DI altogether

5.18.2 User access tokens

User access tokens are stored/cached using the ASP.NET Core authentication session mechanism. For that you need to set the `SaveTokens` flag on the OpenID Connect handler to `true`.

ASP.NET Core stores the authentication session in a cookie by default. You can replace that storage mechanisms by setting the `SessionStore` property on the cookie handler.

If you want to take over the token handling altogether, replace the `IUserTokenStore` implementation in DI.

5.19 Overview

IdentityModel.OidcClient is a C#/.Net Standard 2.0 reference implementation of the "OAuth 2.0 for native Applications" BCP (RFC 8252).

It is also an officially certified OpenId Connect client library.

Supported Platforms:

- netstandard2.0
- .NET Framework >= 4.6.1
- .NET Core >= 2.0
- UWP
- Xamarin iOS & Android

5.20 Manual Mode

In manual mode, OidcClient helps you with creating the necessary start URL and state parameters, but you need to coordinate with whatever browser you want to use, e.g.:

```
var options = new OidcClientOptions
{
    Authority = "https://demo.identityserver.io",
    ClientId = "native",
    RedirectUri = redirectUri,
    Scope = "openid profile api"
};

var client = new OidcClient(options);

// generate start URL, state, nonce, code challenge
var state = await client.PrepareLoginAsync();
```

When the browser work is done, OidcClient can take over to process the response, get the access/refresh tokens, contact userinfo endpoint etc..:

```
var result = await client.ProcessResponseAsync(data, state);
```

The result will contain the tokens and the claims of the user.

5.21 Automatic Mode

In automatic mode, you can encapsulate all browser interactions by implementing the IBrowser interface:

```
var options = new OidcClientOptions
{
    Authority = "https://demo.identityserver.io",
    ClientId = "native",
    RedirectUri = redirectUri,
    Scope = "openid profile api",
    Browser = new SystemBrowser()
};

var client = new OidcClient(options);
```

Once that is done, authentication and token requests become one line of code:

```
var result = await client.LoginAsync();
```

5.22 Logging

OidcClient has support for the standard .NET logging facilities, e.g. using Serilog:

```
var serilog = new LoggerConfiguration()
    .MinimumLevel.Verbose()
    .Enrich.FromLogContext()
    .WriteTo.LiterateConsole(outputTemplate: "[{Timestamp:HH:mm:ss} {Level}]
↪{SourceContext}{NewLine}{Message}{NewLine}{Exception}{NewLine}")
    .CreateLogger();

options.LoggerFactory.AddSerilog(serilog);
```

5.23 Samples

See here for samples using WinForms, Console and Xamarin iOS and Android.

5.24 Overview

Oidc-client is a library to provide OpenID Connect (OIDC) and OAuth2 protocol support for client-side, browser-based JavaScript client applications. Also included is support for user session and access token management.

See here for the current documentation on github.