IdentityModel Documentation

Dominick Baier and Brock Allen

Aug 05, 2019

Protocol Client Libraries

IdentityModel is a .NET standard helper library for claims-based identity, OAuth 2.0 and OpenID Connect.

It has the following high level features:

- client libraries for standard OAuth 2.0 and OpenID Connect endpoints like authorize, token, discovery, introspection, revocation etc.
- helpers for token management
- constants for standard JWT claim types and protocol values
- simplified API to access the X509 certificate store
- misc helpers for base64 URL encoding, time constant string comparison and epoch time

CHAPTER 1

How to get

- github https://github.com/IdentityModel/IdentityModel2
- nuget https://www.nuget.org/packages/IdentityModel/
- CI builds https://www.myget.org/F/identity/

1.1 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");
```

(continues on next page)

(continued from previous page)

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

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

1.1.1 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", Policy = {
        AuthorityValidationStrategy = AuthorityUrlValidationStrategy.Instance
    }
});
```

1.1.2 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());
});
```

1.2 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.

While you can use any means of creating URLs with parameters to create the right strings, the *RequestUrl* class is a simple helper to accomplish this task.

In particular, you can use the `CreateAuthorizeUrl` extension method to create 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>
```

(continues on next page)

(continued from previous page)

```
/// <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,
    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.

1.3 EndSession Endpoint

The *RequestUrl* class can be used to construct URLs to the OpenID Connect 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>
```

(continues on next page)

(continued from previous page)

```
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.

1.4 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`.

1.4.1 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");
```

1.4.2 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"
});
```

1.4.3 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",

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

1.4.4 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"
});
```

1.4.5 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"
});
```

1.4.6 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
});
```

1.5 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 `IntrospectionResponse` 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;
```

1.6 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);
```

1.7 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;
```

1.8 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);

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

1.9 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;
```

1.10 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.

1.10.1 OAuth 2.0 and OpenID Connect Protocol Values

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

1.10.2 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.

1.11 Creating Request URLs

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.

`RequestUrl` is mostly useful when you create extension methods for modelling domain specific URL structures. For examples see the *Authorize Endpoint* and *EndSession Endpoint*.

1.12 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.

1.13 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.

1.14 Epoch Time Conversion

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

IdentityModel contains extensions methods for `DateTime` and `DateTimeOffset` or 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.

1.15 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