IdentityModel Documentation

Dominick Baier and Brock Allen

Oct 13, 2019
# IdentityModel

<table>
<thead>
<tr>
<th>1</th>
<th>IdentityModel</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>IdentityModel.AspNetCore</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>IdentityModel.AspNetCore.OAuthIntrospection</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>IdentityModel.OidcClient</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>oicd-client.js</td>
<td>11</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview</td>
<td>11</td>
</tr>
<tr>
<td>5.2</td>
<td>Discovery Endpoint</td>
<td>12</td>
</tr>
<tr>
<td>5.3</td>
<td>Token Endpoint</td>
<td>14</td>
</tr>
<tr>
<td>5.4</td>
<td>Token Introspection Endpoint</td>
<td>17</td>
</tr>
<tr>
<td>5.5</td>
<td>Token Revocation Endpoint</td>
<td>17</td>
</tr>
<tr>
<td>5.6</td>
<td>UserInfo Endpoint</td>
<td>17</td>
</tr>
<tr>
<td>5.7</td>
<td>Dynamic Client Registration</td>
<td>18</td>
</tr>
<tr>
<td>5.8</td>
<td>Device Authorization Endpoint</td>
<td>19</td>
</tr>
<tr>
<td>5.9</td>
<td>Protocol and Claim Type Constants</td>
<td>19</td>
</tr>
<tr>
<td>5.10</td>
<td>Creating Request URLs (e.g. for Authorize and EndSession endpoints)</td>
<td>19</td>
</tr>
<tr>
<td>5.11</td>
<td>Fluent API for the X.509 Certificate Store</td>
<td>22</td>
</tr>
<tr>
<td>5.12</td>
<td>Base64 URL Encoding</td>
<td>22</td>
</tr>
<tr>
<td>5.13</td>
<td>Epoch Time Conversion</td>
<td>22</td>
</tr>
<tr>
<td>5.14</td>
<td>Time-Constant String Comparison</td>
<td>22</td>
</tr>
<tr>
<td>5.15</td>
<td>Overview</td>
<td>23</td>
</tr>
<tr>
<td>5.16</td>
<td>Manual Mode</td>
<td>23</td>
</tr>
<tr>
<td>5.17</td>
<td>Automatic Mode</td>
<td>23</td>
</tr>
<tr>
<td>5.18</td>
<td>Logging</td>
<td>24</td>
</tr>
<tr>
<td>5.19</td>
<td>Samples</td>
<td>24</td>
</tr>
<tr>
<td>5.20</td>
<td>Overview</td>
<td>24</td>
</tr>
</tbody>
</table>
IdentityModel is a family of libraries for building OAuth 2.0 and OpenID Connect clients.
CHAPTER 1

IdentityModel

Base library for OIDC and OAuth 2.0 related protocol operations and constants and other misc helpers (.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
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
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
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.OidcClient
- **CI builds** https://github.com/orgs/IdentityModel/packages
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 requests 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:

```javascript
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 address) 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:

```csharp
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:

```csharp
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:

```csharp
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)
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:

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

You can then access the document like this:

```csharp
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:

```csharp
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:
```csharp
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:

```
class TokenResponse
{
    public bool IsError { get; set; }
    public string Error { get; set; }
}
```

```csharp
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:

```csharp
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:

```csharp
var response = await client.RequestPasswordTokenAsync(new
    PasswordTokenRequest
    {
        Address = "https://demo.identityserver.io/connect/token",
        ClientId = "client",
        ClientSecret = "secret",
        Scope = "api1"
    });
```

(continues on next 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:

```javascript
    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:

```javascript
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:

```javascript
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:

```csharp
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:

```csharp
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:

```csharp
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:

```csharp
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:

```javascript
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:

```javascript
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:

```javascript
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:

```javascript
if (response.IsError) throw new Exception(response.Error);
var clientId = response.ClientId;
var secret = response.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:

```csharp
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 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:

```csharp
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 protocol 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 creates URLs for the authorize endpoint - it has support the most common parameters:

```csharp
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 display = null,
                                          string codeChallenge = null,
                                          string codeChallengeMethod = null,
                                          string uiLocales = null,
                                          string idTokenHint = null,
                                          string extra = null)
```

(continues on next page)
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:

```csharp
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` extension method supports the most common parameters:

```csharp
public static string CreateEndSessionUrl(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:

```csharp
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:

```csharp
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:

```csharp
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:

```csharp
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.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.

## 5.16 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.:

```csharp
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.:

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

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

## 5.17 Automatic Mode

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

```csharp
var options = new OidcClientOptions
{
    Authority = "https://demo.identityserver.io",
    ClientId = "native",
    RedirectUri = redirectUri,
    Scope = "openid profile api",
};
```
Once that is done, authentication and token requests become one line of code:

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

### 5.18 Logging

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

```javascript
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);
```

### 5.19 Samples

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

### 5.20 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.