
IdentityServer4 Documentation

Release 1.0.0

Brock Allen, Dominick Baier

Jun 16, 2020

1	The Big Picture	3
1.1	Authentication	4
1.2	API Access	5
1.3	OpenID Connect and OAuth 2.0 – better together	5
1.4	How IdentityServer4 can help	5
2	Terminology	7
2.1	IdentityServer	7
2.2	User	8
2.3	Client	8
2.4	Resources	8
2.5	Identity Token	8
2.6	Access Token	8
3	Supported Specifications	9
3.1	OpenID Connect	9
3.2	OAuth 2.0	9
4	Packaging and Builds	11
4.1	IdentityServer4 main repo	11
4.2	Quickstart UI	11
4.3	Access token validation handler	11
4.4	Templates	12
4.5	Dev builds	12
5	Support and Consulting Options	13
5.1	Free support	13
5.2	Commercial support	13
6	Demo Server	15
7	Contributing	17
7.1	How to contribute?	17
7.2	General feedback and discussions?	17
7.3	Bugs and feature requests?	17
7.4	Contributing code and content	17
7.5	Contribution projects	18

8	Overview	19
8.1	Preparation	19
9	Protecting an API using Client Credentials	21
9.1	Source Code	21
9.2	Preparation	21
9.3	Setting up the ASP.NET Core application	21
9.4	Defining an API Resource	22
9.5	Defining the client	23
9.6	Configuring IdentityServer	23
9.7	Adding an API	24
9.7.1	The controller	25
9.7.2	Adding a Nuget Dependency	25
9.7.3	Configuration	25
9.8	Creating the client	26
9.9	Calling the API	27
9.10	Further experiments	29
10	Interactive Applications with ASP.NET Core	31
10.1	Adding the UI	31
10.2	Creating an MVC client	32
10.3	Adding support for OpenID Connect Identity Scopes	34
10.4	Adding Test Users	34
10.5	Adding the MVC Client to the IdentityServer Configuration	34
10.6	Testing the client	35
10.7	Adding sign-out	37
10.8	Further Experiments	38
10.9	Adding Support for External Authentication	38
10.10	Adding Google support	38
10.11	Further experiments	40
11	ASP.NET Core and API access	41
11.1	Modifying the client configuration	41
11.2	Modifying the MVC client	42
11.3	Using the access token	42
11.4	Managing the access token	43
12	Adding a JavaScript client	45
12.1	New Project for the JavaScript client	45
12.2	Modify hosting	45
12.3	Add the static file middleware	46
12.4	Reference oidc-client	46
12.5	Add your HTML and JavaScript files	46
12.6	Add a client registration to IdentityServer for the JavaScript client	49
12.7	Allowing Ajax calls to the Web API with CORS	49
12.8	Run the JavaScript application	50
13	Using EntityFramework Core for configuration and operational data	55
13.1	IdentityServer4.EntityFramework	55
13.2	Using SqlServer	56
13.3	Database Schema Changes and Using EF Migrations	56
13.4	Configuring the Stores	56
13.5	Adding Migrations	57
13.6	Initializing the Database	57
13.7	Run the client applications	59

14	Using ASP.NET Core Identity	61
14.1	New Project for ASP.NET Identity	61
14.2	Inspect the new project	62
14.2.1	IdentityServerAspNetIdentity.csproj	62
14.2.2	Startup.cs	62
14.2.3	Config.cs	62
14.2.4	Program.cs and SeedData.cs	64
14.2.5	AccountController	64
14.3	Logging in with the MVC client	64
14.4	What's Missing?	68
15	Startup	69
15.1	Configuring services	69
15.2	Key material	69
15.3	In-Memory configuration stores	70
15.4	Test stores	70
15.5	Additional services	70
15.6	Caching	71
15.7	Configuring the pipeline	71
16	Defining Resources	73
16.1	Identity Resources	73
16.2	APIs	74
16.2.1	Scopes	75
16.2.2	Authorization based on Scopes	75
16.2.3	Parameterized Scopes	76
16.2.4	API Resources	77
17	Defining Clients	81
17.1	Defining a client for server to server communication	81
17.2	Defining an interactive application for use authentication and delegated API access	82
17.3	Defining clients in appsettings.json	82
18	Sign-in	85
18.1	Cookie authentication	85
18.2	Overriding cookie handler configuration	85
18.3	Login User Interface and Identity Management System	86
18.4	Login Workflow	86
18.5	Login Context	87
18.6	Issuing a cookie and Claims	87
19	Sign-in with External Identity Providers	89
19.1	Adding authentication handlers for external providers	89
19.2	The role of cookies	89
19.3	Triggering the authentication handler	90
19.4	Handling the callback and signing in the user	91
19.5	State, URL length, and ISecureDataFormat	92
20	Windows Authentication	95
20.1	On Windows using IIS hosting	95
21	Sign-out	99
21.1	Removing the authentication cookie	99
21.2	Notifying clients that the user has signed-out	99
21.3	Sign-out initiated by a client application	100

22 Sign-out of External Identity Providers	101
23 Federated Sign-out	103
24 Federation Gateway	105
24.1 Implementation	106
25 Consent	107
25.1 Consent Page	107
25.2 Authorization Context	107
25.3 Informing IdentityServer of the consent result	108
25.4 Returning the user to the authorization endpoint	108
26 Protecting APIs	109
26.1 The IdentityServer authentication handler	110
26.2 Supporting reference tokens	110
26.3 Validating scopes	111
27 Deployment	113
27.1 Typical architecture	113
27.2 Configuration data	114
27.3 Key material	114
27.4 Operational data	114
27.5 ASP.NET Core data protection	114
27.6 ASP.NET Core distributed caching	115
28 Logging	117
28.1 Setup for Serilog	117
29 Events	119
29.1 Emitting events	119
29.2 Custom sinks	120
29.3 Built-in events	120
29.4 Custom events	121
30 Cryptography, Keys and HTTPS	123
30.1 Token signing and validation	123
30.2 Signing key rollover	123
30.3 Data protection	124
30.4 HTTPS	124
31 Grant Types	125
31.1 Machine to Machine Communication	125
31.2 Interactive Clients	126
31.3 Interactive clients without browsers or with constrained input devices	127
31.4 Custom scenarios	127
32 Client Authentication	129
32.1 Creating a shared secret	129
32.2 Authentication using a shared secret	130
32.3 Authentication using an asymmetric Key	130
33 Extension Grants	133
33.1 Example: Simple delegation using an extension grant	134
34 Resource Owner Password Validation	137

35 Refresh Tokens	139
35.1 Additional client settings	139
36 Reference Tokens	141
37 Proof-of-Possession Access Tokens	143
38 Mutual TLS	145
38.1 Server setup	145
38.2 ASP.NET Core setup	145
38.3 IdentityServer setup	146
38.4 Client authentication	147
38.4.1 Using a client certificate to authenticate to IdentityServer	148
38.5 Sender-constrained access tokens	149
38.5.1 Confirmation claim	149
38.5.2 Validating and accepting a client certificate in APIs	150
38.5.3 Introspection and the confirmation claim	151
38.6 Ephemeral client certificates	152
38.6.1 Using an ephemeral certificate to request a token	152
39 Authorize Request Objects	155
39.1 Passing request JWTs by reference	156
39.2 Accessing the request object data	156
40 Custom Token Request Validation and Issuance	157
41 CORS	159
41.1 Client-based CORS Configuration	159
41.2 Custom Cors Policy Service	159
41.3 Mixing IdentityServer's CORS policy with ASP.NET Core's CORS policies	160
42 Discovery	161
42.1 Extending discovery	161
43 Adding more API Endpoints	163
43.1 Discovery	164
43.2 Advanced	164
43.3 Claims Transformation	165
44 Adding new Protocols	167
44.1 Typical authentication workflow	167
44.2 Useful IdentityServer services	167
45 Tools	169
46 Discovery Endpoint	171
47 Authorize Endpoint	173
48 Token Endpoint	175
48.1 Example	176
49 UserInfo Endpoint	177
49.1 Example	177

50 Device Authorization Endpoint	179
50.1 Example	179
51 Introspection Endpoint	181
51.1 Example	181
52 Revocation Endpoint	183
52.1 Example	183
53 End Session Endpoint	185
53.1 Parameters	185
53.2 Example	186
54 IdentityServer Options	187
54.1 Endpoints	187
54.2 Discovery	187
54.3 Authentication	188
54.4 Events	188
54.5 InputLengthRestrictions	188
54.6 UserInteraction	188
54.7 Caching	189
54.8 CORS	189
54.9 CSP (Content Security Policy)	189
54.10 Device Flow	189
54.11 Mutual TLS	190
55 Identity Resource	191
56 API Scope	193
56.1 Defining API scope in appsettings.json	193
57 API Resource	195
57.1 Defining API resources in appsettings.json	195
58 Client	197
58.1 Basics	197
58.2 Authentication/Logout	198
58.3 Token	198
58.4 Consent Screen	199
58.5 Device flow	199
59 GrantValidationResult	201
60 Profile Service	203
60.1 IProfileService APIs	203
60.2 ProfileDataRequestContext	203
60.3 Requested scopes and claims mapping	204
60.4 IsActiveContext	204
61 IdentityServer Interaction Service	205
61.1 IIdentityServerInteractionService APIs	205
61.2 AuthorizationRequest	206
61.3 ErrorMessage	206
61.4 LogoutRequest	206
61.5 ConsentResponse	206
61.6 Consent	207

62 Device Flow Interaction Service	209
62.1 IDeviceFlowInteractionService APIs	209
62.2 DeviceFlowAuthorizationRequest	209
62.3 DeviceFlowInteractionResult	209
63 Entity Framework Support	211
63.1 Configuration Store support for Clients, Resources, and CORS settings	211
63.2 ConfigurationStoreOptions	212
63.3 Operational Store support for authorization grants, consents, and tokens (refresh and reference)	212
63.4 OperationalStoreOptions	213
63.5 Database creation and schema changes across different versions of IdentityServer	213
64 ASP.NET Identity Support	215
65 Training	217
65.1 Identity & Access Control for modern Applications (using ASP.NET Core 2 and IdentityServer4)	217
65.2 PluralSight courses	217
66 Blog posts	219
66.1 Team posts	219
66.1.1 2019	219
66.1.2 2018	219
66.1.3 2017	219
66.2 What's new posts	220
66.3 Community posts	220
67 Videos	221
67.1 2020	221
67.2 2019	221
67.3 2018	221
67.4 2017	221
67.5 2016	222
67.6 2015	222
67.7 2014	222



IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core.

Note: This docs cover the latest version on master. This might not be released yet. Use the version picker in the lower left corner to select docs for a specific version.

It enables the following features in your applications:

Authentication as a Service

Centralized login logic and workflow for all of your applications (web, native, mobile, services). IdentityServer is an officially [certified](#) implementation of OpenID Connect.

Single Sign-on / Sign-out

Single sign-on (and out) over multiple application types.

Access Control for APIs

Issue access tokens for APIs for various types of clients, e.g. server to server, web applications, SPAs and native/mobile apps.

Federation Gateway

Support for external identity providers like Azure Active Directory, Google, Facebook etc. This shields your applications from the details of how to connect to these external providers.

Focus on Customization

The most important part - many aspects of IdentityServer can be customized to fit **your** needs. Since IdentityServer is a framework and not a boxed product or a SaaS, you can write code to adapt the system the way it makes sense for your scenarios.

Mature Open Source

IdentityServer uses the permissive [Apache 2](#) license that allows building commercial products on top of it. It is also part of the [.NET Foundation](#) which provides governance and legal backing.

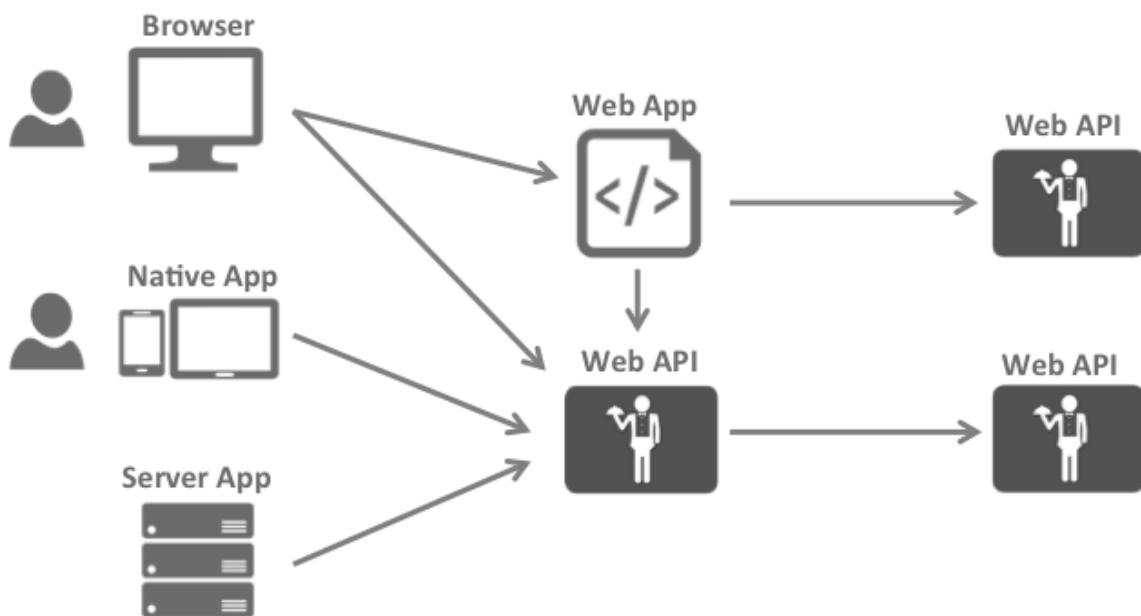
Free and Commercial Support

If you need help building or running your identity platform, *let us know*. There are several ways we can help you out.

CHAPTER 1

The Big Picture

Most modern applications look more or less like this:



The most common interactions are:

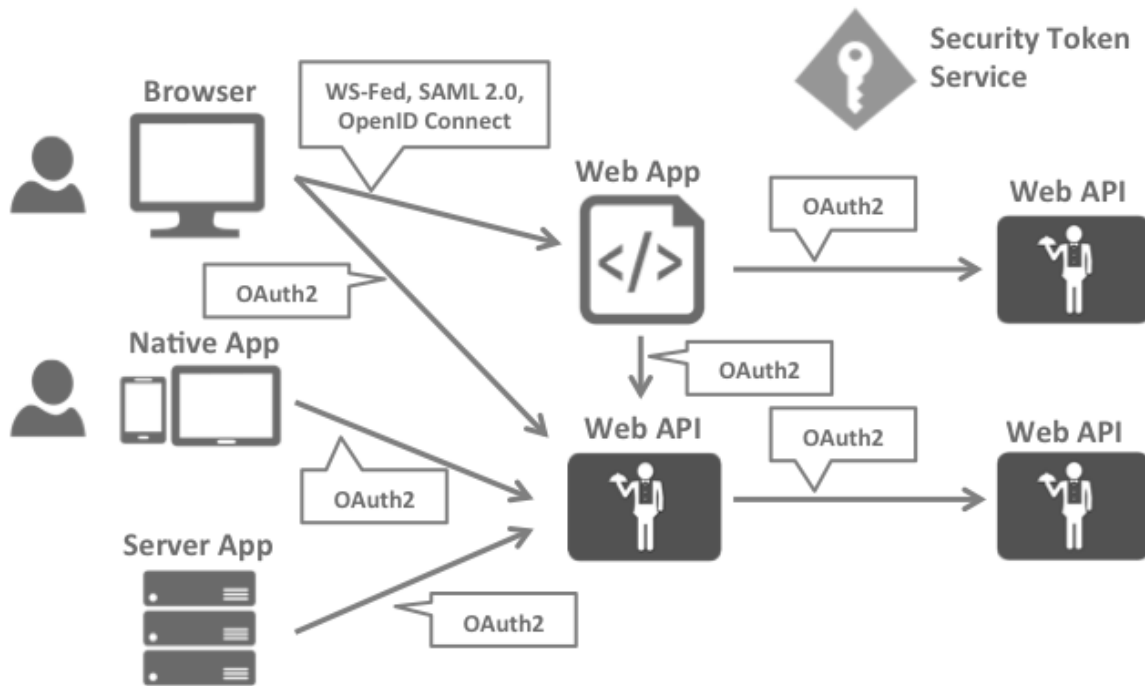
- Browsers communicate with web applications
- Web applications communicate with web APIs (sometimes on their own, sometimes on behalf of a user)
- Browser-based applications communicate with web APIs
- Native applications communicate with web APIs
- Server-based applications communicate with web APIs

- Web APIs communicate with web APIs (sometimes on their own, sometimes on behalf of a user)

Typically each and every layer (front-end, middle-tier and back-end) has to protect resources and implement authentication and/or authorization – often against the same user store.

Outsourcing these fundamental security functions to a security token service prevents duplicating that functionality across those applications and endpoints.

Restructuring the application to support a security token service leads to the following architecture and protocols:



Such a design divides security concerns into two parts:

1.1 Authentication

Authentication is needed when an application needs to know the identity of the current user. Typically these applications manage data on behalf of that user and need to make sure that this user can only access the data for which he is allowed. The most common example for that is (classic) web applications – but native and JS-based applications also have a need for authentication.

The most common authentication protocols are SAML2p, WS-Federation and OpenID Connect – SAML2p being the most popular and the most widely deployed.

OpenID Connect is the newest of the three, but is considered to be the future because it has the most potential for modern applications. It was built for mobile application scenarios right from the start and is designed to be API friendly.

1.2 API Access

Applications have two fundamental ways with which they communicate with APIs – using the application identity, or delegating the user’s identity. Sometimes both methods need to be combined.

OAuth2 is a protocol that allows applications to request access tokens from a security token service and use them to communicate with APIs. This delegation reduces complexity in both the client applications as well as the APIs since authentication and authorization can be centralized.

1.3 OpenID Connect and OAuth 2.0 – better together

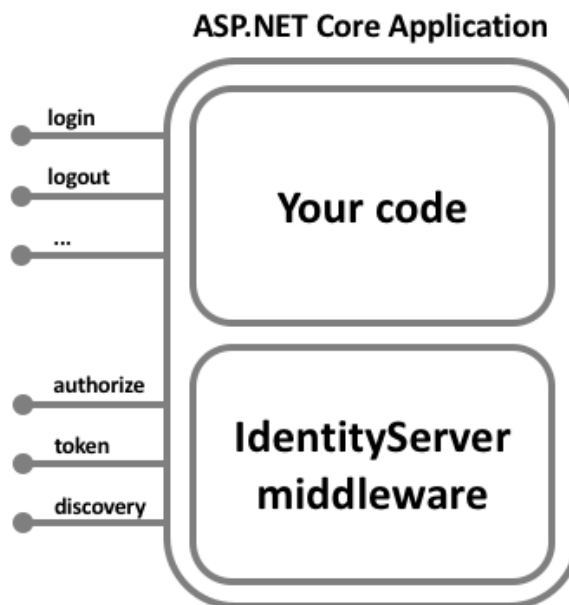
OpenID Connect and OAuth 2.0 are very similar – in fact OpenID Connect is an extension on top of OAuth 2.0. The two fundamental security concerns, authentication and API access, are combined into a single protocol - often with a single round trip to the security token service.

We believe that the combination of OpenID Connect and OAuth 2.0 is the best approach to secure modern applications for the foreseeable future. IdentityServer4 is an implementation of these two protocols and is highly optimized to solve the typical security problems of today’s mobile, native and web applications.

1.4 How IdentityServer4 can help

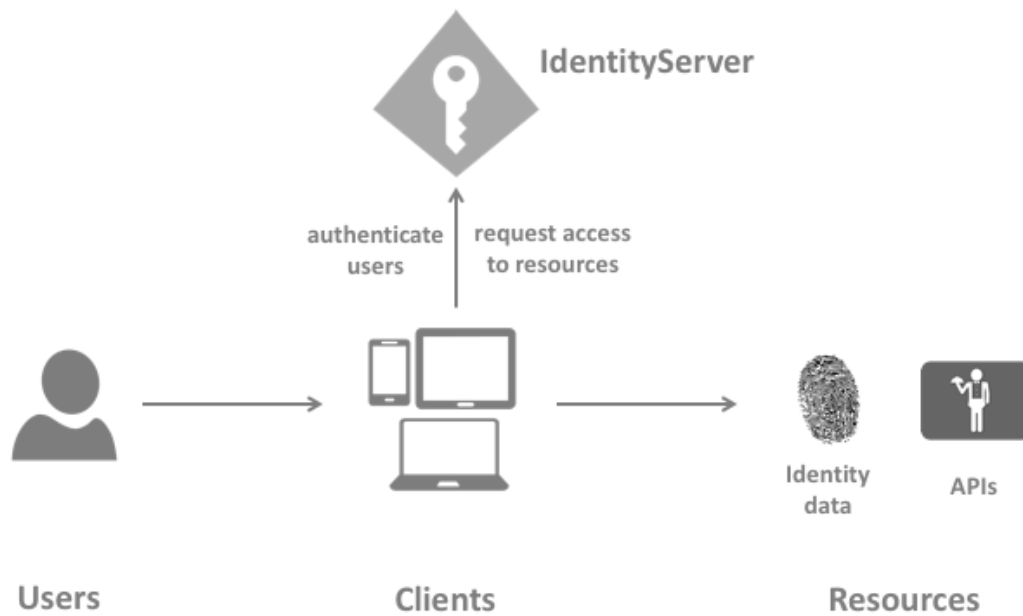
IdentityServer is middleware that adds the spec compliant OpenID Connect and OAuth 2.0 endpoints to an arbitrary ASP.NET Core application.

Typically, you build (or re-use) an application that contains a login and logout page (and maybe consent - depending on your needs), and the IdentityServer middleware adds the necessary protocol heads to it, so that client applications can talk to it using those standard protocols.



The hosting application can be as complex as you want, but we typically recommend to keep the attack surface as small as possible by including authentication related UI only.

The specs, documentation and object model use a certain terminology that you should be aware of.



2.1 IdentityServer

IdentityServer is an OpenID Connect provider - it implements the OpenID Connect and OAuth 2.0 protocols.

Different literature uses different terms for the same role - you probably also find security token service, identity provider, authorization server, IP-STS and more.

But they are in a nutshell all the same: a piece of software that issues security tokens to clients.

IdentityServer has a number of jobs and features - including:

- protect your resources
- authenticate users using a local account store or via an external identity provider
- provide session management and single sign-on
- manage and authenticate clients
- issue identity and access tokens to clients
- validate tokens

2.2 User

A user is a human that is using a registered client to access resources.

2.3 Client

A client is a piece of software that requests tokens from IdentityServer - either for authenticating a user (requesting an identity token) or for accessing a resource (requesting an access token). A client must be first registered with IdentityServer before it can request tokens.

Examples for clients are web applications, native mobile or desktop applications, SPAs, server processes etc.

2.4 Resources

Resources are something you want to protect with IdentityServer - either identity data of your users, or APIs.

Every resource has a unique name - and clients use this name to specify to which resources they want to get access to.

Identity data Identity information (aka claims) about a user, e.g. name or email address.

APIs APIs resources represent functionality a client wants to invoke - typically modelled as Web APIs, but not necessarily.

2.5 Identity Token

An identity token represents the outcome of an authentication process. It contains at a bare minimum an identifier for the user (called the *sub* aka subject claim) and information about how and when the user authenticated. It can contain additional identity data.

2.6 Access Token

An access token allows access to an API resource. Clients request access tokens and forward them to the API. Access tokens contain information about the client and the user (if present). APIs use that information to authorize access to their data.

Supported Specifications

IdentityServer implements the following specifications:

3.1 OpenID Connect

- OpenID Connect Core 1.0 ([spec](#))
- OpenID Connect Discovery 1.0 ([spec](#))
- OpenID Connect Session Management 1.0 - draft 28 ([spec](#))
- OpenID Connect Front-Channel Logout 1.0 - draft 02 ([spec](#))
- OpenID Connect Back-Channel Logout 1.0 - draft 04 ([spec](#))

3.2 OAuth 2.0

- OAuth 2.0 ([RFC 6749](#))
- OAuth 2.0 Bearer Token Usage ([RFC 6750](#))
- OAuth 2.0 Multiple Response Types ([spec](#))
- OAuth 2.0 Form Post Response Mode ([spec](#))
- OAuth 2.0 Token Revocation ([RFC 7009](#))
- OAuth 2.0 Token Introspection ([RFC 7662](#))
- Proof Key for Code Exchange ([RFC 7636](#))
- JSON Web Tokens for Client Authentication ([RFC 7523](#))
- OAuth 2.0 Device Authorization Grant ([RFC 8628](#))
- OAuth 2.0 Mutual TLS Client Authentication and Certificate-Bound Access Tokens ([RFC 8705](#))

- [JWT Secured Authorization Request \(draft\)](#)

IdentityServer consists of a number of nuget packages.

4.1 IdentityServer4 main repo

[github](#)

Contains the core IdentityServer object model, services and middleware as well as the EntityFramework and ASP.NET Identity integration.

nugets:

- [IdentityServer4](#)
- [IdentityServer4.EntityFramework](#)
- [IdentityServer4.AspNetIdentity](#)

4.2 Quickstart UI

[github](#)

Contains a simple starter UI including login, logout and consent pages.

4.3 Access token validation handler

[nuget](#) | [github](#)

ASP.NET Core authentication handler for validating tokens in APIs. The handler allows supporting both JWT and reference tokens in the same API.

4.4 Templates

`nuget | github`

Contains templates for the dotnet CLI.

4.5 Dev builds

In addition we publish CI builds to our package repository. Add the following `nuget.config` to your project:

```
<?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <packageSources>
      <clear />
      <add key="IdentityServer CI" value="https://pkgs.dev.azure.com/
↪netidentity/IdentityServer/_packaging/CI/nuget/v3/index.json" />
    </packageSources>
  </configuration>
```

Support and Consulting Options

We have several free and commercial support and consulting options for IdentityServer.

5.1 Free support

Free support is community-based and uses public forums

StackOverflow

There's an ever growing community of people using IdentityServer that monitor questions on StackOverflow. If time permits, we also try to answer as many questions as possible

You can subscribe to all IdentityServer4 related questions using this feed:

<https://stackoverflow.com/questions/tagged/?tagnames=identityserver4&sort=newest>

Please use the IdentityServer4 tag when asking new questions

Gitter

You can chat with other IdentityServer4 users in our Gitter chat room:

<https://gitter.im/IdentityServer/IdentityServer4>

Reporting a bug

If you think you have found a bug or unexpected behavior, please open an issue on the Github [issue tracker](#). We try to get back to you ASAP. Please understand that we also have day jobs, and might be too busy to reply immediately.

Also check the [contribution](#) guidelines before posting.

5.2 Commercial support

We are doing consulting, mentoring and custom software development around identity & access control architecture in general, and IdentityServer in particular. Please [get in touch](#) with us to discuss possible options.

Training

We are regularly doing workshops around identity & access control for modern applications. Check the agenda and upcoming public dates [here](#). We can also perform the training privately at your company. [Contact us](#) to request the training on-site.

AdminUI, WS-Federation, SAML2p, and FIDO2 support

There are commercial add-on products available from our partners, Rock Solid Knowledge, on identityserver.com.

CHAPTER 6

Demo Server

You can try IdentityServer4 with your favourite client library. We have a test instance at demo.identityserver.io. On the main page you can find instructions on how to configure your client and how to call an API.

We are very open to community contributions, but there are a couple of guidelines you should follow so we can handle this without too much effort.

7.1 How to contribute?

The easiest way to contribute is to open an issue and start a discussion. Then we can decide if and how a feature or a change could be implemented. If you should submit a pull request with code changes, start with a description, only make the minimal changes to start with and provide tests that cover those changes.

Also read this first: [Being a good open source citizen](#)

7.2 General feedback and discussions?

Please start a discussion on the [core repo issue tracker](#).

7.3 Bugs and feature requests?

Please log a new issue in the appropriate GitHub repo:

- [Core](#)
- [AccessTokenValidation](#)

7.4 Contributing code and content

You will need to sign a Contributor License Agreement before you can contribute any code or content. This is an automated process that will start after you opened a pull request.

7.5 Contribution projects

We very much appreciate if you start a contribution project (e.g. support for Database X or Configuration Store Y). Tell us about it so we can tweet and link it in our docs.

We generally don't want to take ownership of those contribution libraries, we are already really busy supporting the core projects.

Naming conventions

As of October 2017, the IdentityServer4.* nuget namespace is reserved for our packages. Please use the following naming conventions:

`YourProjectName.IdentityServer4`

or

`IdentityServer4.Contrib.YourProjectName`

The quickstarts provide step by step instructions for various common IdentityServer scenarios. They start with the absolute basics and become more complex - it is recommended you do them in order.

- adding IdentityServer to an ASP.NET Core application
- configuring IdentityServer
- issuing tokens for various clients
- securing web applications and APIs
- adding support for EntityFramework based configuration
- adding support for ASP.NET Identity

Every quickstart has a reference solution - you can find the code in the [samples](#) folder.

8.1 Preparation

The first thing you should do is install our templates:

```
dotnet new -i IdentityServer4.Templates
```

They will be used as a starting point for the various tutorials.

Note: If you are using private NuGet sources do not forget to add the `-nuget-source` parameter: `-nuget-source https://api.nuget.org/v3/index.json`

OK - let's get started!

Note: The quickstarts target the latest version of IdentityServer and ASP.NET Core (3.0) - there are also quickstarts for [ASP.NET Core 2](#) and [ASP.NET Core 1](#).

Protecting an API using Client Credentials

The following Identity Server 4 quickstart provides step by step instructions for various common IdentityServer scenarios. These start with the absolute basics and become more complex as they progress. We recommend that you follow them in sequence.

To see the full list, please go to [IdentityServer4 Quickstarts Overview](#)

This first quickstart is the most basic scenario for protecting APIs using IdentityServer. In this quickstart you define an API and a Client with which to access it. The client will request an access token from the Identity Server using its client ID and secret will then use the token to gain access to the API.

9.1 Source Code

As with all of these quickstarts you can find the source code for it in the [IdentityServer4](#) repository. The project for this quickstart is [Quickstart #1: Securing an API using Client Credentials](#)

9.2 Preparation

The IdentityServer templates for the dotnet CLI are a good starting point for the quickstarts. To install the templates open a console window and type the following command:

```
dotnet new -i IdentityServer4.Templates
```

They will be used as a starting point for the various tutorials.

9.3 Setting up the ASP.NET Core application

First create a directory for the application - then use our template to create an ASP.NET Core application that includes a basic IdentityServer setup, e.g.:

```
md quickstart
cd quickstart

md src
cd src

dotnet new is4empty -n IdentityServer
```

This will create the following files:

- `IdentityServer.csproj` - the project file and a `Properties\launchSettings.json` file
- `Program.cs` and `Startup.cs` - the main application entry point
- `Config.cs` - IdentityServer resources and clients configuration file

You can now use your favorite text editor to edit or view the files. If you want to have Visual Studio support, you can add a solution file like this:

```
cd ..
dotnet new sln -n Quickstart
```

and let it add your IdentityServer project (keep this command in mind as we will create other projects below):

```
dotnet sln add .\src\IdentityServer\IdentityServer.csproj
```

Note: The protocol used in this Template is `https` and the port is set to 5001 when running on Kestrel or a random one on IISExpress. You can change that in the `Properties\launchSettings.json` file. For production scenarios you should always use `https`.

9.4 Defining an API Resource

An API is a resource in your system that you want to protect. Resource definitions can be loaded in many ways, the template you used to create the project above shows how to use a “code as configuration” approach.

The `Config.cs` is already created for you. Open it, update the code to look like this:

```
public static class Config
{
    public static IEnumerable<ApiResource> Apis =>
        new List<ApiResource>
        {
            new ApiResource("api1", "My API")
        };
}
```

(see the full file [here](#)).

Note: If you will be using this in production it is important to give your API a logical name. Developers will be using this to connect to your api through your Identity server. It should describe your api in simple terms to both developers and users.

9.5 Defining the client

The next step is to define a client application that we will use to access our new API.

For this scenario, the client will not have an interactive user, and will authenticate using the so called client secret with IdentityServer.

For this, add a client definition:

```
public static IEnumerable<Client> Clients =>
    new List<Client>
    {
        new Client
        {
            ClientId = "client",

            // no interactive user, use the clientid/secret for authentication
            AllowedGrantTypes = GrantTypes.ClientCredentials,

            // secret for authentication
            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },

            // scopes that client has access to
            AllowedScopes = { "api1" }
        }
    };
```

You can think of the ClientId and the ClientSecret as the login and password for your application itself. It identifies your application to the identity server so that it knows which application is trying to connect to it.

9.6 Configuring IdentityServer

Loading the resource and client definitions happens in `Startup.cs` - update the code to look like this:

```
public void ConfigureServices(IServiceCollection services)
{
    var builder = services.AddIdentityServer()
        .AddInMemoryApiResources(Config.Apis)
        .AddInMemoryClients(Config.Clients);

    // omitted for brevity
}
```

That's it - your identity server should now be configured. If you run the server and navigate the browser to `https://localhost:5001/.well-known/openid-configuration`, you should see the so-called discovery document. The discovery document is a standard endpoint in identity servers. The discovery document will be used by your clients and APIs to download the necessary configuration data.



```

{
  "issuer": "https://localhost:5001/",
  "jwks_uri": "https://localhost:5001/.well-known/openid-configuration/jwks",
  "authorization_endpoint": "https://localhost:5001/connect/authorize",
  "token_endpoint": "https://localhost:5001/connect/token",
  "userinfo_endpoint": "https://localhost:5001/connect/userinfo",
  "end_session_endpoint": "https://localhost:5001/connect/endsession",
  "check_session_iframe": "https://localhost:5001/connect/checksession",
  "revocation_endpoint": "https://localhost:5001/connect/revocation",
  "introspection_endpoint": "https://localhost:5001/connect/introspect",
  "device_authorization_endpoint": "https://localhost:5001/connect/deviceauthorization",
  "frontchannel_logout_supported": true,
  "frontchannel_logout_session_supported": true,
  "backchannel_logout_supported": true,
  "backchannel_logout_session_supported": true,
  "scopes_supported": [
    "api",
    "offline_access"
  ],
  "claims_supported": [],
  "grant_types_supported": [
    "authorization_code",
    "client_credentials",
    "refresh_token",
    "implicit",
    "urn:ietf:params:oauth:grant-type:device_code"
  ],
  "response_types_supported": [
    "code",
    "token",
    "id_token",
    "id_token token",
    "code id_token",
    "code token",
    "code id_token token"
  ],
  "response_modes_supported": [
    "form_post",
    "query",
    "fragment"
  ],
  "token_endpoint_auth_methods_supported": [
    "client_secret_basic",
    "client_secret_post"
  ],
  "id_token_signing_alg_values_supported": [
    "RS256"
  ],
  "subject_types_supported": [
    "public"
  ],
  "code_challenge_methods_supported": [
    "plain",
    "S256"
  ],
  "request_parameter_supported": true
}

```

At first startup, IdentityServer will create a developer signing key for you, it's a file called `tempkey.rsa`. You don't have to check that file into your source control, it will be re-created if it is not present.

9.7 Adding an API

Next, add an API to your solution.

You can either use the ASP.NET Core Web API template from Visual Studio or use the .NET CLI to create the API

project as we do here. Run from within the `src` folder the following command:

```
dotnet new web -n Api
```

Then add it to the solution by running the following commands:

```
cd ..
dotnet sln add .\src\Api\Api.csproj
```

Configure the API application to run on `https://localhost:6001` only. You can do this by editing the `launchSettings.json` file inside the Properties folder. Change the application URL setting to be:

```
"applicationUrl": "https://localhost:6001"
```

9.7.1 The controller

Add a new class called `IdentityController`:

```
[Route("identity")]
[Authorize]
public class IdentityController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        return new JsonResult(from c in User.Claims select new { c.Type, c.Value });
    }
}
```

This controller will be used later to test the authorization requirement, as well as visualize the claims identity through the eyes of the API.

9.7.2 Adding a Nuget Dependency

In order for the configuration step to work the nuget package dependency has to be added, run this command in the root directory.

```
dotnet add .\src\Api\Api.csproj package Microsoft.AspNetCore.Authentication.JwtBearer
```

9.7.3 Configuration

The last step is to add the authentication services to DI (dependency injection) and the authentication middleware to the pipeline. These will:

- validate the incoming token to make sure it is coming from a trusted issuer
- validate that the token is valid to be used with this api (aka audience)

Update `Startup` to look like this:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
    }
}
```

(continues on next page)

(continued from previous page)

```
services.AddAuthentication("Bearer")
    .AddJwtBearer("Bearer", options =>
    {
        options.Authority = "https://localhost:5001";
        options.RequireHttpsMetadata = false;

        options.Audience = "api1";
    });

public void Configure(IApplicationBuilder app)
{
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

- `AddAuthentication` adds the authentication services to DI and configures Bearer as the default scheme.
- `UseAuthentication` adds the authentication middleware to the pipeline so authentication will be performed automatically on every call into the host.
- `UseAuthorization` adds the authorization middleware to make sure, our API endpoint cannot be accessed by anonymous clients.

Navigating to the controller `https://localhost:6001/identity` on a browser should return a 401 status code. This means your API requires a credential and is now protected by IdentityServer.

9.8 Creating the client

The last step is to write a client that requests an access token, and then uses this token to access the API. For that, add a console project to your solution, remember to create it in the `src`:

```
dotnet new console -n Client
```

Then as before, add it to your solution using:

```
cd ..
dotnet sln add .\src\Client\Client.csproj
```

The token endpoint at IdentityServer implements the OAuth 2.0 protocol, and you could use raw HTTP to access it. However, we have a client library called IdentityModel, that encapsulates the protocol interaction in an easy to use API.

Add the IdentityModel NuGet package to your client. This can be done either via Visual Studio's Nuget Package manager or dotnet CLI:

```
cd src
cd client
dotnet add package IdentityModel
```

IdentityModel includes a client library to use with the discovery endpoint. This way you only need to know the base-address of IdentityServer - the actual endpoint addresses can be read from the metadata:

```
// discover endpoints from metadata
var client = new HttpClient();
var disco = await client.GetDiscoveryDocumentAsync("https://localhost:5001");
if (disco.IsError)
{
    Console.WriteLine(disco.Error);
    return;
}
```

Next you can use the information from the discovery document to request a token to IdentityServer to access api1:

```
// request token
var tokenResponse = await client.RequestClientCredentialsTokenAsync(new
    ClientCredentialsTokenRequest
    {
        Address = disco.TokenEndpoint,

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

if (tokenResponse.IsError)
{
    Console.WriteLine(tokenResponse.Error);
    return;
}

Console.WriteLine(tokenResponse.Json);
```

(full file can be found [here](#))

Note: Copy and paste the access token from the console to [jwt.ms](#) to inspect the raw token.

9.9 Calling the API

To send the access token to the API you typically use the HTTP Authorization header. This is done using the `SetBearerToken` extension method:

```
// call api
var apiClient = new HttpClient();
apiClient.SetBearerToken(tokenResponse.AccessToken);

var response = await apiClient.GetAsync("https://localhost:6001/identity");
if (!response.IsSuccessStatusCode)
{
```

(continues on next page)

(continued from previous page)

```
        Console.WriteLine(response.StatusCode);
    }
    else
    {
        var content = await response.Content.ReadAsStringAsync();
        Console.WriteLine(JArray.Parse(content));
    }
}
```

(If you are in Visual Studio you can right-click on the solution and select “Multiple Startup Projects”, and ensure the Api and IdentityServer will start; then run the solution; then, to step through the Client code, you can right-click on the “Client” project and select Debug... Start New Instance). The output should look like this:

```
C:\WINDOWS\system32\cmd.exe
```

```
{  
  "access_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjcwMDI4ZjE5YyZmYxZDJiYmYzZGQ4NTJiMTFkNGUwIiwidHlwIjoia  
ldUIIn0.eyJyYmYiOiJEONzISNzU5NzksImV4cCI6MTQ3Mjk3OTU3OSwiaXNzIjoiaHR0cDovL2xvY2Fsag9zdDo1MDAwIiwiaXVkaWoiOia  
HR0cDovL2xvY2Fsag9zdDo1MDAwL3Jlc291cmNlcyIsImNsawVudF9pZCI6ImNsawVudCIsInNjb3B1IjoiaYXBpMSJ9.mEgDNNLQrk0j  
87Svt0swjZgTEnZ6q1C0USI7ZMFdzTm8KqH3VOIP4_15TzbVbiKFmjtknoBCDNBr33TLE6VMFGK1XX8gEtCOX-RJGXgP2_hZrqXWMXx8  
wf4tW8-adqf2XHUIed7IGrN-UdknM-boThuHBGiW0wp5XNQf260-XGRwcutb1lpIUcmB76xVIKgu4MEzm5mc1UQXkADaus3xJjxWCc7k  
PWjUcJd8credJE13fAu8znZfvx8isK_G6Ix1pCUGOdNTN0sfupncx-syqGLxhD8oXfeEmikx0vlbwbp-fSpT9-4Qm8LAusI-Nj_7jlRm  
ueYSwwY7N6ovlgUnig",  
  "expires_in": 3600,  
  "token_type": "Bearer"  
}  
  
[  
  {  
    "type": "nbf",  
    "value": "1472979579"  
  },  
  {  
    "type": "exp",  
    "value": "1472979579"  
  },  
  {  
    "type": "iss",  
    "value": "http://localhost:5000"  
  },  
  {  
    "type": "aud",  
    "value": "http://localhost:5000/resources"  
  },  
  {  
    "type": "client_id",  
    "value": "client"  
  },  
  {  
    "type": "scope",  
    "value": "api1"  
  }  
]
```

Note: By default an access token will contain claims about the scope, lifetime (nbf and exp), the client ID (client_id) and the issuer name (iss).

9.10 Further experiments

This walkthrough focused on the success path so far

- client was able to request token
- client could use the token to access the API

You can now try to provoke errors to learn how the system behaves, e.g.

- try to connect to IdentityServer when it is not running (unavailable)
- try to use an invalid client id or secret to request the token
- try to ask for an invalid scope during the token request
- try to call the API when it is not running (unavailable)
- don't send the token to the API
- configure the API to require a different scope than the one in the token

CHAPTER 10

Interactive Applications with ASP.NET Core

Note: For any pre-requisites (like e.g. templates) have a look at the *overview* first.

In this quickstart we want to add support for interactive user authentication via the OpenID Connect protocol to our IdentityServer we built in the previous chapter.

Once that is in place, we will create an MVC application that will use IdentityServer for authentication.

10.1 Adding the UI

All the protocol support needed for OpenID Connect is already built into IdentityServer. You need to provide the necessary UI parts for login, logout, consent and error.

While the look & feel as well as the exact workflows will probably always differ in every IdentityServer implementation, we provide an MVC-based sample UI that you can use as a starting point.

This UI can be found in the [Quickstart UI repo](#). You can clone or download this repo and drop the controllers, views, models and CSS into your IdentityServer web application.

Alternatively you can use the .NET CLI (run from within the `src/IdentityServer` folder):

```
dotnet new is4ui
```

Once you have added the MVC UI, you will also need to enable MVC, both in the DI system and in the pipeline. When you look at `Startup.cs` you will find comments in the `ConfigureServices` and `Configure` method that tell you how to enable MVC.

Note: There is also a template called `is4inmem` which combines a basic IdentityServer including the standard UI.

Run the IdentityServer application, you should now see a home page.

Spend some time inspecting the controllers and models - especially the `AccountController` which is the main UI entry point. The better you understand them, the easier it will be to make future modifications. Most of the code lives in the “Quickstart” folder using a “feature folder” style. If this style doesn’t suit you, feel free to organize the code in any way you want.

10.2 Creating an MVC client

Next you will create an MVC application. Use the ASP.NET Core “Web Application” (i.e. MVC) template for that. run from the `src` folder:

```
dotnet new mvc -n MVCClient
cd ..
dotnet sln add .\src\MVCClient\MVCClient.csproj
```

Once you’ve created the project, configure the application to run on port 5002.

To add support for OpenID Connect authentication to the MVC application, you first need to add the nuget package containing the OpenID Connect handler to your project, e.g.:

```
dotnet add package Microsoft.AspNetCore.Authentication.OpenIdConnect
```

..then add the following to `ConfigureServices` in `Startup`:

```
using System.IdentityModel.Tokens.Jwt;

// ...

JwtSecurityTokenHandler.DefaultMapInboundClaims = false;

services.AddAuthentication(options =>
{
    options.DefaultScheme = "Cookies";
    options.DefaultChallengeScheme = "oidc";
})
.AddCookie("Cookies")
.AddOpenIdConnect("oidc", options =>
{
    options.Authority = "https://localhost:5001";
    options.RequireHttpsMetadata = false;

    options.ClientId = "mvc";
    options.ClientSecret = "secret";
    options.ResponseType = "code";

    options.SaveTokens = true;
});
```

`AddAuthentication` adds the authentication services to DI.

We are using a cookie to locally sign-in the user (via “Cookies” as the `DefaultScheme`), and we set the `DefaultChallengeScheme` to `oidc` because when we need the user to login, we will be using the OpenID Connect protocol.

We then use `AddCookie` to add the handler that can process cookies.

Finally, `AddOpenIdConnect` is used to configure the handler that perform the OpenID Connect protocol. The `Authority` indicates where the trusted token service is located. We then identify this client via the `ClientId` and

the `ClientSecret`. `SaveTokens` is used to persist the tokens from IdentityServer in the cookie (as they will be needed later).

Note: We use the so called `authorization code` flow with PKCE to connect to the OpenID Connect provider. See [here](#) for more information on protocol flows.

And then to ensure the authentication services execute on each request, add `UseAuthentication` to `Configure` in `Startup`:

```
app.UseStaticFiles();

app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapDefaultControllerRoute()
        .RequireAuthorization();
});
```

Note: The `RequireAuthorization` method disables anonymous access for the entire application. You can also use the `[Authorize]` attribute, if you want to specify that on a per controller or action method basis.

Also modify the home view to display the claims of the user as well as the cookie properties:

```
@using Microsoft.AspNetCore.Authentication

<h2>Claims</h2>

<dl>
    @foreach (var claim in User.Claims)
    {
        <dt>@claim.Type</dt>
        <dd>@claim.Value</dd>
    }
</dl>

<h2>Properties</h2>

<dl>
    @foreach (var prop in (await Context.AuthenticateAsync()).Properties.Items)
    {
        <dt>@prop.Key</dt>
        <dd>@prop.Value</dd>
    }
</dl>
```

If you now navigate to the application using the browser, a redirect attempt will be made to IdentityServer - this will result in an error because the MVC client is not registered yet.

10.3 Adding support for OpenID Connect Identity Scopes

Similar to OAuth 2.0, OpenID Connect also uses the scopes concept. Again, scopes represent something you want to protect and that clients want to access. In contrast to OAuth, scopes in OIDC don't represent APIs, but identity data like user id, name or email address.

Add support for the standard `openid` (subject id) and `profile` (first name, last name etc..) scopes by amending the `Ids` property in `Config.cs`:

```
public static IEnumerable<IdentityResource> Ids =>
    new List<IdentityResource>
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Profile(),
    };
```

Register the identity resources with IdentityServer in `startup.cs`:

```
var builder = services.AddIdentityServer()
    .AddInMemoryIdentityResources(Config.Ids)
    .AddInMemoryApiResources(Config.Apis)
    .AddInMemoryClients(Config.Clients);
```

Note: All standard scopes and their corresponding claims can be found in the [OpenID Connect specification](#)

10.4 Adding Test Users

The sample UI also comes with an in-memory “user database”. You can enable this in IdentityServer by adding the `AddTestUsers` extension method:

```
var builder = services.AddIdentityServer()
    .AddInMemoryIdentityResources(Config.Ids)
    .AddInMemoryApiResources(Config.Apis)
    .AddInMemoryClients(Config.Clients)
    .AddTestUsers(TestUsers.Users);
```

When you navigate to the `TestUsers` class, you can see that two users called `alice` and `bob` as well as some identity claims are defined. You can use those users to login.

10.5 Adding the MVC Client to the IdentityServer Configuration

The last step is to add a new configuration entry for the MVC client to IdentityServer.

OpenID Connect-based clients are very similar to the OAuth 2.0 clients we added so far. But since the flows in OIDC are always interactive, we need to add some redirect URLs to our configuration.

The client list should look like this:

```
public static IEnumerable<Client> Clients =>
    new List<Client>
    {
```

(continues on next page)

(continued from previous page)

```

// machine to machine client (from quickstart 1)
new Client
{
    ClientId = "client",
    ClientSecrets = { new Secret("secret".Sha256()) },

    AllowedGrantTypes = GrantTypes.ClientCredentials,
    // scopes that client has access to
    AllowedScopes = { "api1" }
},
// interactive ASP.NET Core MVC client
new Client
{
    ClientId = "mvc",
    ClientSecrets = { new Secret("secret".Sha256()) },

    AllowedGrantTypes = GrantTypes.Code,
    RequireConsent = false,
    RequirePkce = true,

    // where to redirect to after login
    RedirectUri = { "http://localhost:5002/signin-oidc" },

    // where to redirect to after logout
    PostLogoutRedirectUri = { "http://localhost:5002/signout-callback-oidc" }

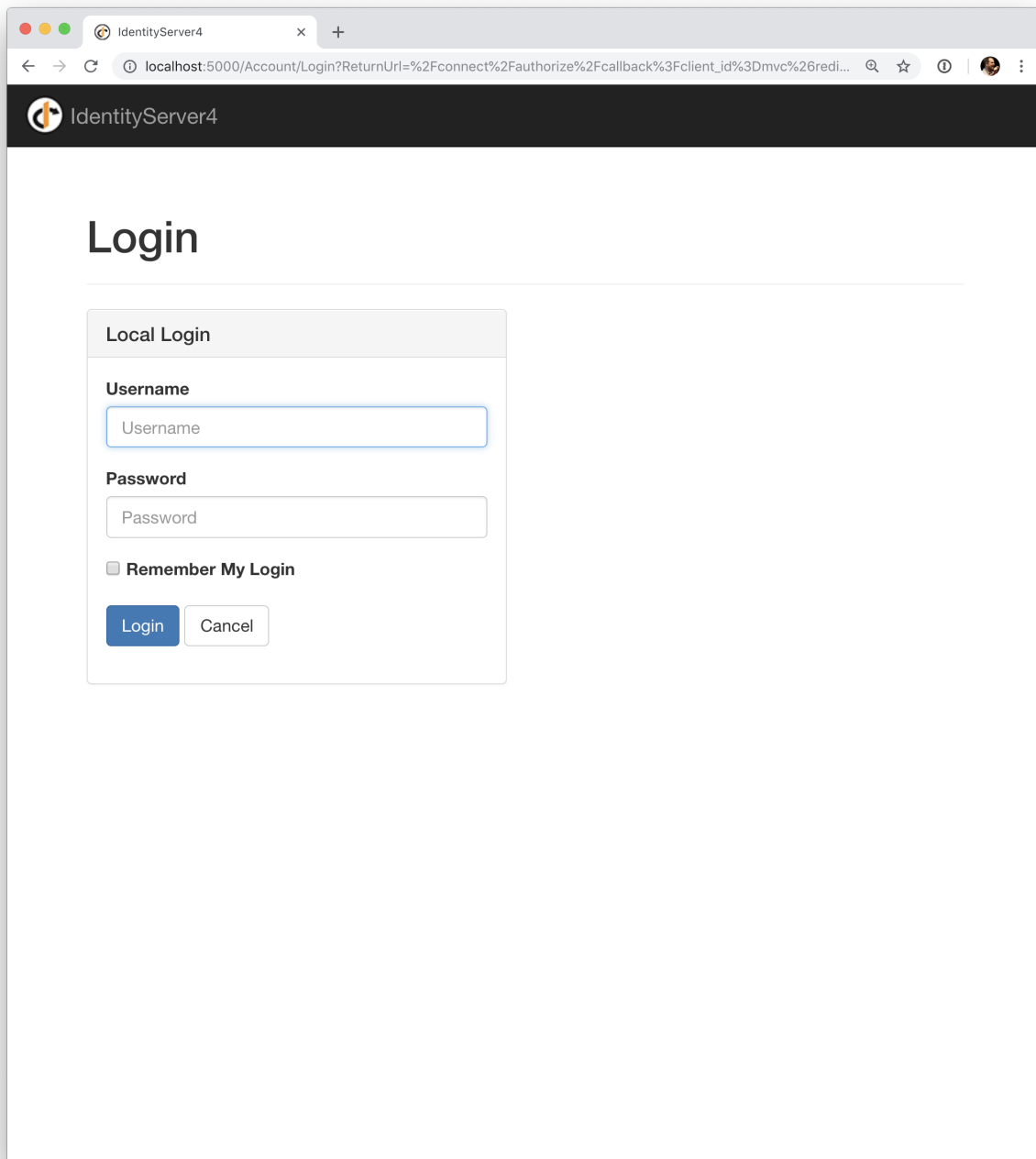
    AllowedScopes = new List<string>
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile
    }
};

```

10.6 Testing the client

Now finally everything should be in place for the new MVC client.

Trigger the authentication handshake by navigating to the protected controller action. You should see a redirect to the login page at IdentityServer.



After that, IdentityServer will redirect back to the MVC client, where the OpenID Connect authentication handler processes the response and signs-in the user locally by setting a cookie. Finally the MVC view will show the contents of the cookie.

The exact protocol steps are implemented inside the OpenID Connect handler, simply add the following code to some controller to trigger the sign-out:

```
public IActionResult Logout ()
{
    return SignOut ("Cookies", "oidc");
}
```

This will clear the local cookie and then redirect to IdentityServer. IdentityServer will clear its cookies and then give the user a link to return back to the MVC application.

10.8 Further Experiments

Feel free to add more claims to the test users - and also more identity resources.

The process for defining an identity resource is as follows:

- add a new identity resource to the list - give it a name and specify which claims should be returned when this resource is requested
- give the client access to the resource via the `AllowedScopes` property on the client configuration
- request the resource by adding it to the `Scopes` collection on the OpenID Connect handler configuration in the client

It is also noteworthy, that the retrieval of claims for tokens is an extensibility point - `IProfileService`. Since we are using `AddTestUsers`, the `TestUserProfileService` is used by default. You can inspect the source code [here](#) to see how it works.

10.9 Adding Support for External Authentication

Next we will add support for external authentication. This is really easy, because all you really need is an ASP.NET Core compatible authentication handler.

ASP.NET Core itself ships with support for Google, Facebook, Twitter, Microsoft Account and OpenID Connect. In addition you can find implementations for many other authentication providers [here](#).

10.10 Adding Google support

To be able to use Google for authentication, you first need to register with them. This is done at their developer [console](#). Create a new project, enable the Google+ API and configure the callback address of your local IdentityServer by adding the `/signin-google` path to your base-address (e.g. <https://localhost:5001/signin-google>).

The developer console will show you a client ID and secret issued by Google - you will need that in the next step.

Add the Google authentication handler to the DI of the IdentityServer host. This is done by first adding the `Microsoft.AspNetCore.Authentication.Google` nuget package and then adding this snippet to `ConfigureServices` in `Startup`:

```
services.AddAuthentication()
    .AddGoogle("Google", options =>
    {
        options.SignInScheme = IdentityServerConstants.
        ↪ExternalCookieAuthenticationScheme;
```

(continues on next page)

(continued from previous page)

```
options.ClientId = "<insert here>";
options.ClientSecret = "<insert here>";
});
```

By default, `IdentityServer` configures a cookie handler specifically for the results of external authentication (with the scheme based on the constant `IdentityServerConstants.ExternalCookieAuthenticationScheme`). The configuration for the Google handler is then using that cookie handler.

Now run the MVC client and try to authenticate - you will see a Google button on the login page:

[illegible]

After authentication with the MVC client, you can see that the claims are now being sourced from Google data.

Note: If you are interested in the magic that automatically renders the Google button on the login page, inspect the

BuildLoginViewModel method on the AccountController.

10.11 Further experiments

You can add an additional external provider. We have a [cloud-hosted demo](#) version of IdentityServer4 which you can integrate using OpenID Connect.

Add the OpenId Connect handler to DI:

```
services.AddAuthentication()
    .AddGoogle("Google", options =>
    {
        options.SignInScheme = IdentityServerConstants.
↪ExternalCookieAuthenticationScheme;

        options.ClientId = "<insert here>";
        options.ClientSecret = "<insert here>";
    })
    .AddOpenIdConnect("oidc", "Demo IdentityServer", options =>
    {
        options.SignInScheme = IdentityServerConstants.
↪ExternalCookieAuthenticationScheme;
        options.SignOutScheme = IdentityServerConstants.SignoutScheme;
        options.SaveTokens = true;

        options.Authority = "https://demo.identityserver.io/";
        options.ClientId = "native.code";
        options.ClientSecret = "secret";
        options.ResponseType = "code";

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

And now a user should be able to use the cloud-hosted demo identity provider.

Note: The quickstart UI auto-provisions external users. As an external user logs in for the first time, a new local user is created, and all the external claims are copied over and associated with the new user. The way you deal with such a situation is completely up to you though. Maybe you want to show some sort of registration UI first. The source code for the default quickstart can be found [here](#). The controller where auto-provisioning is executed can be found [here](#).

ASP.NET Core and API access

In the previous quickstarts we explored both API access and user authentication. Now we want to bring the two parts together.

The beauty of the OpenID Connect & OAuth 2.0 combination is, that you can achieve both with a single protocol and a single exchange with the token service.

So far we only asked for identity resources during the token request, once we start also including API resources, IdentityServer will return two tokens: the identity token containing the information about the authentication and session, and the access token to access APIs on behalf of the logged on user.

11.1 Modifying the client configuration

Updating the client configuration in IdentityServer is straightforward - we simply need to add the `api1` resource to the allowed scopes list. In addition we enable support for refresh tokens via the `AllowOfflineAccess` property:

```
new Client
{
    ClientId = "mvc",
    ClientSecrets = { new Secret("secret".Sha256()) },

    AllowedGrantTypes = GrantTypes.Code,
    RequireConsent = false,
    RequirePkce = true,

    // where to redirect to after login
    RedirectUri = { "http://localhost:5002/signin-oidc" },

    // where to redirect to after logout
    PostLogoutRedirectUri = { "http://localhost:5002/signout-callback-oidc" },

    AllowedScopes = new List<string>
    {
```

(continues on next page)

(continued from previous page)

```
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        "apil"
    },

    AllowOfflineAccess = true
}
```

11.2 Modifying the MVC client

All that's left to do now in the client is to ask for the additional resources via the scope parameter. This is done in the OpenID Connect handler configuration:

```
services.AddAuthentication(options =>
{
    options.DefaultScheme = "Cookies";
    options.DefaultChallengeScheme = "oidc";
})
.AddCookie("Cookies")
.AddOpenIdConnect("oidc", options =>
{
    options.Authority = "https://localhost:5001";
    options.RequireHttpsMetadata = false;

    options.ClientId = "mvc";
    options.ClientSecret = "secret";
    options.ResponseType = "code";

    options.SaveTokens = true;

    options.Scope.Add("apil");
    options.Scope.Add("offline_access");
});
```

Since `SaveTokens` is enabled, ASP.NET Core will automatically store the resulting access and refresh token in the authentication session. You should be able to inspect the data on the page that prints out the contents of the session that you created earlier.

11.3 Using the access token

You can access the tokens in the session using the standard ASP.NET Core extension methods that you can find in the `Microsoft.AspNetCore.Authentication` namespace:

```
var accessToken = await HttpContext.GetTokenAsync("access_token");
var refreshToken = await HttpContext.GetTokenAsync("refresh_token");
```

For accessing the API using the access token, all you need to do is retrieve the token, and set it on your `HttpClient`:

```
public async Task<IActionResult> CallApi()
{
    var accessToken = await HttpContext.GetTokenAsync("access_token");
```

(continues on next page)

(continued from previous page)

```
var client = new HttpClient();
client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer
↪", accessToken);
var content = await client.GetStringAsync("https://localhost:6001/identity");

ViewBag.Json = JObject.Parse(content).ToString();
return View("json");
}
```

Create a view called json.cshtml that outputs the json like this:

```
<pre>@ViewBag.Json</pre>
```

Make sure the API is running, start the MVC client and call `/home/CallApi` after authentication.

11.4 Managing the access token

By far the most complex task for a typical client is to manage the access token. You typically want to

- request the access and refresh token at login time
- cache those tokens
- use the access token to call APIs until it expires
- use the refresh token to get a new access token
- start over

ASP.NET Core has many built-in facility that can help you with those tasks (like caching or sessions), but there is still quite some work left to do. Feel free to have a look at [this](#) library, which can automate many of the boilerplate tasks.

CHAPTER 12

Adding a JavaScript client

Note: For any pre-requisites (like e.g. templates) have a look at the *overview* first.

This quickstart will show how to build a browser-based JavaScript client application (sometimes referred to as a “Single Page Application” or “SPA”).

The user will login to IdentityServer, invoke the web API with an access token issued by IdentityServer, and logout of IdentityServer. All of this will be driven from the JavaScript running in the browser.

12.1 New Project for the JavaScript client

Create a new project for the JavaScript application. It can simply be an empty web project, an empty ASP.NET Core application, or something else like a Node.js application. This quickstart will use an ASP.NET Core application.

Create a new “Empty” ASP.NET Core web application in the `~/src` directory. You can use Visual Studio or do this from the command line:

```
md JavaScriptClient
cd JavaScriptClient
dotnet new web
```

As we have done before, with other client projects, add this project also to your solution. run this from the rootfolder which has the sln file:

```
dotnet sln add .\src\JavaScriptClient\JavaScriptClient.csproj
```

12.2 Modify hosting

Modify the *JavaScriptClient* project to run on port 5003.

12.3 Add the static file middleware

Given that this project is designed to run client-side, all we need ASP.NET Core to do is to serve up the static HTML and JavaScript files that will make up our application. The static file middleware is designed to do this.

Register the static file middleware in *Startup.cs* in the `Configure` method (and at the same time remove everything else):

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

This middleware will now serve up static files from the application's `~/wwwroot` folder. This is where we will put our HTML and JavaScript files. If that folder does not exist in your project, create it now.

12.4 Reference oidc-client

In one of the previous quickstarts in the ASP.NET Core MVC-based client project we used a library to handle the OpenID Connect protocol. In this quickstart in the *JavaScriptClient* project we need a similar library, except one that works in JavaScript and is designed to run in the browser. The [oidc-client library](#) is one such library. It is available via [NPM](#), [Bower](#), as well as a [direct download](#) from github.

NPM

If you want to use NPM to download *oidc-client*, then run these commands from your *JavaScriptClient* project directory:

```
npm i oidc-client
copy node_modules\oidc-client\dist\* wwwroot
```

This downloads the latest *oidc-client* package locally, and then copies the relevant JavaScript files into `~/wwwroot` so they can be served up by your application.

Manual download

If you want to simply download the *oidc-client* JavaScript files manually, browse to [the GitHub repository](#) and download the JavaScript files. Once downloaded, copy them into `~/wwwroot` so they can be served up by your application.

12.5 Add your HTML and JavaScript files

Next is to add your HTML and JavaScript files to `~/wwwroot`. We will have two HTML files and one application-specific JavaScript file (in addition to the *oidc-client.js* library). In `~/wwwroot`, add a HTML file named *index.html* and *callback.html*, and add a JavaScript file called *app.js*.

index.html

This will be the main page in our application. It will simply contain the HTML for the buttons for the user to login, logout, and call the web API. It will also contain the `<script>` tags to include our two JavaScript files. It will also contain a `<pre>` used for showing messages to the user.

It should look like this:


```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
</head>
<body>
  <button id="login">Login</button>
  <button id="api">Call API</button>
  <button id="logout">Logout</button>

  <pre id="results"></pre>

  <script src="oidc-client.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

app.js

This will contain the main code for our application. The first thing is to add a helper function to log messages to the `<pre>`:

```
function log() {
  document.getElementById('results').innerText = '';

  Array.prototype.forEach.call(arguments, function (msg) {
    if (msg instanceof Error) {
      msg = "Error: " + msg.message;
    }
    else if (typeof msg !== 'string') {
      msg = JSON.stringify(msg, null, 2);
    }
    document.getElementById('results').innerHTML += msg + '\r\n';
  });
}
```

Next, add code to register click event handlers to the three buttons:

```
document.getElementById("login").addEventListener("click", login, false);
document.getElementById("api").addEventListener("click", api, false);
document.getElementById("logout").addEventListener("click", logout, false);
```

Next, we can use the `UserManager` class from the `oidc-client` library to manage the OpenID Connect protocol. It requires similar configuration that was necessary in the MVC Client (albeit with different values). Add this code to configure and instantiate the `UserManager`:

```
var config = {
  authority: "http://localhost:5000",
  client_id: "js",
  redirect_uri: "http://localhost:5003/callback.html",
  response_type: "code",
  scope: "openid profile api1",
  post_logout_redirect_uri : "http://localhost:5003/index.html",
};
var mgr = new Oidc.UserManager(config);
```

Next, the `UserManager` provides a `getUser` API to know if the user is logged into the JavaScript application. It

uses a JavaScript Promise to return the results asynchronously. The returned `User` object has a `profile` property which contains the claims for the user. Add this code to detect if the user is logged into the JavaScript application:

```
mgr.getUser().then(function (user) {
    if (user) {
        log("User logged in", user.profile);
    }
    else {
        log("User not logged in");
    }
});
```

Next, we want to implement the `login`, `api`, and `logout` functions. The `UserManager` provides a `signinRedirect` to log the user in, and a `signoutRedirect` to log the user out. The `User` object that we obtained in the above code also has an `access_token` property which can be used to authenticate to a web API. The `access_token` will be passed to the web API via the *Authorization* header with the *Bearer* scheme. Add this code to implement those three functions in our application:

```
function login() {
    mgr.signinRedirect();
}

function api() {
    mgr.getUser().then(function (user) {
        var url = "http://localhost:5001/identity";

        var xhr = new XMLHttpRequest();
        xhr.open("GET", url);
        xhr.onload = function () {
            log(xhr.status, JSON.parse(xhr.responseText));
        }
        xhr.setRequestHeader("Authorization", "Bearer " + user.access_token);
        xhr.send();
    });
}

function logout() {
    mgr.signoutRedirect();
}
```

Note: See the *client credentials quickstart* for information on how to create the api used in the code above.

callback.html

This HTML file is the designated `redirect_uri` page once the user has logged into IdentityServer. It will complete the OpenID Connect protocol sign-in handshake with IdentityServer. The code for this is all provided by the `UserManager` class we used earlier. Once the sign-in is complete, we can then redirect the user back to the main *index.html* page. Add this code to complete the signin process:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
```

(continues on next page)

(continued from previous page)

```

<script src="oidc-client.js"></script>
<script>
    new Oidc.UserManager({response_mode:"query"}).signinRedirectCallback().
    then(function() {
        window.location = "index.html";
    }).catch(function(e) {
        console.error(e);
    });
</script>
</body>
</html>

```

12.6 Add a client registration to IdentityServer for the JavaScript client

Now that the client application is ready to go, we need to define a configuration entry in IdentityServer for this new JavaScript client. In the IdentityServer project locate the client configuration (in *Config.cs*). Add a new *Client* to the list for our new JavaScript application. It should have the configuration listed below:

```

// JavaScript Client
new Client
{
    ClientId = "js",
    ClientName = "JavaScript Client",
    AllowedGrantTypes = GrantTypes.Code,
    RequirePkce = true,
    RequireClientSecret = false,

    RedirectUris = { "http://localhost:5003/callback.html" },
    PostLogoutRedirectUris = { "http://localhost:5003/index.html" },
    AllowedCorsOrigins = { "http://localhost:5003" },

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        "api1"
    }
}

```

12.7 Allowing Ajax calls to the Web API with CORS

One last bit of configuration that is necessary is to configure CORS in the web API project. This will allow Ajax calls to be made from *http://localhost:5003* to *http://localhost:5001*.

Configure CORS

Add the CORS services to the dependency injection system in *ConfigureServices* in *Startup.cs*:

```

public void ConfigureServices(IServiceCollection services)
{

```

(continues on next page)

(continued from previous page)

```
services.AddControllers();

services.AddAuthentication("Bearer")
    .AddIdentityServerAuthentication(options =>
    {
        options.Authority = "http://localhost:5000";
        options.RequireHttpsMetadata = false;

        options.ApiName = "api1";
    });

services.AddCors(options =>
{
    // this defines a CORS policy called "default"
    options.AddPolicy("default", policy =>
    {
        policy.WithOrigins("http://localhost:5003")
            .AllowAnyHeader()
            .AllowAnyMethod();
    });
});
}
```

Add the CORS middleware to the pipeline in Configure:

```
public void Configure(IApplicationBuilder app)
{
    app.UseRouting();

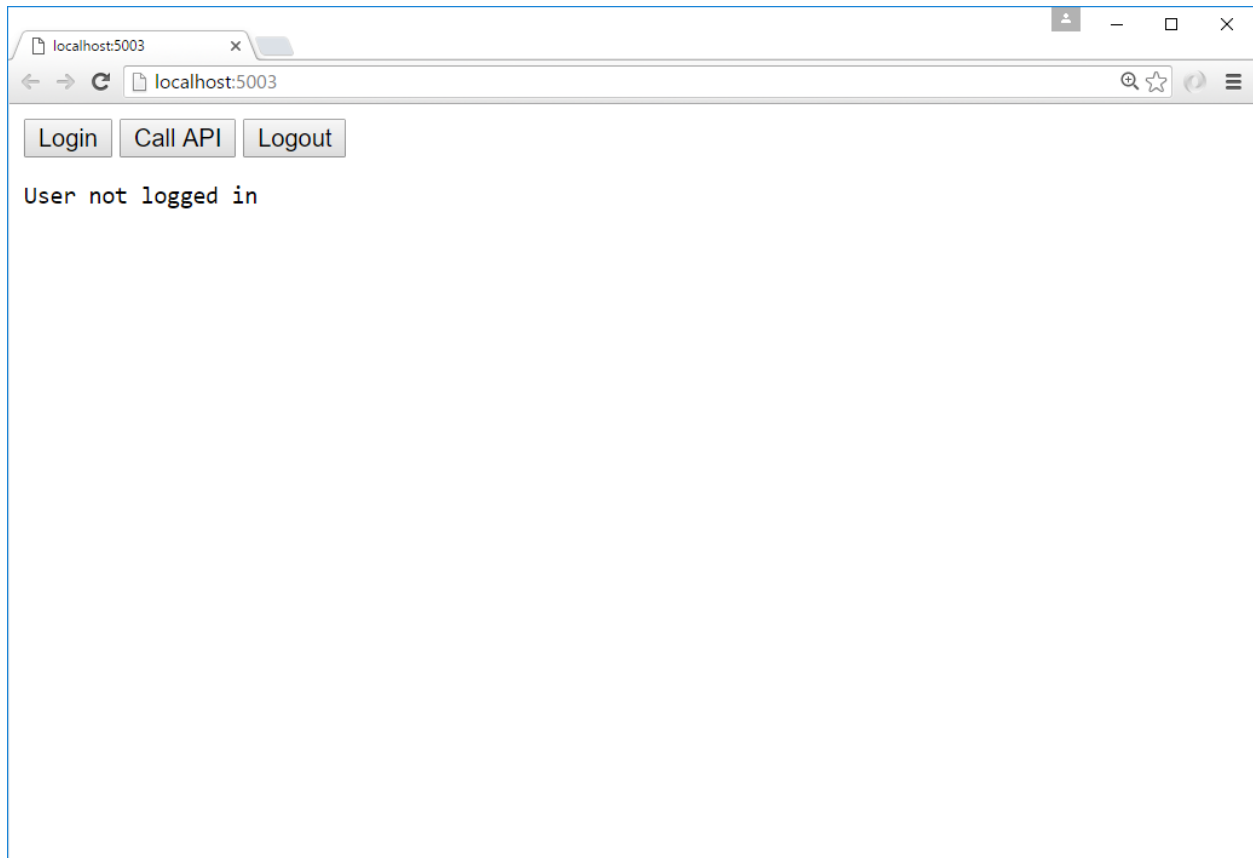
    app.UseCors("default");

    app.UseAuthentication();
    app.UseAuthorization();

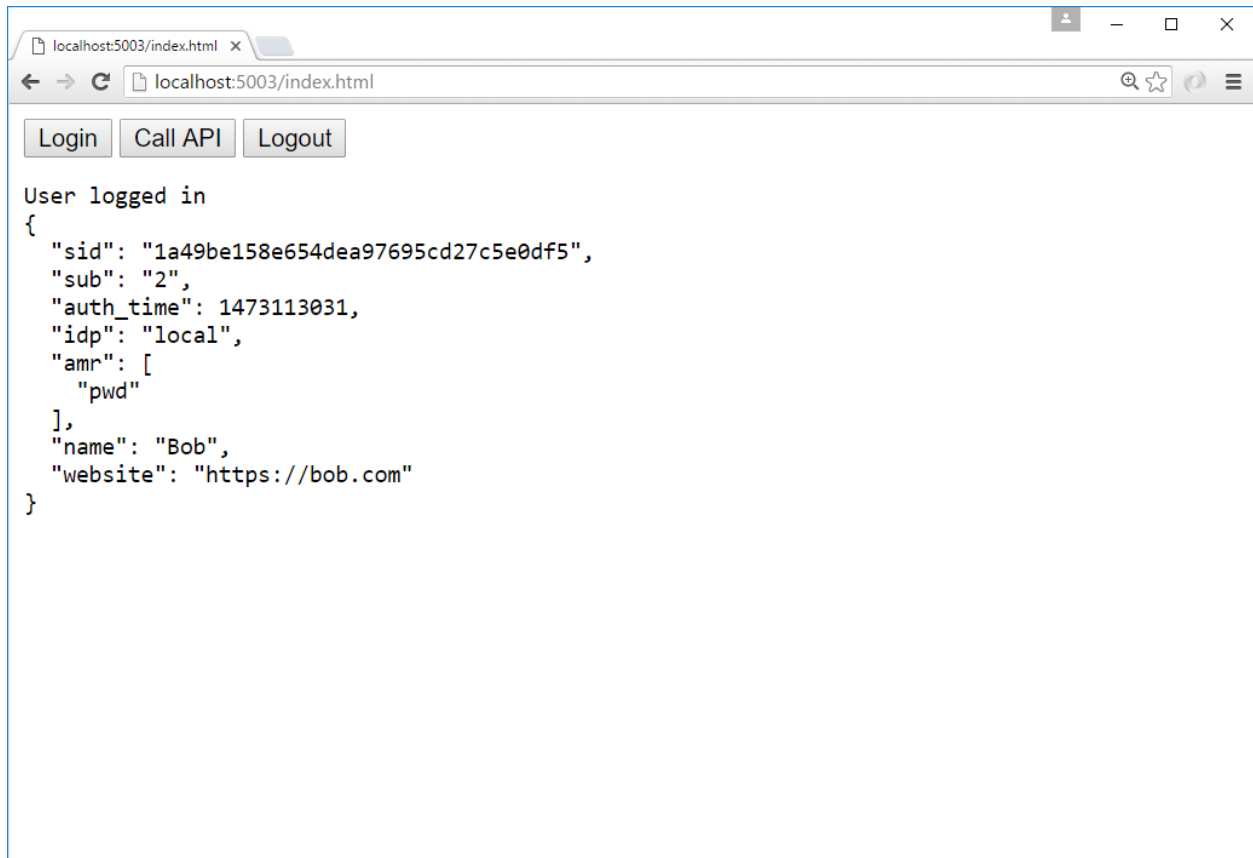
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

12.8 Run the JavaScript application

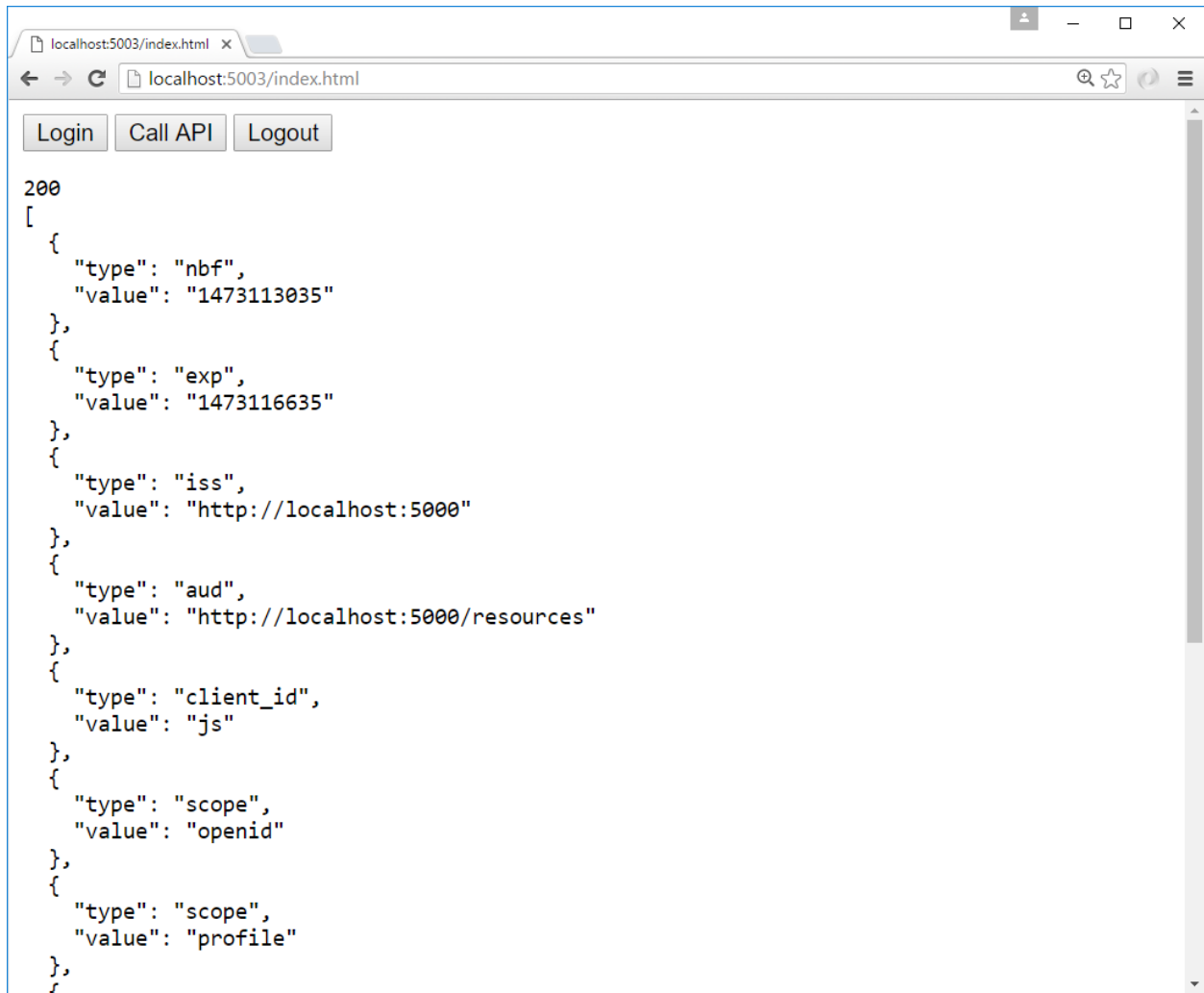
Now you should be able to run the JavaScript client application:



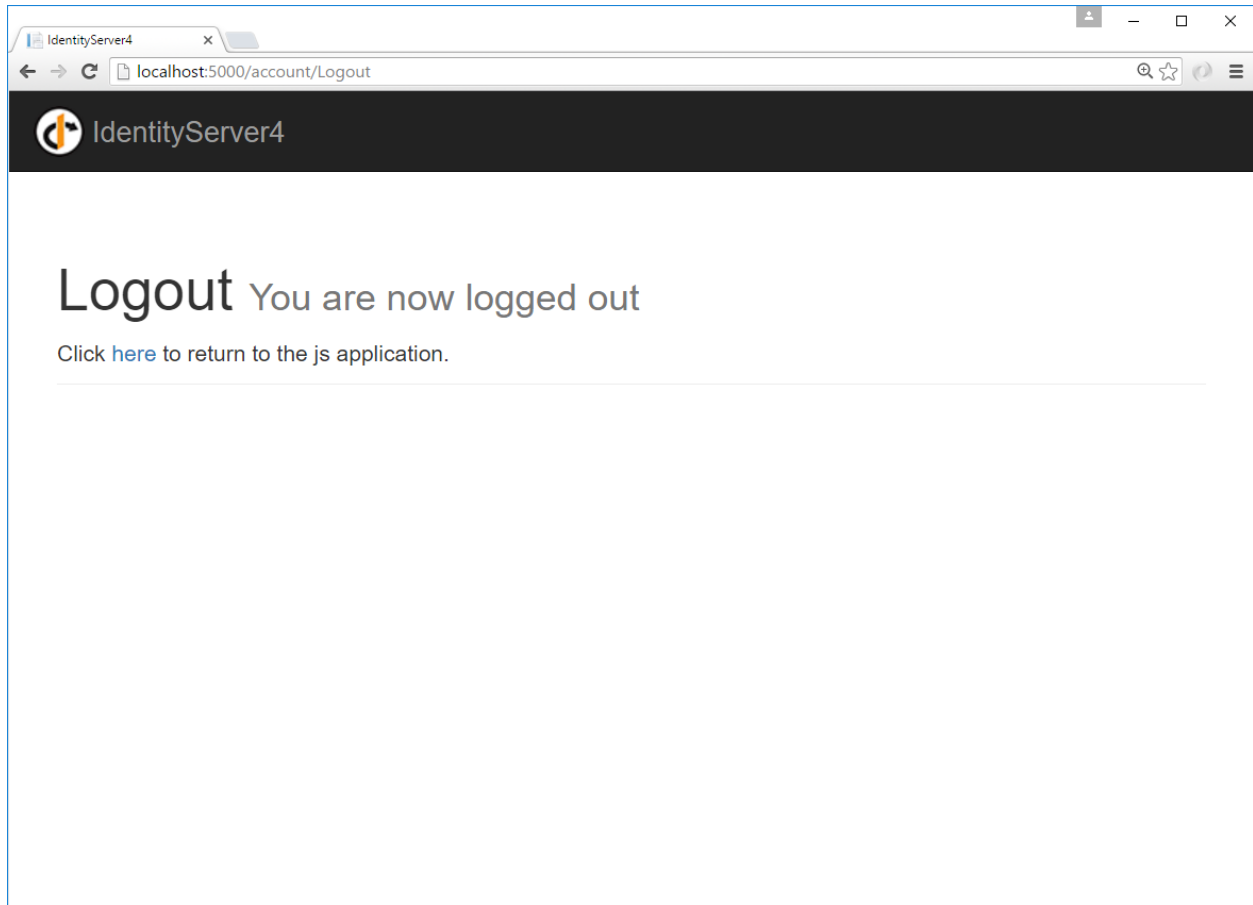
Click the “Login” button to sign the user in. Once the user is returned back to the JavaScript application, you should see their profile information:



And click the “API” button to invoke the web API:



And finally click “Logout” to sign the user out.



You now have the start of a JavaScript client application that uses IdentityServer for sign-in, sign-out, and authenticating calls to web APIs.

Using EntityFramework Core for configuration and operational data

In the previous quickstarts, we created our client and scope data in code. On startup, IdentityServer loaded this configuration data into memory. If we wanted to modify this configuration data, we had to stop and start IdentityServer.

IdentityServer also generates temporary data, such as authorization codes, consent choices, and refresh tokens. By default, these are also stored in-memory.

To move this data into a database that is persistent between restarts and across multiple IdentityServer instances, we can use the IdentityServer4 Entity Framework library.

Note: In addition to manually configuring EF support, there is also an IdentityServer template to create a new project with EF support, using `dotnet new is4ef`.

13.1 IdentityServer4.EntityFramework

IdentityServer4.EntityFramework implements the required stores and services using the following DbContext's:

- ConfigurationDbContext - used for configuration data such as clients, resources, and scopes
- PersistedGrantDbContext - used for temporary operational data such as authorization codes, and refresh tokens

These contexts are suitable for any Entity Framework Core compatible relational database.

You can find these contexts, their entities, and the IdentityServer4 stores that use them in the `IdentityServer4.EntityFramework.Storage` nuget package.

You can find the extension methods to register them in your IdentityServer in `IdentityServer4.EntityFramework`, which we will do now:

```
dotnet add package IdentityServer4.EntityFramework
```

13.2 Using SqlServer

For this quickstart, we will use the LocalDb version of SQLServer that comes with Visual Studio. To add SQL Server support to our IdentityServer project, you'll need the following nuget package:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

13.3 Database Schema Changes and Using EF Migrations

The `IdentityServer4.EntityFramework.Storage` package contains entity classes that map from IdentityServer's models. As IdentityServer's models change, so will the entity classes in `IdentityServer4.EntityFramework.Storage`. As you use `IdentityServer4.EntityFramework.Storage` and upgrade over time, you are responsible for your database schema and changes necessary to that schema as the entity classes change. One approach for managing those changes is to use [EF migrations](#), which is what we'll use in this quickstart. If migrations are not your preference, then you can manage the schema changes in any way you see fit.

Note: You can find the [latest SQL scripts](#) for SqlServer in the `IdentityServer4.EntityFramework.Storage` repository.

13.4 Configuring the Stores

To start using these stores, you'll need to replace any existing calls to `AddInMemoryClients`, `AddInMemoryIdentityResources`, `AddInMemoryApiResources`, and `AddInMemoryPersistedGrants` in your `ConfigureServices` method in *Startup.cs* with `AddConfigurationStore` and `AddOperationalStore`.

These methods each require a `DbContextOptionsBuilder`, meaning your code will look something like this:

```
var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;
const string connectionString = @"Data Source=(LocalDb)\MSSQLLocalDB;
↪database=IdentityServer4.Quickstart.EntityFramework-3.0.0;trusted_connection=yes;";

services.AddIdentityServer()
    .AddTestUsers(TestUsers.Users)
    .AddConfigurationStore(options =>
    {
        options.ConfigureDbContext = b => b.UseSqlServer(connectionString,
            sql => sql.MigrationsAssembly(migrationsAssembly));
    })
    .AddOperationalStore(options =>
    {
        options.ConfigureDbContext = b => b.UseSqlServer(connectionString,
            sql => sql.MigrationsAssembly(migrationsAssembly));
    });
```

You might need these namespaces added to the file:

```
using Microsoft.EntityFrameworkCore;
using System.Reflection;
```

Because we are using EF migrations in this quickstart, the call to `MigrationsAssembly` is used to inform Entity Framework that the host project will contain the migrations code. This is necessary since the host project is in a different assembly than the one that contains the `DbContext` classes.

13.5 Adding Migrations

Once the IdentityServer has been configured to use Entity Framework, we'll need to generate some migrations.

To create migrations, you will need to install the Entity Framework Core CLI on your machine and the `Microsoft.EntityFrameworkCore.Design` nuget package in IdentityServer:

```
dotnet tool install --global dotnet-ef
dotnet add package Microsoft.EntityFrameworkCore.Design
```

To create the migrations, open a command prompt in the IdentityServer project directory and run the following two commands:

```
dotnet ef migrations add InitialIdentityServerPersistedGrantDbMigration -c
↳ PersistedGrantDbContext -o Data/Migrations/IdentityServer/PersistedGrantDb
dotnet ef migrations add InitialIdentityServerConfigurationDbMigration -c
↳ ConfigurationDbContext -o Data/Migrations/IdentityServer/ConfigurationDb
```

You should now see a `~/Data/Migrations/IdentityServer` folder in your project containing the code for your newly created migrations.

13.6 Initializing the Database

Now that we have the migrations, we can write code to create the database from the migrations. We can also seed the database with the in-memory configuration data that we already defined in the previous quickstarts.

Note: The approach used in this quickstart is used to make it easy to get IdentityServer up and running. You should devise your own database creation and maintenance strategy that is appropriate for your architecture.

In *Startup.cs* add this method to help initialize the database:

```
private void InitializeDatabase(IApplicationBuilder app)
{
    using (var serviceScope = app.ApplicationServices.GetService<IServiceScopeFactory>
↳ ().CreateScope())
    {
        serviceScope.ServiceProvider.GetRequiredService<PersistedGrantDbContext>().
↳ Database.Migrate();

        var context = serviceScope.ServiceProvider.GetRequiredService
↳ <ConfigurationDbContext>();
        context.Database.Migrate();
        if (!context.Clients.Any())
        {
            foreach (var client in Config.Clients)
            {
                context.Clients.Add(client.ToEntity());
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
        context.SaveChanges();
    }

    if (!context.IdentityResources.Any())
    {
        foreach (var resource in Config.Ids)
        {
            context.IdentityResources.Add(resource.ToEntity());
        }
        context.SaveChanges();
    }

    if (!context.ApiResources.Any())
    {
        foreach (var resource in Config.Apis)
        {
            context.ApiResources.Add(resource.ToEntity());
        }
        context.SaveChanges();
    }
}
```

The above code may require you to add the following namespaces to your file:

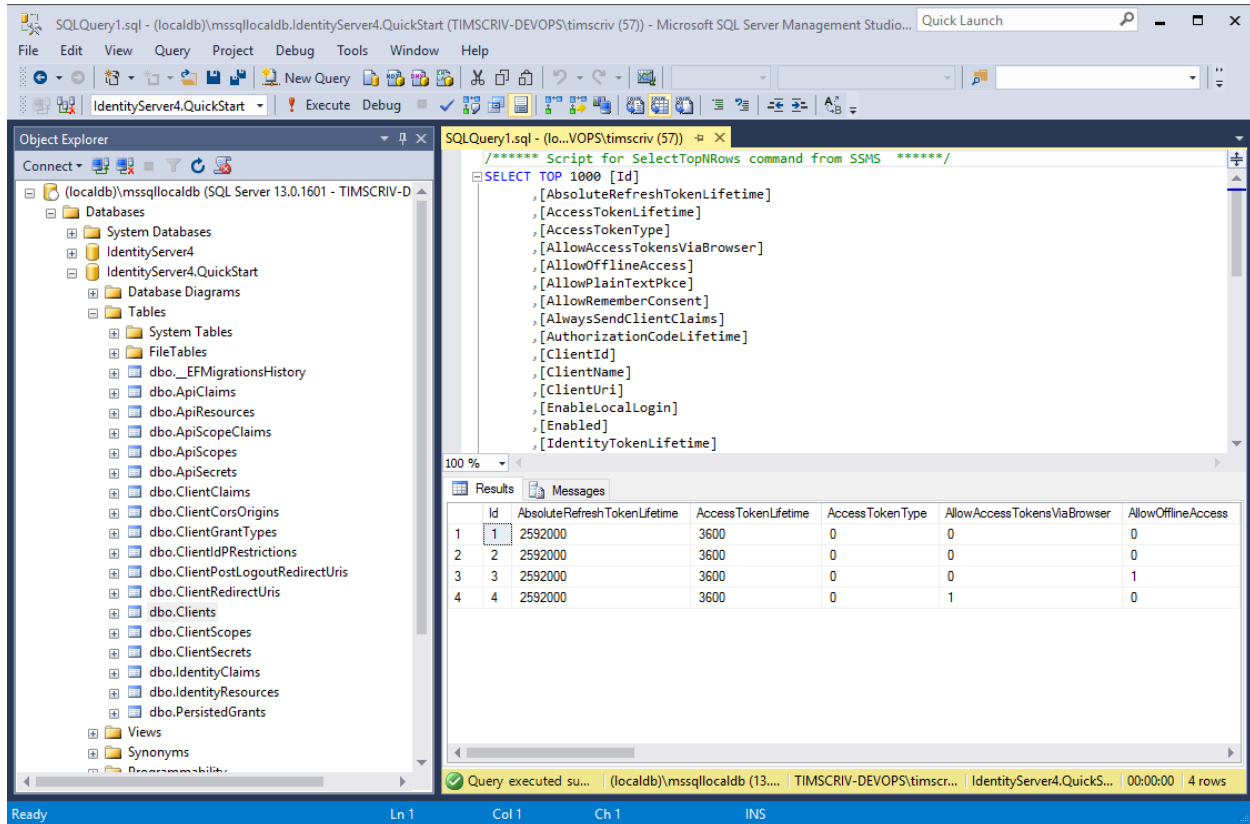
```
using System.Linq;
using IdentityServer4.EntityFramework.DbContexts;
using IdentityServer4.EntityFramework.Mappers;
```

And then we can invoke this from the Configure method:

```
public void Configure(IApplicationBuilder app)
{
    // this will do the initial DB population
    InitializeDatabase(app);

    // the rest of the code that was already here
    // ...
}
```

Now if you run the IdentityServer project, the database should be created and seeded with the quickstart configuration data. You should be able to use SQL Server Management Studio or Visual Studio to connect and inspect the data.



Note: The above `InitializeDatabase` helper API is convenient to seed the database, but this approach is not ideal to leave in to execute each time the application runs. Once your database is populated, consider removing the call to the API.

13.7 Run the client applications

You should now be able to run any of the existing client applications and sign-in, get tokens, and call the API – all based upon the database configuration.

CHAPTER 14

Using ASP.NET Core Identity

Note: For any pre-requisites (like e.g. templates) have a look at the *overview* first.

IdentityServer is designed for flexibility and part of that is allowing you to use any database you want for your users and their data (including passwords). If you are starting with a new user database, then ASP.NET Identity is one option you could choose. This quickstart shows how to use ASP.NET Identity with IdentityServer.

The approach this quickstart takes to using ASP.NET Identity is to create a new project for the IdentityServer host. This new project will replace the prior IdentityServer project we built up in the previous quickstarts. The reason for this new project is due to the differences in UI assets when using ASP.NET Identity (mainly around the differences in login and logout). All the other projects in this solution (for the clients and the API) will remain the same.

Note: This quickstart assumes you are familiar with how ASP.NET Identity works. If you are not, it is recommended that you first [learn about it](#).

14.1 New Project for ASP.NET Identity

The first step is to add a new project for ASP.NET Identity to your solution. We provide a template that contains the minimal UI assets needed to ASP.NET Identity with IdentityServer. You will eventually delete the old project for IdentityServer, but there are some items that you will need to migrate over.

Start by creating a new IdentityServer project that will use ASP.NET Identity:

```
cd quickstart/src
dotnet new is4aspid -n IdentityServerAspNetIdentity
```

When prompted to “seed” the user database, choose “Y” for “yes”. This populates the user database with our “alice” and “bob” users. Their passwords are “Pass123\$”.

Note: The template uses Sqlite as the database for the users, and EF migrations are pre-created in the template. If you wish to use a different database provider, you will need to change the provider used in the code and re-create the EF migrations.

14.2 Inspect the new project

Open the new project in the editor of your choice, and inspect the generated code. Be sure to look at:

14.2.1 IdentityServerAspNetIdentity.csproj

Notice the reference to *IdentityServer4.AspNetIdentity*. This NuGet package contains the ASP.NET Identity integration components for IdentityServer.

14.2.2 Startup.cs

In *ConfigureServices* notice the necessary `AddDbContext<ApplicationDbContext>` and `AddIdentity<ApplicationUser, IdentityRole>` calls are done to configure ASP.NET Identity.

Also notice that much of the same IdentityServer configuration you did in the previous quickstarts is already done. The template uses the in-memory style for clients and resources, and those are sourced from *Config.cs*.

Finally, notice the addition of the new call to `AddAspNetIdentity<ApplicationUser>`. `AddAspNetIdentity` adds the integration layer to allow IdentityServer to access the user data for the ASP.NET Identity user database. This is needed when IdentityServer must add claims for the users into tokens.

Note that `AddIdentity<ApplicationUser, IdentityRole>` must be invoked before `AddIdentityServer`.

14.2.3 Config.cs

Config.cs contains the hard-coded in-memory clients and resource definitions. To keep the same clients and API working as the prior quickstarts, we need to copy over the configuration data from the old IdentityServer project into this one. Do that now, and afterwards *Config.cs* should look like this:

```
public static class Config
{
    public static IEnumerable<IdentityResource> GetIdentityResources()
    {
        return new List<IdentityResource>
        {
            new IdentityResources.OpenId(),
            new IdentityResources.Profile(),
        };
    }

    public static IEnumerable<ApiResource> GetApis()
    {
        return new List<ApiResource>
        {
            new ApiResource("api1", "My API")
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    };
}

public static IEnumerable<Client> GetClients()
{
    return new List<Client>
    {
        new Client
        {
            ClientId = "client",

            // no interactive user, use the clientid/secret for authentication
            AllowedGrantTypes = GrantTypes.ClientCredentials,

            // secret for authentication
            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },

            // scopes that client has access to
            AllowedScopes = { "api1" }
        },
        // resource owner password grant client
        new Client
        {
            ClientId = "ro.client",
            AllowedGrantTypes = GrantTypes.ResourceOwnerPassword,

            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },
            AllowedScopes = { "api1" }
        },
        // OpenID Connect hybrid flow client (MVC)
        new Client
        {
            ClientId = "mvc",
            ClientName = "MVC Client",
            AllowedGrantTypes = GrantTypes.Hybrid,

            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },

            RedirectUris = { "http://localhost:5002/signin-oidc" },
            PostLogoutRedirectUris = { "http://localhost:5002/signout-callback-
oidc" },

            AllowedScopes =
            {
                IdentityServerConstants.StandardScopes.OpenId,
                IdentityServerConstants.StandardScopes.Profile,
                "api1"
            },
        },
    },
};

```

(continues on next page)

(continued from previous page)

```
        AllowOfflineAccess = true
    },
    // JavaScript Client
    new Client
    {
        ClientId = "js",
        ClientName = "JavaScript Client",
        AllowedGrantTypes = GrantTypes.Code,
        RequirePkce = true,
        RequireClientSecret = false,

        RedirectUris = { "http://localhost:5003/callback.html" },
        PostLogoutRedirectUris = { "http://localhost:5003/index.html" },
        AllowedCorsOrigins = { "http://localhost:5003" },

        AllowedScopes =
        {
            IdentityServerConstants.StandardScopes.OpenId,
            IdentityServerConstants.StandardScopes.Profile,
            "api1"
        }
    }
};
}
```

At this point, you no longer need the old IdentityServer project.

14.2.4 Program.cs and SeedData.cs

Program.cs's *Main* is a little different than most ASP.NET Core projects. Notice how this looks for a command line argument called */seed* which is used as a flag to seed the users in the ASP.NET Identity database.

Look at the *SeedData* class' code to see how the database is created and the first users are created.

14.2.5 AccountController

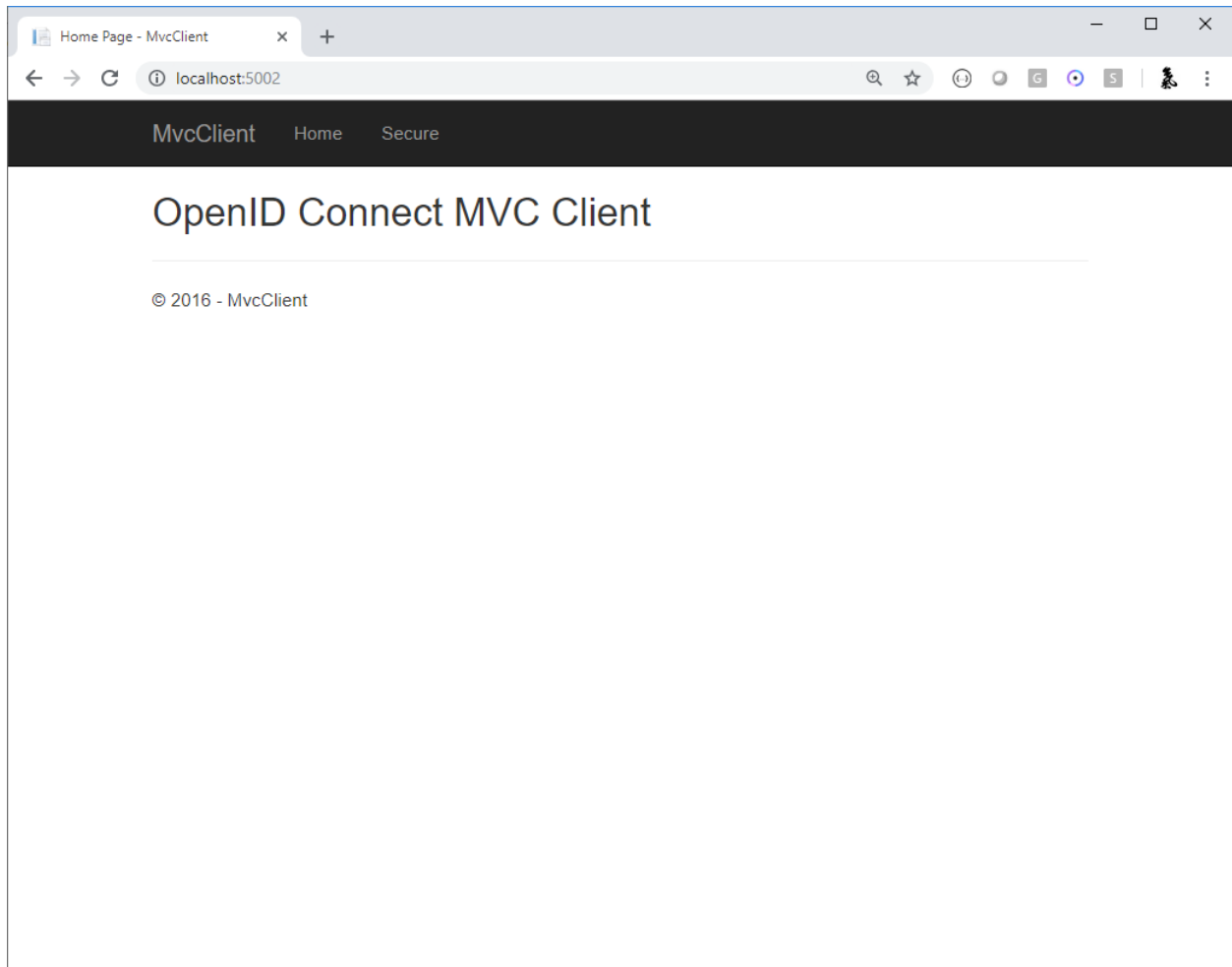
The last code to inspect in this template is the *AccountController*. This contains a slightly different login and logout code than the prior quickstart and templates. Notice the use of the *SignInManager<ApplicationUser>* and *userManager<ApplicationUser>* from ASP.NET Identity to validate credentials and manage the authentication session.

Much of the rest of the code is the same from the prior quickstarts and templates.

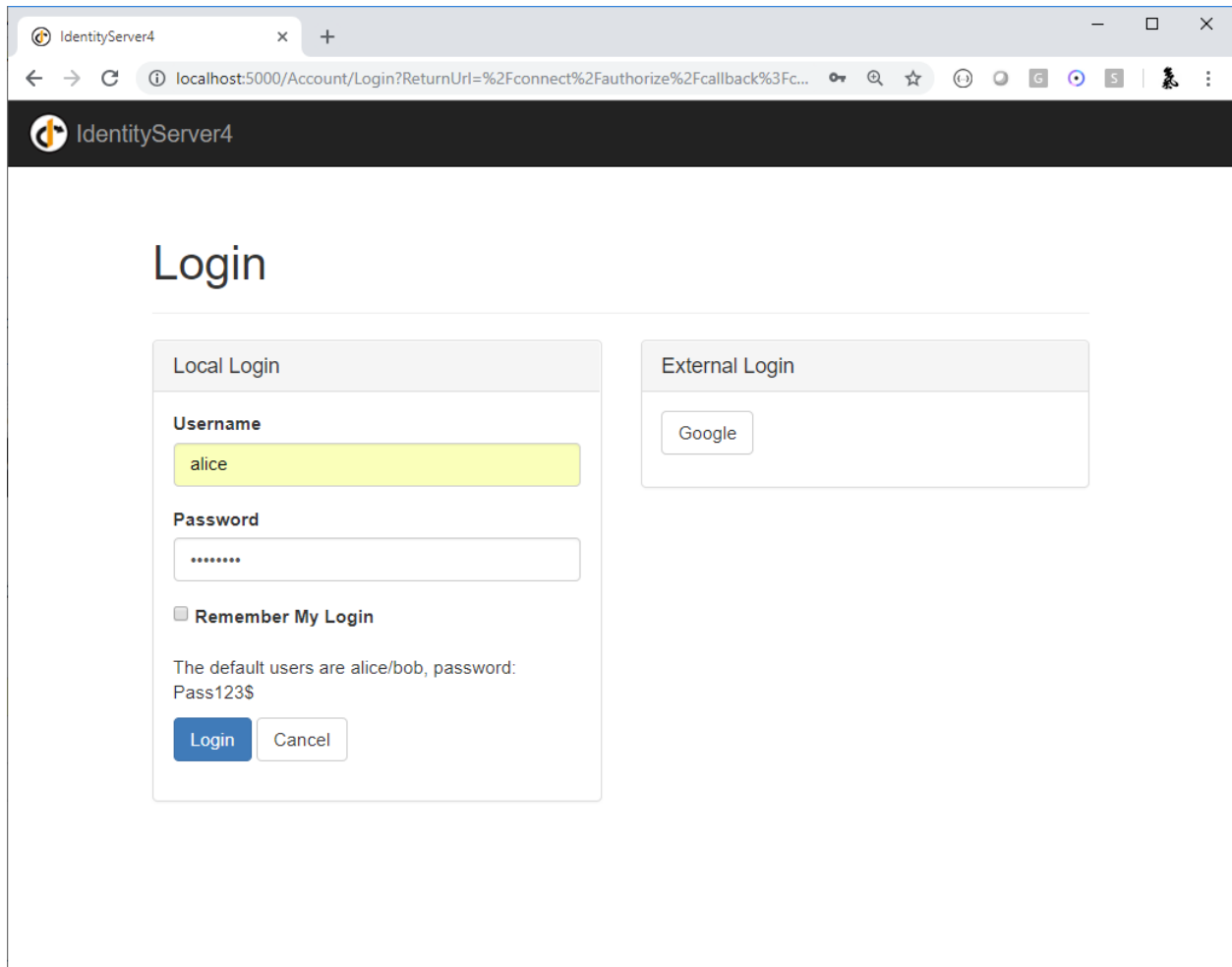
14.3 Logging in with the MVC client

At this point, you should be able to run all of the existing clients and samples. One exception is the *ResourceOwnerClient* – the password will need to be updated to *Pass123\$* from *password*.

Launch the MVC client application, and you should be able to click the “Secure” link to get logged in.



You should be redirected to the ASP.NET Identity login page. Login with your newly created user:



The screenshot shows a web browser window with the URL `localhost:5000/Account/Login?ReturnUrl=%2Fconnect%2Fauthorize%2Fcallback%3Fc...`. The page title is "IdentityServer4". The main heading is "Login". There are two login sections: "Local Login" and "External Login".

Local Login

Username

alice

Password

.....

☐ Remember My Login

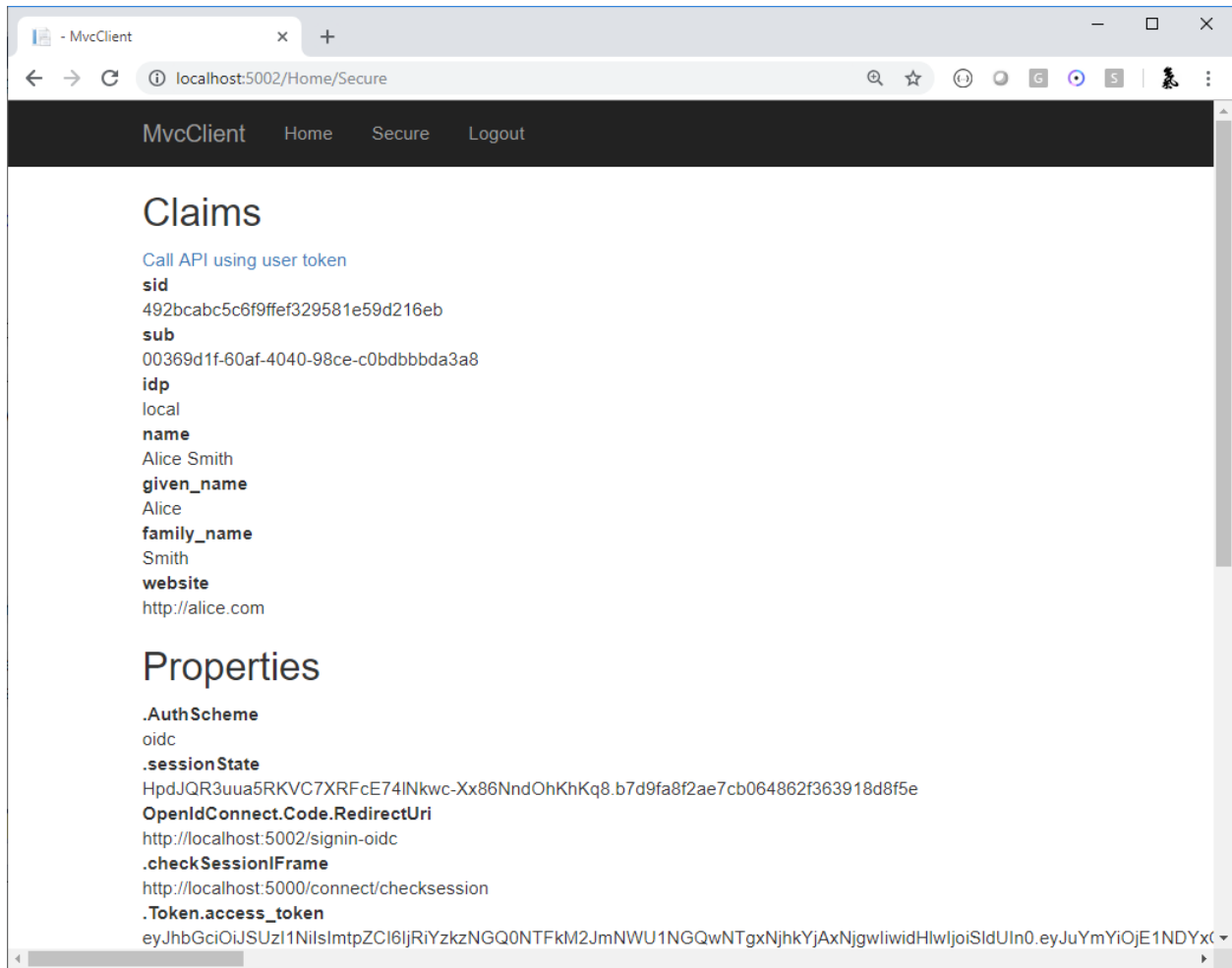
The default users are alice/bob, password:
Pass123\$

Login Cancel

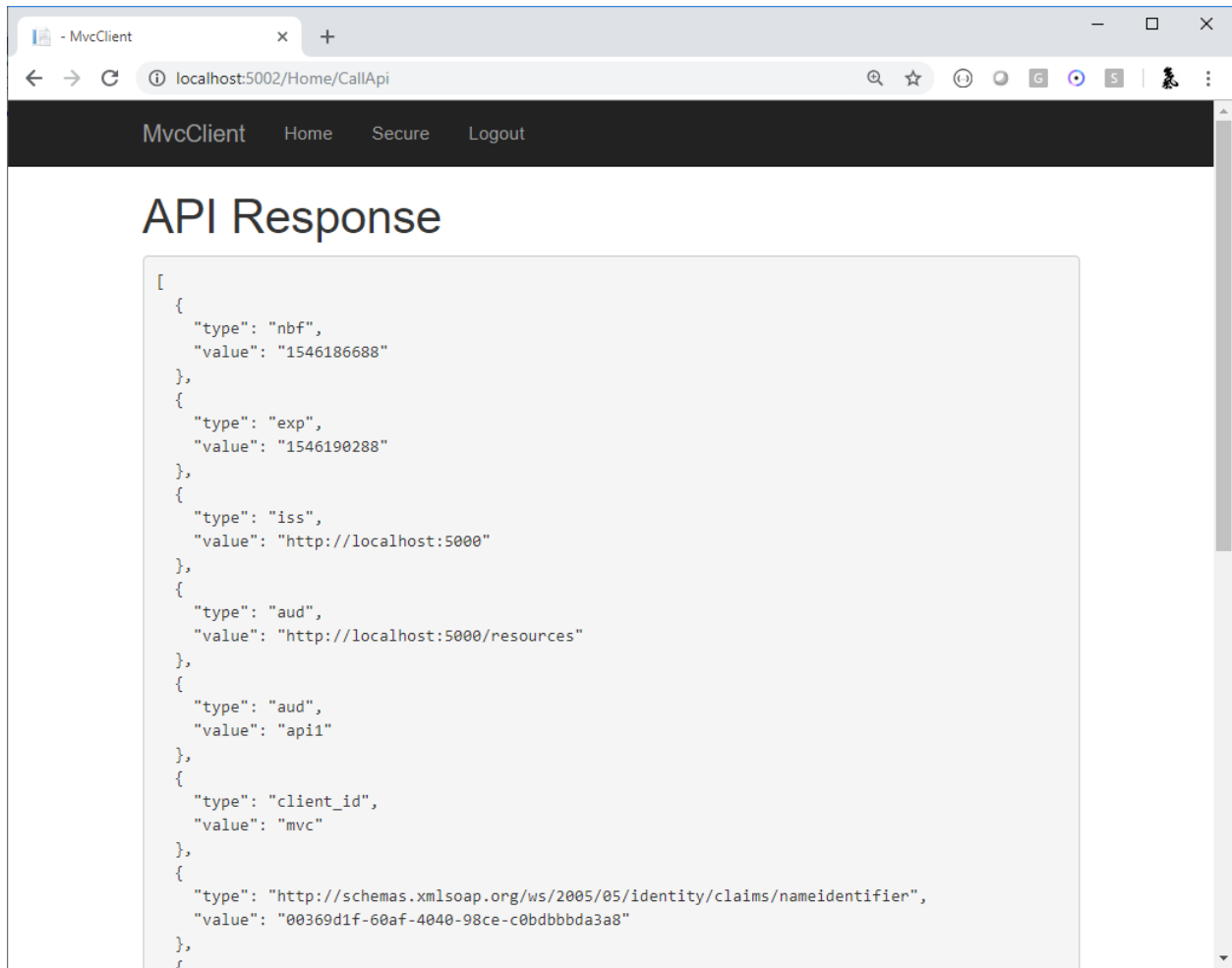
External Login

Google

After login you see the normal consent page. After consent you will be redirected back to the MVC client application where your user's claims should be listed.



You should also be able to click “Call API using application identity” to invoke the API on behalf of the user:



And now you're using users from ASP.NET Identity in IdentityServer.

14.4 What's Missing?

Much of the rest of the code in this template is similar to the other quickstart and templates we provide. The one thing you will notice that is missing from this template is UI code for user registration, password reset, and the other things you might expect from the Visual Studio ASP.NET Identity template.

Given the variety of requirements and different approaches to using ASP.NET Identity, our template deliberately does not provide those features. You are expected to know how ASP.NET Identity works sufficiently well to add those features to your project. Alternatively, you can create a new project based on the Visual Studio ASP.NET Identity template and add the IdentityServer features you have learned about in these quickstarts to that project.

IdentityServer is a combination of middleware and services. All configuration is done in your startup class.

15.1 Configuring services

You add the IdentityServer services to the DI system by calling:

```
public void ConfigureServices(IServiceCollection services)
{
    var builder = services.AddIdentityServer();
}
```

Optionally you can pass in options into this call. See [here](#) for details on options.

This will return you a builder object that in turn has a number of convenience methods to wire up additional services.

15.2 Key material

IdentityServer supports X.509 certificates (both raw files and a reference to the Windows certificate store), RSA keys and EC keys for token signatures and validation. Each key can be configured with a (compatible) signing algorithm, e.g. RS256, RS384, RS512, PS256, PS384, PS512, ES256, ES384 or ES512.

You can configure the key material with the following methods:

- **AddSigningCredential** Adds a signing key that provides the specified key material to the various token creation/validation services.
- **AddDeveloperSigningCredential** Creates temporary key material at startup time. This is for dev scenarios. The generated key will be persisted in the local directory by default.
- **AddValidationKey** Adds a key for validating tokens. They will be used by the internal token validator and will show up in the discovery document.

15.3 In-Memory configuration stores

The various “in-memory” configuration APIs allow for configuring IdentityServer from an in-memory list of configuration objects. These “in-memory” collections can be hard-coded in the hosting application, or could be loaded dynamically from a configuration file or a database. By design, though, these collections are only created when the hosting application is starting up.

Use of these configuration APIs are designed for use when prototyping, developing, and/or testing where it is not necessary to dynamically consult database at runtime for the configuration data. This style of configuration might also be appropriate for production scenarios if the configuration rarely changes, or it is not inconvenient to require restarting the application if the value must be changed.

- **AddInMemoryClients** Registers `IClientStore` and `ICorsPolicyService` implementations based on the in-memory collection of `Client` configuration objects.
- **AddInMemoryIdentityResources** Registers `IResourceStore` implementation based on the in-memory collection of `IdentityResource` configuration objects.
- **AddInMemoryApiResources** Registers `IResourceStore` implementation based on the in-memory collection of `ApiResource` configuration objects.

15.4 Test stores

The `TestUser` class models a user, their credentials, and claims in IdentityServer. Use of `TestUser` is similar to the use of the “in-memory” stores in that it is intended for when prototyping, developing, and/or testing. The use of `TestUser` is not recommended in production.

- **AddTestUsers** Registers `TestUserStore` based on a collection of `TestUser` objects. `TestUserStore` is used by the default quickstart UI. Also registers implementations of `IProfileService` and `IResourceOwnerPasswordValidator`.

15.5 Additional services

- **AddExtensionGrantValidator** Adds `IExtensionGrantValidator` implementation for use with extension grants.
- **AddSecretParser** Adds `ISecretParser` implementation for parsing client or API resource credentials.
- **AddSecretValidator** Adds `ISecretValidator` implementation for validating client or API resource credentials against a credential store.
- **AddResourceOwnerValidator** Adds `IResourceOwnerPasswordValidator` implementation for validating user credentials for the resource owner password credentials grant type.
- **AddProfileService** Adds `IProfileService` implementation for connecting to your *custom user profile store*. The `DefaultProfileService` class provides the default implementation which relies upon the authentication cookie as the only source of claims for issuing in tokens.
- **AddAuthorizeInteractionResponseGenerator** Adds `IAuthorizeInteractionResponseGenerator` implementation to customize logic at authorization endpoint for when a user must be shown a UI for error, login, consent, or any other custom page. The `AuthorizeInteractionResponseGenerator` class provides a default implementation, so consider deriving from this existing class if you need to augment the existing behavior.

- **AddCustomAuthorizeRequestValidator** Adds `ICustomAuthorizeRequestValidator` implementation to customize request parameter validation at the authorization endpoint.
- **AddCustomTokenRequestValidator** Adds `ICustomTokenRequestValidator` implementation to customize request parameter validation at the token endpoint.
- **AddRedirectUriValidator** Adds `IRedirectUriValidator` implementation to customize redirect URI validation.
- **AddAppAuthRedirectUriValidator** Adds a an “AppAuth” (OAuth 2.0 for Native Apps) compliant redirect URI validator (does strict validation but also allows <http://127.0.0.1> with random port).
- **AddJwtBearerClientAuthentication** Adds support for client authentication using JWT bearer assertions.
- **AddMutualTlsSecretValidators** Adds the X509 secret validators for mutual TLS.

15.6 Caching

Client and resource configuration data is used frequently by IdentityServer. If this data is being loaded from a database or other external store, then it might be expensive to frequently re-load the same data.

- **AddInMemoryCaching** To use any of the caches described below, an implementation of `ICache<T>` must be registered in DI. This API registers a default in-memory implementation of `ICache<T>` that’s based on ASP.NET Core’s `MemoryCache`.
- **AddClientStoreCache** Registers a `IClientStore` decorator implementation which will maintain an in-memory cache of `Client` configuration objects. The cache duration is configurable on the `Caching` configuration options on the `IdentityServerOptions`.
- **AddResourceStoreCache** Registers a `IResourceStore` decorator implementation which will maintain an in-memory cache of `IdentityResource` and `ApiResource` configuration objects. The cache duration is configurable on the `Caching` configuration options on the `IdentityServerOptions`.
- **AddCorsPolicyCache** Registers a `ICorsPolicyService` decorator implementation which will maintain an in-memory cache of the results of the CORS policy service evaluation. The cache duration is configurable on the `Caching` configuration options on the `IdentityServerOptions`.

Further customization of the cache is possible:

The default caching relies upon the `ICache<T>` implementation. If you wish to customize the caching behavior for the specific configuration objects, you can replace this implementation in the dependency injection system.

The default implementation of the `ICache<T>` itself relies upon the `IMemoryCache` interface (and `MemoryCache` implementation) provided by .NET. If you wish to customize the in-memory caching behavior, you can replace the `IMemoryCache` implementation in the dependency injection system.

15.7 Configuring the pipeline

You need to add IdentityServer to the pipeline by calling:

```
public void Configure(IApplicationBuilder app)
{
    app.UseIdentityServer();
}
```

Note: `UseIdentityServer` includes a call to `UseAuthentication`, so it's not necessary to have both.

There is no additional configuration for the middleware.

Be aware that order matters in the pipeline. For example, you will want to add `IdentitySever` before the UI framework that implements the login screen.

CHAPTER 16

Defining Resources

The ultimate job of an OpenID Connect/OAuth token service is to control access to resources.

The two fundamental resource types in `IdentityServer` are:

- **identity resources:** represent claims about a user like user ID, display name, email address etc. . .
- **API resources:** represent functionality a client wants to access. Typically, they are HTTP-based endpoints (aka APIs), but could be also message queuing endpoints or similar.

Note: You can define resources using a C# object model - or load them from a data store. An implementation of `IResourceStore` deals with these low-level details. For this document we are using the in-memory implementation.

16.1 Identity Resources

An identity resource is a named group of claims that can be requested using the *scope* parameter.

The OpenID Connect specification [suggests](#) a couple of standard scope name to claim type mappings that might be useful to you for inspiration, but you can freely design them yourself.

One of them is actually mandatory, the *openid* scope, which tells the provider to return the *sub* (subject id) claim in the identity token.

This is how you could define the *openid* scope in code:

```
public static IEnumerable<IdentityResource> GetIdentityResources()
{
    return new List<IdentityResource>
    {
        new IdentityResource(
            name: "openid",
            claimTypes: new[] { "sub" },
```

(continues on next page)

(continued from previous page)

```
        displayName: "Your user identifier")
    };
}
```

But since this is one of the standard scopes from the spec you can shorten that to:

```
public static IEnumerable<IdentityResource> GetIdentityResources()
{
    return new List<IdentityResource>
    {
        new IdentityResources.OpenId()
    };
}
```

Note: see the reference section for more information on `IdentityResource`.

The following example shows a custom identity resource called *profile* that represents the display name, email address and website claim:

```
public static IEnumerable<IdentityResource> GetIdentityResources()
{
    return new List<IdentityResource>
    {
        new IdentityResource(
            name: "profile",
            claimTypes: new[] { "name", "email", "website" },
            displayName: "Your profile data")
    };
}
```

Once the resource is defined, you can give access to it to a client via the `AllowedScopes` option (other properties omitted):

```
var client = new Client
{
    ClientId = "client",

    AllowedScopes = { "openid", "profile" }
};
```

The client can then request the resource using the scope parameter (other parameters omitted):

```
https://demo.identityserver.io/connect/authorize?client_id=client&scope=openid profile
```

IdentityServer will then use the scope names to create a list of requested claim types, and present that to your implementation of the *profile service*.

16.2 APIs

Designing your API surface can be a complicated task. IdentityServer provides a couple of primitives to help you with that.

The original OAuth 2.0 specification has the concept of scopes, which is just defined as *the scope of access* that the client requests. Technically speaking, the *scope* parameter is a list of space delimited values - you need to provide the structure and semantics of it.

In more complex systems, often the notion of a *resource* is introduced. This might be e.g. a physical or logical API. In turn each API can potentially have scopes as well. Some scopes might be exclusive to that resource, and some scopes might be shared.

Let's start with simple scopes first, and then we'll have a look how resources can help structure scopes.

16.2.1 Scopes

Let's model something very simple - a system that has three logical operations *read*, *write*, and *delete*.

You can define them using the `ApiScope` class:

```
public static IEnumerable<ApiScope> GetApiScopes()
{
    return new List<ApiScope>
    {
        new ApiScope(name: "read",    displayName: "Read your data."),
        new ApiScope(name: "write",   displayName: "Write your data."),
        new ApiScope(name: "delete",  displayName: "Delete your data.")
    };
}
```

You can then assign the scopes to various clients, e.g.:

```
var webViewer = new Client
{
    ClientId = "web_viewer",

    AllowedScopes = { "openid", "profile", "read" }
};

var mobileApp = new Client
{
    ClientId = "mobile_app",

    AllowedScopes = { "openid", "profile", "read", "write", "delete" }
}
```

16.2.2 Authorization based on Scopes

When a client asks for a scope (and that scope is allowed via configuration and not denied via consent), the value of that scope will be included in the resulting access token as a claim of type *scope* (for both JWTs and introspection), e.g.:

```
{
  "typ": "at+jwt"
}.
{
  "client_id": "mobile_app",
  "sub": "123",
```

(continues on next page)

(continued from previous page)

```
"scope": "read write delete"
}
```

The consumer of the access token can use that data to make sure that the client is actually allowed to invoke the corresponding functionality.

Note: Be aware, that scopes are purely for authorizing clients - not users. IOW - the *write* scope allows the client to invoke the functionality associated with that. Still that client can most probably only write the data the belongs to the current user. This additional user centric authorization is application logic and not covered by OAuth.

You can add more identity information about the user by deriving additional claims from the scope request. The following scope definition tells the configuration system, that when a *write* scope gets granted, the *user_level* claim should be added to the access token:

```
var writeScope = new ApiScope(
    name: "write",
    displayName: "Write your data.",
    userClaims: new[] { "user_level" });
```

This will pass the *user_level* claim as a requested claim type to the profile service, so that the consumer of the access token can use this data as input for authorization decisions or business logic.

Note: When using the scope-only model, no aud (audience) claim will be added to the token, since this concept does not apply. If you need an aud claim, you can enable the `EmitStaticAudience` setting on the options. This will emit an aud claim in the `issuer_name/resources` format. If you need more control of the aud claim, use API resources.

16.2.3 Parameterized Scopes

Sometimes scopes have a certain structure, e.g. a scope name with an additional parameter: *transaction:id* or *read_patient:patientid*.

In this case you would create a scope without the parameter part and assign that name to a client, but in addition provide some logic to parse the structure of the scope at runtime using the `IScopeParser` interface or by deriving from our default implementation, e.g.:

```
public class ParameterizedScopeParser : DefaultScopeParser
{
    public ParameterizedScopeParser(ILogger<DefaultScopeParser> logger) : base(logger)
    {
    }

    public override void ParseScopeValue(ParseScopeContext scopeContext)
    {
        const string transactionScopeName = "transaction";
        const string separator = ":";
        const string transactionScopePrefix = transactionScopeName + separator;

        var scopeValue = scopeContext.RawValue;

        if (scopeValue.StartsWith(transactionScopePrefix))
```

(continues on next page)

(continued from previous page)

```

    {
        // we get in here with a scope like "transaction:something"
        var parts = scopeValue.Split(separator, StringSplitOptions.
↪RemoveEmptyEntries);
        if (parts.Length == 2)
        {
            scopeContext.SetParsedValues(transactionScopeName, parts[1]);
        }
        else
        {
            scopeContext.SetError("transaction scope missing transaction_
↪parameter value");
        }
    }
    else if (scopeValue != transactionScopeName)
    {
        // we get in here with a scope not like "transaction"
        base.ParseScopeValue(scopeContext);
    }
    else
    {
        // we get in here with a scope exactly "transaction", which is to say we
↪'re ignoring it
        // and not including it in the results
        scopeContext.SetIgnore();
    }
}
}

```

You then have access to the parsed value throughout the pipeline, e.g. in the profile service:

```

public class HostProfileService : IProfileService
{
    public override async Task GetProfileDataAsync(ProfileDataRequestContext context)
    {
        var transaction = context.RequestedResources.ParsedScopes.FirstOrDefault(x =>
↪x.ParsedName == "transaction");
        if (transaction?.ParsedParameter != null)
        {
            context.IssuedClaims.Add(new Claim("transaction_id", transaction.
↪ParsedParameter));
        }
    }
}

```

16.2.4 API Resources

When the API surface gets larger, a flat list of scopes like the one used above might not be feasible.

You typically need to introduce some sort of namespacing to organize the scope names, and maybe you also want to group them together and get some higher-level constructs like an *audience* claim in access tokens. You might also have scenarios, where multiple resources should support the same scope names, whereas sometime you explicitly want to isolate a scope to a certain resource.

In IdentityServer, the `ApiResource` class allows some additional organization. Let's use the following scope definition:

```
public static IEnumerable<ApiScope> GetApiScopes()
{
    return new List<ApiScope>
    {
        // invoice API specific scopes
        new ApiScope(name: "invoice.read",    displayName: "Reads your invoices."),
        new ApiScope(name: "invoice.pay",     displayName: "Pays your invoices."),

        // customer API specific scopes
        new ApiScope(name: "customer.read",   displayName: "Reads your customers_
↳information."),
        new ApiScope(name: "customer.contact", displayName: "Allows contacting one of_
↳your customers."),

        // shared scope
        new ApiScope(name: "manage", displayName: "Provides administrative access to_
↳invoice and customer data.")
    };
}
```

With `ApiResource` you can now create two logical APIs and their corresponding scopes:

```
public static readonly IEnumerable<ApiResource> GetApiResources()
{
    return new List<ApiResource>
    {
        new ApiResource("invoices", "Invoice API")
        {
            Scopes = { "invoice.read", "invoice.pay", "manage" }
        },

        new ApiResource("customers", "Customer API")
        {
            Scopes = { "customer.read", "customer.contact", "manage" }
        }
    };
}
```

Using the API resource grouping gives you the following additional features

- support for the JWT *aud* claim. The value(s) of the audience claim will be the name of the API resource(s)
- support for adding common user claims across all contained scopes
- support for introspection by assigning a API secret to the resource
- support for configuring the access token signing algorithm for the resource

Let's have a look at some example access tokens for the above resource configuration.

Client requests `invoice.read` and `invoice.pay`:

```
{
  "typ": "at+jwt"
}.
{
  "client_id": "client",
  "sub": "123",
```

(continues on next page)

(continued from previous page)

```
"aud": "invoice",  
"scope": "invoice.read invoice.pay"  
}
```

Client requests invoice.read and customer.read:

```
{  
  "typ": "at+jwt"  
}.  
{  
  "client_id": "client",  
  "sub": "123",  
  
  "aud": [ "invoice", "customer" ]  
  "scope": "invoice.read customer.read"  
}
```

Client requests manage:

```
{  
  "typ": "at+jwt"  
}.  
{  
  "client_id": "client",  
  "sub": "123",  
  
  "aud": [ "invoice", "customer" ]  
  "scope": "manage"  
}
```


CHAPTER 17

Defining Clients

Clients represent applications that can request tokens from your identityserver.

The details vary, but you typically define the following common settings for a client:

- a unique client ID
- a secret if needed
- the allowed interactions with the token service (called a grant type)
- a network location where identity and/or access token gets sent to (called a redirect URI)
- a list of scopes (aka resources) the client is allowed to access

Note: At runtime, clients are retrieved via an implementation of the `IClientStore`. This allows loading them from arbitrary data sources like config files or databases. For this document we will use the in-memory version of the client store. You can wire up the in-memory store in `ConfigureServices` via the `AddInMemoryClients` extensions method.

17.1 Defining a client for server to server communication

In this scenario no interactive user is present - a service (aka client) wants to communicate with an API (aka scope):

```
public class Clients
{
    public static IEnumerable<Client> Get()
    {
        return new List<Client>
        {
            new Client
            {
                ClientId = "service.client",
```

(continues on next page)

(continued from previous page)

```
        ClientSecrets = { new Secret("secret".Sha256()) },

        AllowedGrantTypes = GrantTypes.ClientCredentials,
        AllowedScopes = { "api1", "api2.read_only" }
    };
}
```

17.2 Defining an interactive application for use authentication and delegated API access

Interactive applications (e.g. web applications or native desktop/mobile) applications use the authorization code flow. This flow gives you the best security because the access tokens are transmitted via back-channel calls only (and gives you access to refresh tokens):

```
var interactiveClient = new Client
{
    ClientId = "interactive",

    AllowedGrantTypes = GrantTypes.Code,
    AllowOfflineAccess = true,
    ClientSecrets = { new Secret("secret".Sha256()) },

    RedirectUri = { "http://localhost:21402/signin-oidc" },
    PostLogoutRedirectUri = { "http://localhost:21402/" },
    FrontChannelLogoutUri = "http://localhost:21402/signout-oidc",

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        IdentityServerConstants.StandardScopes.Email,

        "api1", "api2.read_only"
    },
};
```

Note: see the *grant types* topic for more information on choosing the right grant type for your client.

17.3 Defining clients in appsettings.json

The `AddInMemoryClients` extensions method also supports adding clients from the ASP.NET Core configuration file. This allows you to define static clients directly from the `appsettings.json` file:

```
"IdentityServer": {
  "IssuerUri": "urn:sso.company.com",
  "Clients": [
    {
```

(continues on next page)

(continued from previous page)

```
"Enabled": true,
"ClientId": "local-dev",
"ClientName": "Local Development",
"ClientSecrets": [ { "Value": "<Insert Sha256 hash of the secret encoded as_
↪Base64 string>" } ],
"AllowedGrantTypes": [ "client_credentials" ],
"AllowedScopes": [ "api1" ],
}
]
}
```

Then pass the configuration section to the `AddInMemoryClients` method:

```
AddInMemoryClients(configuration.GetSection("IdentityServer:Clients"))
```


In order for IdentityServer to issue tokens on behalf of a user, that user must sign-in to IdentityServer.

18.1 Cookie authentication

Authentication is tracked with a cookie managed by the `cookie authentication` handler from ASP.NET Core.

IdentityServer registers two cookie handlers (one for the authentication session and one for temporary external cookies). These are used by default and you can get their names from the `IdentityServerConstants` class (`DefaultCookieAuthenticationScheme` and `ExternalCookieAuthenticationScheme`) if you want to reference them manually.

We only expose basic settings for these cookies (expiration and sliding), and you can register your own cookie handlers if you need more control. IdentityServer uses whichever cookie handler matches the `DefaultAuthenticateScheme` as configured on the `AuthenticationOptions` when using `AddAuthentication` from ASP.NET Core.

18.2 Overriding cookie handler configuration

If you wish to use your own cookie authentication handler, then you must configure it yourself. This must be done in `ConfigureServices` after registering IdentityServer in DI (with `AddIdentityServer`). For example:

```
services.AddIdentityServer()  
    .AddInMemoryClients(Clients.Get())  
    .AddInMemoryIdentityResources(Resources.GetIdentityResources())  
    .AddInMemoryApiResources(Resources.GetApiResources())  
    .AddDeveloperSigningCredential()  
    .AddTestUsers(TestUsers.Users);  
  
services.AddAuthentication("MyCookie")  
    .AddCookie("MyCookie", options =>
```

(continues on next page)

(continued from previous page)

```
{  
    options.ExpireTimeSpan = ...;  
});
```

Note: IdentityServer internally calls both `AddAuthentication` and `AddCookie` with a custom scheme (via the constant `IdentityServerConstants.DefaultCookieAuthenticationScheme`), so to override them you must make the same calls after `AddIdentityServer`.

18.3 Login User Interface and Identity Management System

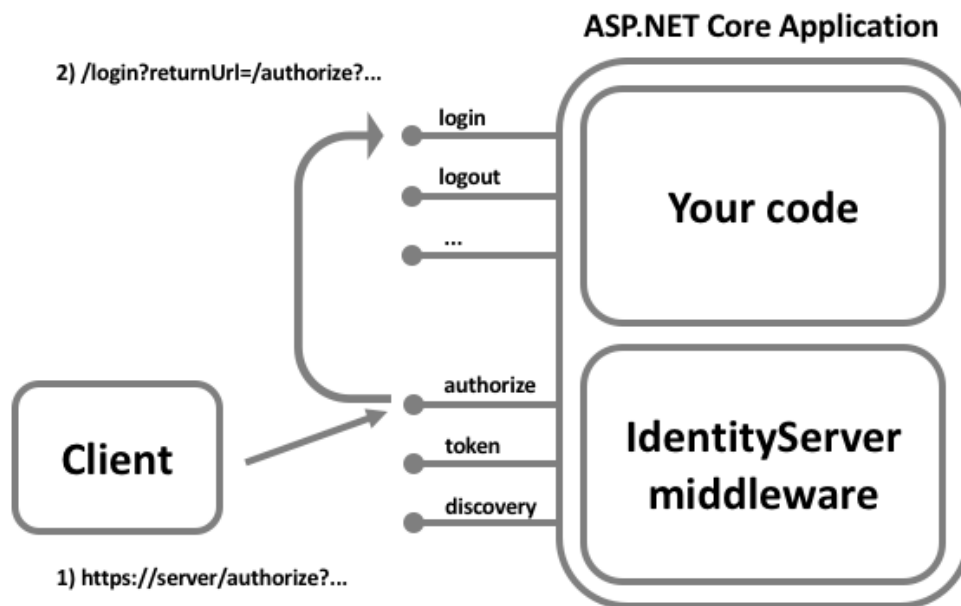
IdentityServer does not provide any user-interface or user database for user authentication. These are things you are expected to provide or develop yourself.

If you need a starting point for a basic UI (login, logout, consent and manage grants), you can use our [quickstart UI](#).

The quickstart UI authenticates users against an in-memory database. You would replace those bits with access to your real user store. We have samples that use *ASP.NET Identity*.

18.4 Login Workflow

When IdentityServer receives a request at the authorization endpoint and the user is not authenticated, the user will be redirected to the configured login page. You must inform IdentityServer of the path to your login page via the `UserInteraction` settings on the `options` (the default is `/account/login`). A `returnUrl` parameter will be passed informing your login page where the user should be redirected once login is complete.



Note: Beware [open-redirect attacks](#) via the `returnUrl` parameter. You should validate that the `returnUrl` refers to well-known location. See the *interaction service* for APIs to validate the `returnUrl` parameter.

18.5 Login Context

On your login page you might require information about the context of the request in order to customize the login experience (such as client, prompt parameter, IdP hint, or something else). This is made available via the `GetAuthorizationContextAsync` API on the *interaction service*.

18.6 Issuing a cookie and Claims

There are authentication-related extension methods on the `HttpContext` from ASP.NET Core to issue the authentication cookie and sign a user in. The authentication scheme used must match the cookie handler you are using (see above).

When you sign the user in you must issue at least a `sub` claim and a `name` claim. IdentityServer also provides a few `SignInAsync` extension methods on the `HttpContext` to make this more convenient.

You can also optionally issue an `idp` claim (for the identity provider name), an `amr` claim (for the authentication method used), and/or an `auth_time` claim (for the epoch time a user authenticated). If you do not provide these, then IdentityServer will provide default values.

Sign-in with External Identity Providers

ASP.NET Core has a flexible way to deal with external authentication. This involves a couple of steps.

Note: If you are using ASP.NET Identity, many of the underlying technical details are hidden from you. It is recommended that you also read the Microsoft [docs](#) and do the ASP.NET Identity *quickstart*.

19.1 Adding authentication handlers for external providers

The protocol implementation that is needed to talk to an external provider is encapsulated in an *authentication handler*. Some providers use proprietary protocols (e.g. social providers like Facebook) and some use standard protocols, e.g. OpenID Connect, WS-Federation or SAML2p.

See this *quickstart* for step-by-step instructions for adding external authentication and configuring it.

19.2 The role of cookies

One option on an external authentication handlers is called `SignInScheme`, e.g.:

```
services.AddAuthentication()
    .AddGoogle("Google", options =>
    {
        options.SignInScheme = "scheme of cookie handler to use";

        options.ClientId = "...";
        options.ClientSecret = "...";
    })
```

The signin scheme specifies the name of the cookie handler that will temporarily store the outcome of the external authentication, e.g. the claims that got sent by the external provider. This is necessary, since there are typically a couple of redirects involved until you are done with the external authentication process.

Given that this is such a common practise, IdentityServer registers a cookie handler specifically for this external provider workflow. The scheme is represented via the `IdentityServerConstants.ExternalCookieAuthenticationScheme` constant. If you were to use our external cookie handler, then for the `SignInScheme` above you'd assign the value to be the `IdentityServerConstants.ExternalCookieAuthenticationScheme` constant:

```
services.AddAuthentication()
    .AddGoogle("Google", options =>
    {
        options.SignInScheme = IdentityServerConstants.
↪ExternalCookieAuthenticationScheme;

        options.ClientId = "...";
        options.ClientSecret = "...";
    })
```

You can also register your own custom cookie handler instead, like this:

```
services.AddAuthentication()
    .AddCookie("YourCustomScheme")
    .AddGoogle("Google", options =>
    {
        options.SignInScheme = "YourCustomScheme";

        options.ClientId = "...";
        options.ClientSecret = "...";
    })
```

Note: For specialized scenarios, you can also short-circuit the external cookie mechanism and forward the external user directly to the main cookie handler. This typically involves handling events on the external handler to make sure you do the correct claims transformation from the external identity source.

19.3 Triggering the authentication handler

You invoke an external authentication handler via the `ChallengeAsync` extension method on the `HttpContext` (or using the MVC `ChallengeResult`).

You typically want to pass in some options to the challenge operation, e.g. the path to your callback page and the name of the provider for bookkeeping, e.g.:

```
var callbackUrl = Url.Action("ExternalLoginCallback");

var props = new AuthenticationProperties
{
    RedirectUri = callbackUrl,
    Items =
    {
        { "scheme", provider },
        { "returnUrl", returnUrl }
    }
};

return Challenge(provider, props);
```

19.4 Handling the callback and signing in the user

On the callback page your typical tasks are:

- inspect the identity returned by the external provider.
- make a decision how you want to deal with that user. This might be different based on the fact if this is a new user or a returning user.
- new users might need additional steps and UI before they are allowed in.
- probably create a new internal user account that is linked to the external provider.
- store the external claims that you want to keep.
- delete the temporary cookie
- sign-in the user

Inspecting the external identity:

```
// read external identity from the temporary cookie
var result = await HttpContext.AuthenticateAsync(IdentityServerConstants.
    ↪ExternalCookieAuthenticationScheme);
if (result?.Succeeded != true)
{
    throw new Exception("External authentication error");
}

// retrieve claims of the external user
var externalUser = result.Principal;
if (externalUser == null)
{
    throw new Exception("External authentication error");
}

// retrieve claims of the external user
var claims = externalUser.Claims.ToList();

// try to determine the unique id of the external user - the most common claim type
↪for that are the sub claim and the NameIdentifier
// depending on the external provider, some other claim type might be used
var userIdClaim = claims.FirstOrDefault(x => x.Type == JwtClaimTypes.Subject);
if (userIdClaim == null)
{
    userIdClaim = claims.FirstOrDefault(x => x.Type == ClaimTypes.NameIdentifier);
}
if (userIdClaim == null)
{
    throw new Exception("Unknown userid");
}

var externalUserId = userIdClaim.Value;
var externalProvider = userIdClaim.Issuer;

// use externalProvider and externalUserId to find your user, or provision a new user
```

Clean-up and sign-in:

```
// issue authentication cookie for user
await HttpContext.SignInAsync(user.SubjectId, user.Username, provider, props,
    ↪additionalClaims.ToArray());

// delete temporary cookie used during external authentication
await HttpContext.SignOutAsync(IdentityServerConstants.
    ↪ExternalCookieAuthenticationScheme);

// validate return URL and redirect back to authorization endpoint or a local page
if (_interaction.IsValidReturnUrl(returnUrl) || Url.IsLocalUrl(returnUrl))
{
    return Redirect(returnUrl);
}

return Redirect("~/");
```

19.5 State, URL length, and ISecureDataFormat

When redirecting to an external provider for sign-in, frequently state from the client application must be round-tripped. This means that state is captured prior to leaving the client and preserved until the user has returned to the client application. Many protocols, including OpenID Connect, allow passing some sort of state as a parameter as part of the request, and the identity provider will return that state on the response. The OpenID Connect authentication handler provided by ASP.NET Core utilizes this feature of the protocol, and that is how it implements the `returnUrl` feature mentioned above.

The problem with storing state in a request parameter is that the request URL can get too large (over the common limit of 2000 characters). The OpenID Connect authentication handler does provide an extensibility point to store the state in your server, rather than in the request URL. You can implement this yourself by implementing `ISecureDataFormat<AuthenticationProperties>` and configuring it on the `OpenIdConnectOptions`.

Fortunately, IdentityServer provides an implementation of this for you, backed by the `IDistributedCache` implementation registered in the DI container (e.g. the standard `MemoryDistributedCache`). To use the IdentityServer provided secure data format implementation, simply call the `AddOidcStateDataFormatterCache` extension method on the `IServiceCollection` when configuring DI. If no parameters are passed, then all OpenID Connect handlers configured will use the IdentityServer provided secure data format implementation:

```
public void ConfigureServices(IServiceCollection services)
{
    // configures the OpenIdConnect handlers to persist the state parameter into the
    ↪server-side IDistributedCache.
    services.AddOidcStateDataFormatterCache();

    services.AddAuthentication()
        .AddOpenIdConnect("demoidsrv", "IdentityServer", options =>
        {
            // ...
        })
        .AddOpenIdConnect("aad", "Azure AD", options =>
        {
            // ...
        })
        .AddOpenIdConnect("adfs", "ADFS", options =>
        {
            // ...
        })
}
```

(continues on next page)

(continued from previous page)

```
    });  
}
```

If only particular schemes are to be configured, then pass those schemes as parameters:

```
public void ConfigureServices(IServiceCollection services)  
{  
    // configures the OpenIdConnect handlers to persist the state parameter into the_  
    ↪server-side IDistributedCache.  
    services.AddOidcStateDataFormatterCache("aad", "demoidsrv");  
  
    services.AddAuthentication()  
        .AddOpenIdConnect("demoidsrv", "IdentityServer", options =>  
        {  
            // ...  
        })  
        .AddOpenIdConnect("aad", "Azure AD", options =>  
        {  
            // ...  
        })  
        .AddOpenIdConnect("adfs", "ADFS", options =>  
        {  
            // ...  
        });  
}
```

Windows Authentication

There are several ways how you can enable Windows authentication in ASP.NET Core (and thus in IdentityServer).

- On Windows using IIS hosting (both in- and out-of process)
- On Windows using HTTP.SYS hosting
- On any platform using the Negotiate authentication handler (added in ASP.NET Core 3.0)

Note: We only have documentation for IIS hosting. If you want to contribute to the docs, please open a PR. thanks!

20.1 On Windows using IIS hosting

The typical `CreateDefaultBuilder` host setup enables support for IIS-based Windows authentication when hosting in IIS. Make sure that Windows authentication is enabled in `launchSettings.json` or your IIS configuration.

The IIS integration layer will configure a Windows authentication handler into DI that can be invoked via the authentication service. Typically in IdentityServer it is advisable to disable the automatic behavior.

This is done in `ConfigureServices` (details vary depending on in-proc vs out-of-proc hosting):

```
// configures IIS out-of-proc settings (see https://github.com/aspnet/AspNetCore/
↪issues/14882)
services.Configure<IISSettings>(iis =>
{
    iis.AuthenticationDisplayName = "Windows";
    iis.AuthenticationType = "Basic";
    iis.AutomaticAuthentication = false;
});

// ..or configures IIS in-proc settings
services.Configure<IISServerOptions>(iis =>
{
```

(continues on next page)

(continued from previous page)

```
iis.AuthenticationDisplayName = "Windows";
iisAutomaticAuthentication = false;
});
```

You trigger Windows authentication by calling `ChallengeAsync` on the Windows scheme (or if you want to use a constant: `Microsoft.AspNetCore.Server.IISIntegration.IISDefaults.AuthenticationScheme`).

This will send the `Www-Authenticate` header back to the browser which will then re-load the current URL including the Windows identity. You can tell that Windows authentication was successful, when you call `AuthenticateAsync` on the Windows scheme and the principal returned is of type `WindowsPrincipal`.

The principal will have information like user and group SID and the Windows account name. The following snippet shows how to trigger authentication, and if successful convert the information into a standard `ClaimsPrincipal` for the temp-Cookie approach:

```
private async Task<IActionResult> ChallengeWindowsAsync(string returnUrl)
{
    // see if windows auth has already been requested and succeeded
    var result = await HttpContext.AuthenticateAsync("Windows");
    if (result?.Principal is WindowsPrincipal wp)
    {
        // we will issue the external cookie and then redirect the
        // user back to the external callback, in essence, treating windows
        // auth the same as any other external authentication mechanism
        var props = new AuthenticationProperties()
        {
            RedirectUri = Url.Action("Callback"),
            Items =
            {
                { "returnUrl", returnUrl },
                { "scheme", "Windows" },
            }
        };

        var id = new ClaimsIdentity("Windows");

        // the sid is a good sub value
        id.AddClaim(new Claim(JwtClaimTypes.Subject, wp.FindFirst(ClaimTypes.
        ↪PrimarySid).Value));

        // the account name is the closest we have to a display name
        id.AddClaim(new Claim(JwtClaimTypes.Name, wp.Identity.Name));

        // add the groups as claims -- be careful if the number of groups is too large
        var wi = wp.Identity as WindowsIdentity;

        // translate group SIDs to display names
        var groups = wi.Groups.Translate(typeof(NTAccount));
        var roles = groups.Select(x => new Claim(JwtClaimTypes.Role, x.Value));
        id.AddClaims(roles);

        await HttpContext.SignInAsync(
            IdentityServerConstants.ExternalCookieAuthenticationScheme,
            new ClaimsPrincipal(id),
            props);
    }
}
```

(continues on next page)

(continued from previous page)

```
        return Redirect(props.RedirectUri);
    }
    else
    {
        // trigger windows auth
        // since windows auth don't support the redirect uri,
        // this URL is re-triggered when we call challenge
        return Challenge("Windows");
    }
}
```


Signing out of IdentityServer is as simple as removing the authentication cookie, but for doing a complete federated sign-out, we must consider signing the user out of the client applications (and maybe even up-stream identity providers) as well.

21.1 Removing the authentication cookie

To remove the authentication cookie, simply use the `SignOutAsync` extension method on the `HttpContext`. You will need to pass the scheme used (which is provided by `IdentityServerConstants.DefaultCookieAuthenticationScheme` unless you have changed it):

```
await HttpContext.SignOutAsync(IdentityServerConstants.  
    DefaultCookieAuthenticationScheme);
```

Or you can use the convenience extension method that is provided by IdentityServer:

```
await HttpContext.SignOutAsync();
```

Note: Typically you should prompt the user for signout (meaning require a POST), otherwise an attacker could hotlink to your logout page causing the user to be automatically logged out.

21.2 Notifying clients that the user has signed-out

As part of the signout process you will want to ensure client applications are informed that the user has signed out. IdentityServer supports the [front-channel](#) specification for server-side clients (e.g. MVC), the [back-channel](#) specification for server-side clients (e.g. MVC), and the [session management](#) specification for browser-based JavaScript clients (e.g. SPA, React, Angular, etc.).

Front-channel server-side clients

To signout the user from the server-side client applications via the front-channel spec, the “logged out” page in IdentityServer must render an `<iframe>` to notify the clients that the user has signed out. Clients that wish to be notified must have the `FrontChannelLogoutUri` configuration value set. IdentityServer tracks which clients the user has signed into, and provides an API called `GetLogoutContextAsync` on the `IIdentityServerInteractionService` (*details*). This API returns a `LogoutRequest` object with a `SignOutIFrameUrl` property that your logged out page must render into an `<iframe>`.

Back-channel server-side clients

To signout the user from the server-side client applications via the back-channel spec, the `SignOutIFrameUrl` endpoint in IdentityServer will automatically trigger server-to-server invocation passing a signed sign-out request to the client. This means that even if there are no front-channel clients, the “logged out” page in IdentityServer must still render an `<iframe>` to the `SignOutIFrameUrl` as described above. Clients that wish to be notified must have the `BackChannelLogoutUri` configuration value set.

Browser-based JavaScript clients

Given how the [session management](#) specification is designed, there is nothing special in IdentityServer that you need to do to notify these clients that the user has signed out. The clients, though, must perform monitoring on the `check_session_iframe`, and this is implemented by the [oidc-client JavaScript library](#).

21.3 Sign-out initiated by a client application

If sign-out was initiated by a client application, then the client first redirected the user to the *end session endpoint*. Processing at the end session endpoint might require some temporary state to be maintained (e.g. the client’s post logout redirect uri) across the redirect to the logout page. This state might be of use to the logout page, and the identifier for the state is passed via a `logoutId` parameter to the logout page.

The `GetLogoutContextAsync` API on the *interaction service* can be used to load the state. Of interest on the `LogoutRequest` model context class is the `ShowSignoutPrompt` which indicates if the request for sign-out has been authenticated, and therefore it’s safe to not prompt the user for sign-out.

By default this state is managed as a protected data structure passed via the `logoutId` value. If you wish to use some other persistence between the end session endpoint and the logout page, then you can implement `IMessageStore<LogoutMessage>` and register the implementation in DI.

Sign-out of External Identity Providers

When a user is *signing-out* of IdentityServer, and they have used an *external identity provider* to sign-in then it is likely that they should be redirected to also sign-out of the external provider. Not all external providers support sign-out, as it depends on the protocol and features they support.

To detect that a user must be redirected to an external identity provider for sign-out is typically done by using a `idp` claim issued into the cookie at IdentityServer. The value set into this claim is the `AuthenticationScheme` of the corresponding authentication middleware. At sign-out time this claim is consulted to know if an external sign-out is required.

Redirecting the user to an external identity provider is problematic due to the cleanup and state management already required by the normal sign-out workflow. The only way to then complete the normal sign-out and cleanup process at IdentityServer is to then request from the external identity provider that after its logout that the user be redirected back to IdentityServer. Not all external providers support post-logout redirects, as it depends on the protocol and features they support.

The workflow at sign-out is then to revoke IdentityServer's authentication cookie, and then redirect to the external provider requesting a post-logout redirect. The post-logout redirect should maintain the necessary sign-out state described *here* (i.e. the `logoutId` parameter value). To redirect back to IdentityServer after the external provider sign-out, the `RedirectUri` should be used on the `AuthenticationProperties` when using ASP.NET Core's `SignOutAsync` API, for example:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Logout(LoginInputModel model)
{
    // build a model so the logged out page knows what to display
    var vm = await _account.BuildLoggedOutViewModelAsync(model.LogoutId);

    var user = HttpContext.User;
    if (user?.Identity.IsAuthenticated == true)
    {
        // delete local authentication cookie
        await HttpContext.SignOutAsync();
    }
}
```

(continues on next page)

(continued from previous page)

```
        // raise the logout event
        await _events.RaiseAsync(new UserLogoutSuccessEvent(user.GetSubjectId(), user.
↪GetName()));
    }

    // check if we need to trigger sign-out at an upstream identity provider
    if (vm.TriggerExternalSignout)
    {
        // build a return URL so the upstream provider will redirect back
        // to us after the user has logged out. this allows us to then
        // complete our single sign-out processing.
        string url = Url.Action("Logout", new { logoutId = vm.LogoutId });

        // this triggers a redirect to the external provider for sign-out
        return SignOut(new AuthenticationProperties { RedirectUri = url }, vm.
↪ExternalAuthenticationScheme);
    }

    return View("LoggedOut", vm);
}
```

Once the user is signed-out of the external provider and then redirected back, the normal sign-out processing at IdentityServer should execute which involves processing the `logoutId` and doing all necessary cleanup.

Federated Sign-out

Federated sign-out is the situation where a user has used an external identity provider to log into IdentityServer, and then the user logs out of that external identity provider via a workflow unknown to IdentityServer. When the user signs out, it will be useful for IdentityServer to be notified so that it can sign the user out of IdentityServer and all of the applications that use IdentityServer.

Not all external identity providers support federated sign-out, but those that do will provide a mechanism to notify clients that the user has signed out. This notification usually comes in the form of a request in an `<iframe>` from the external identity provider's "logged out" page. IdentityServer must then notify all of its clients (as discussed *here*), also typically in the form of a request in an `<iframe>` from within the external identity provider's `<iframe>`.

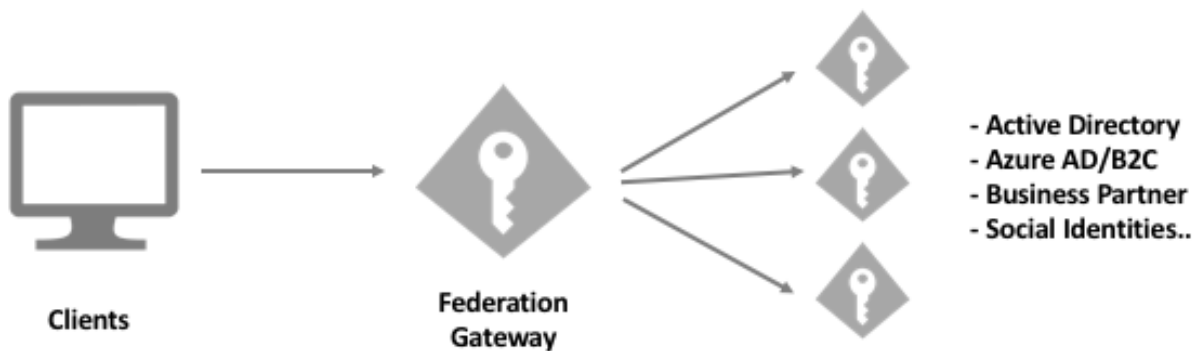
What makes federated sign-out a special case (when compared to a normal *sign-out*) is that the federated sign-out request is not to the normal sign-out endpoint in IdentityServer. In fact, each external IdentityProvider will have a different endpoint into your IdentityServer host. This is due to that fact that each external identity provider might use a different protocol, and each middleware listens on different endpoints.

The net effect of all of these factors is that there is no "logged out" page being rendered as we would on the normal sign-out workflow, which means we are missing the sign-out notifications to IdentityServer's clients. We must add code for each of these federated sign-out endpoints to render the necessary notifications to achieve federated sign-out.

Fortunately IdentityServer already contains this code. When requests come into IdentityServer and invoke the handlers for external authentication providers, IdentityServer detects if these are federated signout requests and if they are it will automatically render the same `<iframe>` as *described here for signout*. In short, federated signout is automatically supported.

Federation Gateway

A common architecture is the so-called federation gateway. In this approach IdentityServer acts as a gateway to one or more external identity providers.



This architecture has the following advantages

- your applications only need to know about the one token service (the gateway) and are shielded from all the details about connecting to the external provider(s). This also means that you can add or change those external providers without needing to update your applications.
- you control the gateway (as opposed to some external service provider) - this means you can make any changes to it and can protect your applications from changes those external providers might do to their own services.
- most external providers only support a fixed set of claims and claim types - having a gateway in the middle allows post-processing the response from the providers to transform/add/amend domain specific identity information.
- some providers don't support access tokens (e.g. social providers) - since the gateway knows about your APIs, it can issue access tokens based on the external identities.
- some providers charge by the number of applications you connect to them. The gateway acts as a single application to the external provider. Internally you can connect as many applications as you want.

- some providers use proprietary protocols or made proprietary modifications to standard protocols - with a gateway there is only one place you need to deal with that.
- forcing every authentication (internal or external) through one single place gives you tremendous flexibility with regards to identity mapping, providing a stable identity to all your applications and dealing with new requirements

In other words - owning your federation gateway gives you a lot of control over your identity infrastructure. And since the identity of your users is one of your most important assets, we recommend taking control over the gateway.

24.1 Implementation

Our [quick start UI](#) utilizes some of the below features. Also check out the *external authentication quickstart* and the docs about *external providers*.

- You can add support for external identity providers by adding authentication handlers to your IdentityServer application.
- You can programmatically query those external providers by calling `IAuthenticationSchemeProvider`. This allows to dynamically render your login page based on the registered external providers.
- Our client configuration model allows restricting the available providers on a per client basis (use the `IdentityProviderRestrictions` property).
- You can also use the `EnableLocalLogin` property on the client to tell your UI whether the username/password input should be rendered.
- Our quickstart UI funnels all external authentication calls through a single callback (see `ExternalLoginCallback` on the `AccountController` class). This allows for a single point for post-processing.

During an authorization request, if IdentityServer requires user consent the browser will be redirected to the consent page.

Consent is used to allow an end user to grant a client access to resources (*identity* or *API*). This is typically only necessary for third-party clients, and can be enabled/disabled per-client on the *client settings*.

25.1 Consent Page

In order for the user to grant consent, a consent page must be provided by the hosting application. The [quickstart UI](#) has a basic implementation of a consent page.

A consent page normally renders the display name of the current user, the display name of the client requesting access, the logo of the client, a link for more information about the client, and the list of resources the client is requesting access to. It's also common to allow the user to indicate that their consent should be “remembered” so they are not prompted again in the future for the same client.

Once the user has provided consent, the consent page must inform IdentityServer of the consent, and then the browser must be redirected back to the authorization endpoint.

25.2 Authorization Context

IdentityServer will pass a *returnUrl* parameter (configurable on the *user interaction options*) to the consent page which contains the parameters of the authorization request. These parameters provide the context for the consent page, and can be read with help from the *interaction service*. The `GetAuthorizationContextAsync` API will return an instance of `AuthorizationRequest`.

Additional details about the client or resources can be obtained using the `IClientStore` and `IResourceStore` interfaces.

25.3 Informing IdentityServer of the consent result

The `GrantConsentAsync` API on the *interaction service* allows the consent page to inform IdentityServer of the outcome of consent (which might also be to deny the client access).

IdentityServer will temporarily persist the outcome of the consent. This persistence uses a cookie by default, as it only needs to last long enough to convey the outcome back to the authorization endpoint. This temporary persistence is different than the persistence used for the “remember my consent” feature (and it is the authorization endpoint which persists the “remember my consent” for the user). If you wish to use some other persistence between the consent page and the authorization redirect, then you can implement `IMessageStore<ConsentResponse>` and register the implementation in DI.

25.4 Returning the user to the authorization endpoint

Once the consent page has informed IdentityServer of the outcome, the user can be redirected back to the *returnUrl*. Your consent page should protect against open redirects by verifying that the *returnUrl* is valid. This can be done by calling `IsValidReturnUrl` on the *interaction service*. Also, if `GetAuthorizationContextAsync` returns a non-null result, then you can also trust that the *returnUrl* is valid.

CHAPTER 26

Protecting APIs

IdentityServer issues access tokens in the **JWT** (JSON Web Token) format by default.

Every relevant platform today has support for validating JWT tokens, a good list of JWT libraries can be found [here](#). Popular libraries are e.g.:

- [JWT bearer authentication handler](#) for ASP.NET Core
- [JWT bearer authentication middleware](#) for Katana
- [IdentityServer authentication middleware](#) for Katana
- [jsonwebtoken](#) for nodejs

Protecting a ASP.NET Core-based API is only a matter of configuring the JWT bearer authentication handler in DI, and adding the authentication middleware to the pipeline:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();

        services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
            .AddJwtBearer(options =>
            {
                // base-address of your identityserver
                options.Authority = "https://demo.identityserver.io";

                // name of the API resource
                options.Audience = "api1";
            });
    }

    public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
    {
        app.UseAuthentication();
    }
}
```

(continues on next page)

(continued from previous page)

```
        app.UseMvc();
    }
}
```

26.1 The IdentityServer authentication handler

Our authentication handler serves the same purpose as the above handler (in fact it uses the Microsoft JWT library internally), but adds a couple of additional features:

- support for both JWTs and reference tokens
- extensible caching for reference tokens
- unified configuration model
- scope validation

For the simplest case, our handler configuration looks very similar to the above snippet:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();

        services.AddAuthentication(IdentityServerAuthenticationDefaults.
↪AuthenticationScheme)
            .AddIdentityServerAuthentication(options =>
            {
                // base-address of your identityserver
                options.Authority = "https://demo.identityserver.io";

                // name of the API resource
                options.ApiName = "api1";
            });
    }

    public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
    {
        app.UseAuthentication();
        app.UseMvc();
    }
}
```

You can get the package from [nuget](#) or [github](#).

26.2 Supporting reference tokens

If the incoming token is not a JWT, our middleware will contact the introspection endpoint found in the discovery document to validate the token. Since the introspection endpoint requires authentication, you need to supply the configured API secret, e.g.:


```
.AddIdentityServerAuthentication(options =>
{
    // base-address of your identityserver
    options.Authority = "https://demo.identityserver.io";

    // name of the API resource
    options.ApiName = "api1";
    options.ApiSecret = "secret";
})
```

Typically, you don't want to do a roundtrip to the introspection endpoint for each incoming request. The middleware has a built-in cache that you can enable like this:

```
.AddIdentityServerAuthentication(options =>
{
    // base-address of your identityserver
    options.Authority = "https://demo.identityserver.io";

    // name of the API resource
    options.ApiName = "api1";
    options.ApiSecret = "secret";

    options.EnableCaching = true;
    options.CacheDuration = TimeSpan.FromMinutes(10); // that's the default
})
```

The handler will use whatever *IDistributedCache* implementation is registered in the DI container (e.g. the standard *MemoryDistributedCache*).

26.3 Validating scopes

The *ApiName* property checks if the token has a matching audience (or short aud) claim.

In IdentityServer you can also sub-divide APIs into multiple scopes. If you need that granularity you can use the ASP.NET Core authorization policy system to check for scopes.

Creating a global policy:

```
services
    .AddMvcCore(options =>
    {
        // require scopel or scope2
        var policy = ScopePolicy.Create("scopel", "scope2");
        options.Filters.Add(new AuthorizeFilter(policy));
    })
    .AddJsonFormatters()
    .AddAuthorization();
```

Composing a scope policy:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("myPolicy", builder =>
    {
        // require scopel
```

(continues on next page)

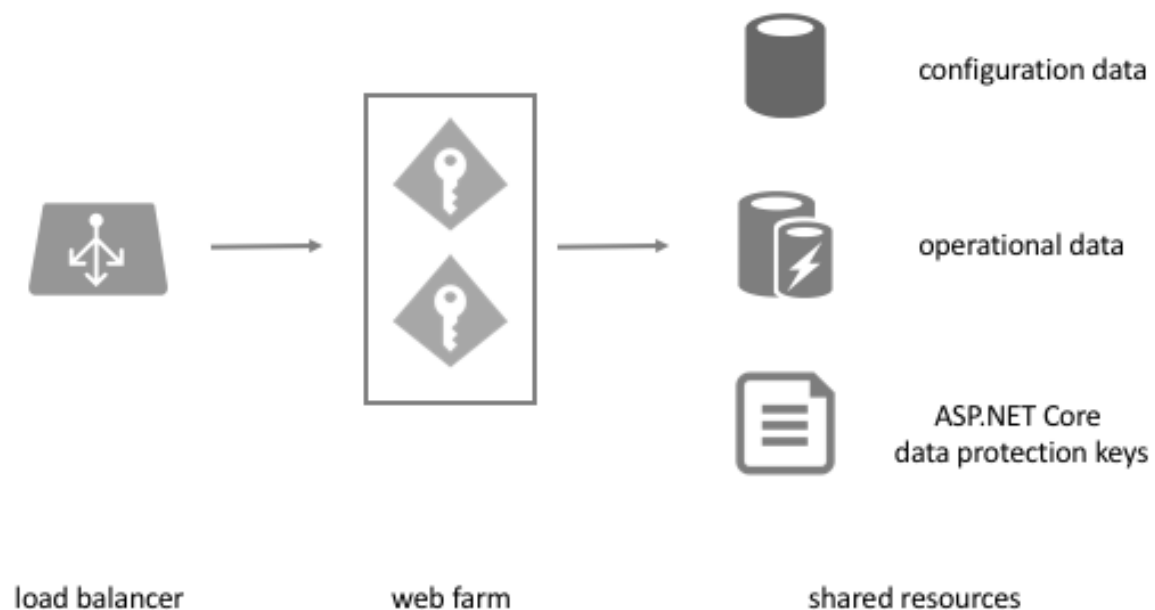
(continued from previous page)

```
builder.RequireScope("scope1");  
// and require scope2 or scope3  
builder.RequireScope("scope2", "scope3");  
});  
});
```

Your identity server is *just* a standard ASP.NET Core application including the IdentityServer middleware. Read the official Microsoft [documentation](#) on publishing and deployment first (and especially the [section](#) about load balancers and proxies).

27.1 Typical architecture

Typically you will design your IdentityServer deployment for high availability:



IdentityServer itself is stateless and does not require server affinity - but there is data that needs to be shared between the instances.

27.2 Configuration data

This typically includes:

- resources
- clients
- startup configuration, e.g. key material, external provider settings etc. . .

The way you store that data depends on your environment. In situations where configuration data rarely changes we recommend using the in-memory stores and code or configuration files.

In highly dynamic environments (e.g. SaaS) we recommend using a database or configuration service to load configuration dynamically.

IdentityServer supports code configuration and configuration files (see [here](#)) out of the box. For databases we provide support for [Entity Framework Core](#) based databases.

You can also build your own configuration stores by implementing `IResourceStore` and `IClientStore`.

27.3 Key material

Another important piece of startup configuration is your key material, see [here](#) for more details on key material and cryptography.

27.4 Operational data

For certain operations, IdentityServer needs a persistence store to keep state, this includes:

- issuing authorization codes
- issuing reference and refresh tokens
- storing consent

You can either use a traditional database for storing operational data, or use a cache with persistence features like Redis. The EF Core implementation mentioned above has also support for operational data.

You can also implement support for your own custom storage mechanism by implementing `IPersistedGrantStore` - by default IdentityServer injects an in-memory version.

27.5 ASP.NET Core data protection

ASP.NET Core itself needs shared key material for protecting sensitive data like cookies, state strings etc. See the official docs [here](#).

You can either re-use one of the above persistence store or use something simple like a shared file if possible.

27.6 ASP.NET Core distributed caching

Some components rely on ASP.NET Core distributed caching. In order to work in a multi server environment, this needs to be set up correctly. The [official docs](#) describe several options.

The following components rely on `IDistributedCache`:

- `services.AddOidcStateDataFormatterCache()` configures the OpenIdConnect handlers to persist the state parameter into the server-side `IDistributedCache`.
- `DefaultReplayCache`
- `DistributedDeviceFlowThrottlingService`
- `DistributedCacheAuthorizationParametersMessageStore`

IdentityServer uses the standard logging facilities provided by ASP.NET Core. The Microsoft [documentation](#) has a good intro and a description of the built-in logging providers.

We are roughly following the Microsoft guidelines for usage of log levels:

- **Trace** For information that is valuable only to a developer troubleshooting an issue. These messages may contain sensitive application data like tokens and should not be enabled in a production environment.
- **Debug** For following the internal flow and understanding why certain decisions are made. Has short-term usefulness during development and debugging.
- **Information** For tracking the general flow of the application. These logs typically have some long-term value.
- **Warning** For abnormal or unexpected events in the application flow. These may include errors or other conditions that do not cause the application to stop, but which may need to be investigated.
- **Error** For errors and exceptions that cannot be handled. Examples: failed validation of a protocol request.
- **Critical** For failures that require immediate attention. Examples: missing store implementation, invalid key material...

28.1 Setup for Serilog

We personally like [Serilog](#) and the `Serilog.AspNetCore` package a lot. Give it a try:

```
public class Program
{
    public static int Main(string[] args)
    {
        Activity.DefaultIdFormat = ActivityIdFormat.W3C;

        Log.Logger = new LoggerConfiguration()
            .MinimumLevel.Debug()
```

(continues on next page)

(continued from previous page)

```

        .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
        .MinimumLevel.Override("Microsoft.Hosting.Lifetime", LogEventLevel.
↪Information)
        .MinimumLevel.Override("System", LogEventLevel.Warning)
        .MinimumLevel.Override("Microsoft.AspNetCore.Authentication", ↪
↪LogEventLevel.Information)
        .Enrich.FromLogContext()
        .WriteTo.Console(outputTemplate: "[{Timestamp:HH:mm:ss} {Level}]
↪{SourceContext}{NewLine}{Message:l}{NewLine}{Exception}{NewLine}", theme: ↪
↪AnsiConsoleTheme.Code)
        .CreateLogger();

    try
    {
        Log.Information("Starting host...");
        CreateHostBuilder(args).Build().Run();
        return 0;
    }
    catch (Exception ex)
    {
        Log.Fatal(ex, "Host terminated unexpectedly.");
        return 1;
    }
    finally
    {
        Log.CloseAndFlush();
    }
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Microsoft.Extensions.Hosting.Host.CreateDefaultBuilder(args)
        .UseSerilog()
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
}

```


While logging is more low level “printf” style - events represent higher level information about certain operations in IdentityServer. Events are structured data and include event IDs, success/failure information, categories and details. This makes it easy to query and analyze them and extract useful information that can be used for further processing.

Events work great with event stores like [ELK](#), [Seq](#) or [Splunk](#).

29.1 Emitting events

Events are not turned on by default - but can be globally configured in the `ConfigureServices` method, e.g.:

```
services.AddIdentityServer(options =>
{
    options.Events.RaiseSuccessEvents = true;
    options.Events.RaiseFailureEvents = true;
    options.Events.RaiseErrorEvents = true;
});
```

To emit an event use the `IService` from the DI container and call the `RaiseAsync` method, e.g.:

```
public async Task<IActionResult> Login(LoginInputModel model)
{
    if (_users.ValidateCredentials(model.Username, model.Password))
    {
        // issue authentication cookie with subject ID and username
        var user = _users.FindByUsername(model.Username);
        await _events.RaiseAsync(new UserLoginSuccessEvent(user.Username, user.
↪SubjectId, user.Username));
    }
    else
    {
        await _events.RaiseAsync(new UserLoginFailureEvent(model.Username, "invalid_
↪redentials"));
    }
}
```

(continues on next page)

(continued from previous page)

```
}  
}
```

29.2 Custom sinks

Our default event sink will simply serialize the event class to JSON and forward it to the ASP.NET Core logging system. If you want to connect to a custom event store, implement the `IEventSink` interface and register it with DI.

The following example uses [Seq](#) to emit events:

```
public class SeqEventSink : IEventSink  
{  
    private readonly Logger _log;  
  
    public SeqEventSink()  
    {  
        _log = new LoggerConfiguration()  
            .WriteTo.Seq("http://localhost:5341")  
            .CreateLogger();  
    }  
  
    public Task PersistAsync(Event evt)  
    {  
        if (evt.EventType == EventTypes.Success ||  
            evt.EventType == EventTypes.Information)  
        {  
            _log.Information("{Name} ({Id}), Details: {@details}",  
                evt.Name,  
                evt.Id,  
                evt);  
        }  
        else  
        {  
            _log.Error("{Name} ({Id}), Details: {@details}",  
                evt.Name,  
                evt.Id,  
                evt);  
        }  
  
        return Task.CompletedTask;  
    }  
}
```

Add the `Serilog.Sinks.Seq` package to your host to make the above code work.

29.3 Built-in events

The following events are defined in IdentityServer:

ApiAuthenticationFailureEvent & ApiAuthenticationSuccessEvent Gets raised for successful/failed API authentication at the introspection endpoint.

ClientAuthenticationSuccessEvent & ClientAuthenticationFailureEvent Gets raised for successful/failed client authentication at the token endpoint.

TokenIssuedSuccessEvent & TokenIssuedFailureEvent Gets raised for successful/failed attempts to request identity tokens, access tokens, refresh tokens and authorization codes.

TokenIntrospectionSuccessEvent & TokenIntrospectionFailureEvent Gets raised for successful token introspection requests.

TokenRevokedSuccessEvent Gets raised for successful token revocation requests.

UserLoginSuccessEvent & UserLoginFailureEvent Gets raised by the quickstart UI for successful/failed user logins.

UserLogoutSuccessEvent Gets raised for successful logout requests.

ConsentGrantedEvent & ConsentDeniedEvent Gets raised in the consent UI.

UnhandledExceptionEvent Gets raised for unhandled exceptions.

DeviceAuthorizationFailureEvent & DeviceAuthorizationSuccessEvent Gets raised for successful/failed device authorization requests.

29.4 Custom events

You can create your own events and emit them via our infrastructure.

You need to derive from our base `Event` class which injects contextual information like activity ID, timestamp, etc. Your derived class can then add arbitrary data fields specific to the event context:

```
public class UserLoginFailureEvent : Event
{
    public UserLoginFailureEvent(string username, string error)
        : base(EventCategories.Authentication,
              "User Login Failure",
              EventTypes.Failure,
              EventIds.UserLoginFailure,
              error)
    {
        Username = username;
    }

    public string Username { get; set; }
}
```

Cryptography, Keys and HTTPS

IdentityServer relies on a couple of crypto mechanisms to do its job.

30.1 Token signing and validation

IdentityServer needs an asymmetric key pair to sign and validate JWTs. This keymaterial can be either packaged as a certificate or just raw keys. Both RSA and ECDSA keys are supported and the supported signing algorithms are: RS256, RS384, RS512, PS256, PS384, PS512, ES256, ES384 and ES512.

You can use multiple signing keys simultaneously, but only one signing key per algorithm is supported. The first signing key you register is considered the default signing key.

Both *clients* and *API resources* can express preferences on the signing algorithm. If you request a single token for multiple API resources, all resources need to agree on at least one allowed signing algorithm.

Loading of signing key and the corresponding validation part is done by implementations of `ISigningCredentialStore` and `IValidationKeysStore`. If you want to customize the loading of the keys, you can implement those interfaces and register them with DI.

The DI builder extensions has a couple of convenience methods to set signing and validation keys - see [here](#).

30.2 Signing key rollover

While you can only use one signing key at a time, you can publish more than one validation key to the discovery document. This is useful for key rollover.

In a nutshell, a rollover typically works like this:

1. you request/create new key material
2. you publish the new validation key in addition to the current one. You can use the `AddValidationKey` builder extension method for that.

3. all clients and APIs now have a chance to learn about the new key the next time they update their local copy of the discovery document
4. after a certain amount of time (e.g. 24h) all clients and APIs should now accept both the old and the new key material
5. keep the old key material around for as long as you like, maybe you have long-lived tokens that need validation
6. retire the old key material when it is not used anymore
7. all clients and APIs will “forget” the old key next time they update their local copy of the discovery document

This requires that clients and APIs use the discovery document, and also have a feature to periodically refresh their configuration.

Brock wrote a more detailed [blog post](#) about key rotation, and also created a [commercial component](#), that can automatically take care of all those details.

30.3 Data protection

Cookie authentication in ASP.NET Core (or anti-forgery in MVC) use the ASP.NET Core data protection feature. Depending on your deployment scenario, this might require additional configuration. See the Microsoft [docs](#) for more information.

30.4 HTTPS

We don’t enforce the use of HTTPS, but for production it is mandatory for every interaction with IdentityServer.

Grant Types

The OpenID Connect and OAuth 2.0 specifications define so-called grant types (often also called flows - or protocol flows). Grant types specify how a client can interact with the token service.

You need to specify which grant types a client can use via the `AllowedGrantTypes` property on the `Client` configuration. This allows locking down the protocol interactions that are allowed for a given client.

A client can be configured to use more than a single grant type (e.g. Authorization Code flow for user centric operations and client credentials for server to server communication). The `GrantTypes` class can be used to pick from typical grant type combinations:

```
Client.AllowedGrantTypes = GrantTypes.CodeAndClientCredentials;
```

You can also specify the grant types list manually:

```
Client.AllowedGrantTypes =  
{  
    GrantType.Code,  
    GrantType.ClientCredentials,  
    "my_custom_grant_type"  
};
```

While `IdentityServer` supports all standard grant types, you really only need to know two of them for common application scenarios.

31.1 Machine to Machine Communication

This is the simplest type of communication. Tokens are always requested on behalf of a client, no interactive user is present.

In this scenario, you send a token request to the token endpoint using the `client_credentials` grant type. The client typically has to authenticate with the token endpoint using its client ID and secret.

See the [Client Credentials Quick Start](#) for a sample how to use it.

31.2 Interactive Clients

This is the most common type of client scenario: web applications, SPAs or native/mobile apps with interactive users.

Note: Feel free to skip to the summary, if you don't care about all the technical details.

For this type of clients, the `authorization code` flow was designed. That flow consists of two physical operations:

- a front-channel step via the browser where all “interactive” things happen, e.g. login page, consent etc. This step results in an authorization code that represents the outcome of the front-channel operation.
- a back-channel step where the authorization code from step 1 gets exchanged with the requested tokens. Confidential clients need to authenticate at this point.

This flow has the following security properties:

- no data (besides the authorization code which is basically a random string) gets leaked over the browser channel
- authorization codes can only be used once
- the authorization code can only be turned into tokens when (for confidential clients - more on that later) the client secret is known

This sounds all very good - still there is one problem called `code substitution attack`. There are two modern mitigation techniques for this:

OpenID Connect Hybrid Flow

This uses a response type of `code id_token` to add an additional identity token to the response. This token is signed and protected against substitution. In addition it contains the hash of the code via the `c_hash` claim. This allows checking that you indeed got the right code (experts call this a detached signature).

This solves the problem but has the following down-sides:

- the `id_token` gets transmitted over the front-channel and might leak additional (personal identifiable) data
- all the mitigation steps (e.g. crypto) need to be implemented by the client. This results in more complicated client library implementations.

RFC 7636 - Proof Key for Code Exchange (PKCE)

This essentially introduces a per-request secret for code flow (please read up on the details [here](#)). All the client has to implement for this, is creating a random string and hashing it using SHA256.

This also solves the substitution problem, because the client can prove that it is the same client on front and back-channel, and has the following additional advantages:

- the client implementation is very simple compared to hybrid flow
- it also solves the problem of the absence of a static secret for public clients
- no additional front-channel response artifacts are needed

Summary

Interactive clients should use an authorization code-based flow. To protect against code substitution, either hybrid flow or PKCE should be used. If PKCE is available, this is the simpler solution to the problem.

PKCE is already the official recommendation for [native](#) applications and [SPAs](#) - and with the release of ASP.NET Core 3 also by default supported in the OpenID Connect handler as well.

This is how you would configure an interactive client:


```
var client = new Client
{
    ClientId = "...",

    // set client secret for confidential clients
    ClientSecret = { ... },

    // ...or turn off for public clients
    RequireClientSecret = false,

    AllowedGrantTypes = GrantTypes.Code,
    RequirePkce = true
};
```

31.3 Interactive clients without browsers or with constrained input devices

This grant type is detailed [RFC 8628](#).

This flow outsources user authentication and consent to an external device (e.g. a smart phone). It is typically used by devices that don't have proper keyboards (e.g. TVs, gaming consoles...) and can request both identity and API resources.

31.4 Custom scenarios

Extension grants allow extending the token endpoint with new grant types. See *this* for more details.

Client Authentication

In certain situations, clients need to authenticate with IdentityServer, e.g.

- confidential applications (aka clients) requesting tokens at the token endpoint
- APIs validating reference tokens at the introspection endpoint

For that purpose you can assign a list of secrets to a client or an API resource.

Secret parsing and validation is an extensibility point in identityserver, out of the box it supports shared secrets as well as transmitting the shared secret via a basic authentication header or the POST body.

32.1 Creating a shared secret

The following code sets up a hashed shared secret:

```
var secret = new Secret("secret".Sha256());
```

This secret can now be assigned to either a `Client` or an `ApiResource`. Notice that both do not only support a single secret, but multiple. This is useful for secret rollover and rotation:

```
var client = new Client
{
    ClientId = "client",
    ClientSecrets = new List<Secret> { secret },

    AllowedGrantTypes = GrantTypes.ClientCredentials,
    AllowedScopes =
    {
        "api1", "api2"
    }
};
```

In fact you can also assign a description and an expiration date to a secret. The description will be used for logging, and the expiration date for enforcing a secret lifetime:

```
var secret = new Secret(
    "secret".Sha256(),
    "2016 secret",
    new DateTime(2016, 12, 31));
```

32.2 Authentication using a shared secret

You can either send the client id/secret combination as part of the POST body:

```
POST /connect/token

client_id=client1&
client_secret=secret&
...
```

..or as a basic authentication header:

```
POST /connect/token

Authorization: Basic xxxxx

...
```

You can manually create a basic authentication header using the following C# code:

```
var credentials = string.Format("{0}:{1}", clientId, clientSecret);
var headerValue = Convert.ToBase64String(Encoding.UTF8.GetBytes(credentials));

var client = new HttpClient();
client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Basic",
    headerValue);
```

The `IdentityModel` library has helper classes called `TokenClient` and `IntrospectionClient` that encapsulate both authentication and protocol messages.

32.3 Authentication using an asymmetric Key

There are other techniques to authenticate clients, e.g. based on public/private key cryptography. IdentityServer includes support for private key JWT client secrets (see [RFC 7523](#) and [here](#)).

Secret extensibility typically consists of three things:

- a secret definition
- a secret parser that knows how to extract the secret from the incoming request
- a secret validator that knows how to validate the parsed secret based on the definition

Secret parsers and validators are implementations of the `ISecretParser` and `ISecretValidator` interfaces. To make them available to IdentityServer, you need to register them with the DI container, e.g.:

```
builder.AddSecretParser<JwtBearerClientAssertionSecretParser>()
builder.AddSecretValidator<PrivateKeyJwtSecretValidator>()
```

Extension Grants

OAuth 2.0 defines standard grant types for the token endpoint, such as `password`, `authorization_code` and `refresh_token`. Extension grants are a way to add support for non-standard token issuance scenarios like token translation, delegation, or custom credentials.

You can add support for additional grant types by implementing the `IExtensionGrantValidator` interface:

```
public interface IExtensionGrantValidator
{
    /// <summary>
    /// Handles the custom grant request.
    /// </summary>
    /// <param name="request">The validation context.</param>
    Task ValidateAsync(ExtensionGrantValidationContext context);

    /// <summary>
    /// Returns the grant type this validator can deal with
    /// </summary>
    /// <value>
    /// The type of the grant.
    /// </value>
    string GrantType { get; }
}
```

The `ExtensionGrantValidationContext` object gives you access to:

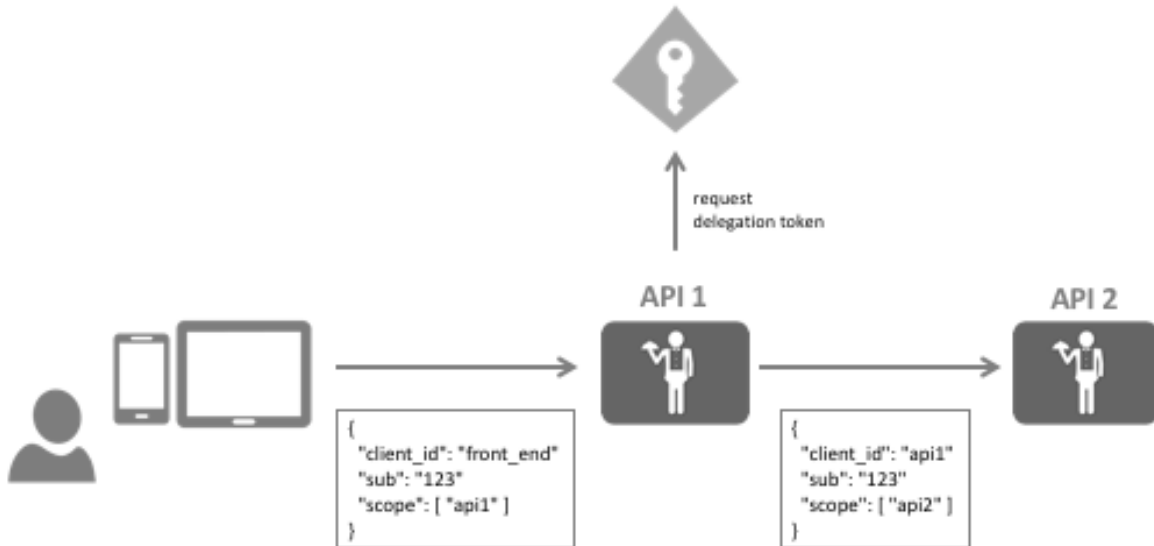
- the incoming token request - both the well-known validated values, as well as any custom values (via the `Raw` collection)
- the result - either error or success
- custom response parameters

To register the extension grant, add it to DI:

```
builder.AddExtensionGrantValidator<MyExtensionsGrantValidator>();
```

33.1 Example: Simple delegation using an extension grant

Imagine the following scenario - a front end client calls a middle tier API (API 1) using a token acquired via an interactive flow (e.g. hybrid flow). This middle tier API (API 1) now wants to call a back end API (API 2) on behalf of the interactive user:



In other words, the middle tier API (API 1) needs an access token containing the user's identity, but with the scope of the back end API (API 2).

Note: You might have heard of the term *poor man's delegation* where the access token from the front end is simply forwarded to the back end. This has some shortcomings, e.g. API 2 must now accept the API 1 scope which would allow the user to call API 2 directly. Also - you might want to add some delegation specific claims into the token, e.g. the fact that the call path is via API 1.

Implementing the extension grant

The front end would send the token to API 1, and now this token needs to be exchanged at IdentityServer with a new token for API 2.

On the wire the call to token service for the exchange could look like this:

```
POST /connect/token

grant_type=delegation&
scope=api2&
token=...&
client_id=api1.client
client_secret=secret
```

It's the job of the extension grant validator to handle that request by validating the incoming token, and returning a result that represents the new token:

```
public class DelegationGrantValidator : IExtensionGrantValidator
{
```

(continues on next page)

(continued from previous page)

```

private readonly ITokenValidator _validator;

public DelegationGrantValidator(ITokenValidator validator)
{
    _validator = validator;
}

public string GrantType => "delegation";

public async Task ValidateAsync(ExtensionGrantValidationContext context)
{
    var userToken = context.Request.Raw.Get("token");

    if (string.IsNullOrEmpty(userToken))
    {
        context.Result = new GrantValidationResult(TokenRequestErrors.
↪InvalidGrant);
        return;
    }

    var result = await _validator.ValidateAccessTokenAsync(userToken);
    if (result.IsError)
    {
        context.Result = new GrantValidationResult(TokenRequestErrors.
↪InvalidGrant);
        return;
    }

    // get user's identity
    var sub = result.Claims.FirstOrDefault(c => c.Type == "sub").Value;

    context.Result = new GrantValidationResult(sub, GrantType);
    return;
}
}

```

Don't forget to register the validator with DI.

Registering the delegation client

You need a client registration in IdentityServer that allows a client to use this new extension grant, e.g.:

```

var client = new client
{
    ClientId = "api1.client",
    ClientSecrets = new List<Secret>
    {
        new Secret("secret".Sha256())
    },

    AllowedGrantTypes = { "delegation" },

    AllowedScopes = new List<string>
    {
        "api2"
    }
}

```

Calling the token endpoint

In API 1 you can now construct the HTTP payload yourself, or use the *IdentityModel* helper library:

```
public async Task<TokenResponse> DelegateAsync(string userToken)
{
    var client = _httpClientFactory.CreateClient();
    // or
    // var client = new HttpClient();

    // send custom grant to token endpoint, return response
    return await client.RequestTokenAsync(new TokenRequest
    {
        Address = disco.TokenEndpoint,
        GrantType = "delegation",

        ClientId = "api1.client",
        ClientSecret = "secret",

        Parameters =
        {
            { "scope", "api2" },
            { "token", userToken }
        }
    });
}
```

The `TokenResponse.AccessToken` will now contain the delegation access token.

Resource Owner Password Validation

If you want to use the OAuth 2.0 resource owner password credential grant (aka password), you need to implement and register the `IResourceOwnerPasswordValidator` interface:

```
public interface IResourceOwnerPasswordValidator
{
    /// <summary>
    /// Validates the resource owner password credential
    /// </summary>
    /// <param name="context">The context.</param>
    Task ValidateAsync(ResourceOwnerPasswordValidationContext context);
}
```

On the context you will find already parsed protocol parameters like `UserName` and `Password`, but also the raw request if you want to look at other input data.

Your job is then to implement the password validation and set the `Result` on the context accordingly. See the *GrantValidationResult* documentation.

Refresh Tokens

Since access tokens have finite lifetimes, refresh tokens allow requesting new access tokens without user interaction.

Refresh tokens are supported for the following flows: authorization code, hybrid and resource owner password credential flow. The client needs to be explicitly authorized to request refresh tokens by setting `AllowOfflineAccess` to `true`.

35.1 Additional client settings

AbsoluteRefreshTokenLifetime Maximum lifetime of a refresh token in seconds. Defaults to 2592000 seconds / 30 days. Zero allows refresh tokens that, when used with `RefreshTokenExpiration = Sliding` only expire after the `SlidingRefreshTokenLifetime` is passed.

SlidingRefreshTokenLifetime Sliding lifetime of a refresh token in seconds. Defaults to 1296000 seconds / 15 days

RefreshTokenUsage `ReUse` the refresh token handle will stay the same when refreshing tokens

`OneTime` the refresh token handle will be updated when refreshing tokens

RefreshTokenExpiration `Absolute` the refresh token will expire on a fixed point in time (specified by the `AbsoluteRefreshTokenLifetime`)

`Sliding` when refreshing the token, the lifetime of the refresh token will be renewed (by the amount specified in `SlidingRefreshTokenLifetime`). The lifetime will not exceed `AbsoluteRefreshTokenLifetime`.

UpdateAccessTokenClaimsOnRefresh Gets or sets a value indicating whether the access token (and its claims) should be updated on a refresh token request.

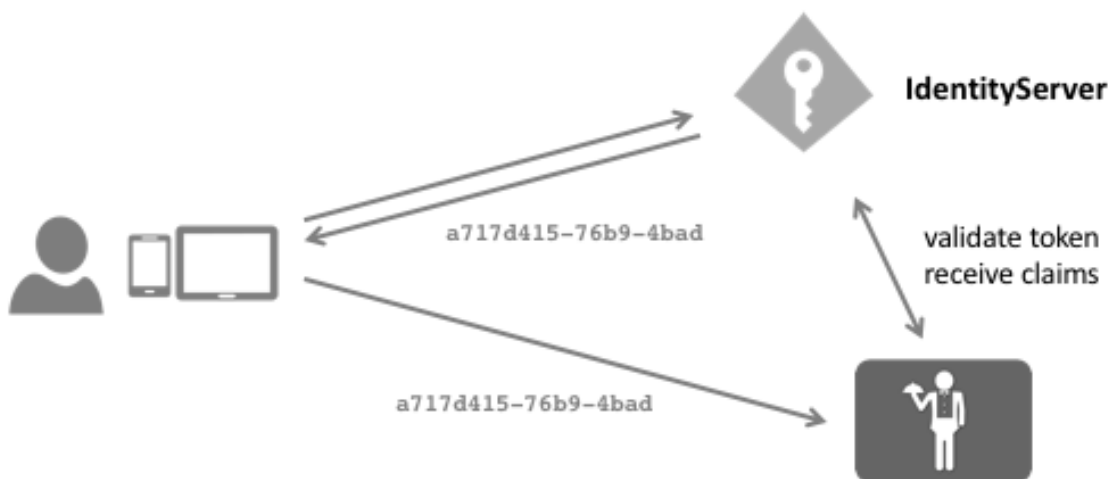
CHAPTER 36

Reference Tokens

Access tokens can come in two flavours - self-contained or reference.

A JWT token would be a self-contained access token - it's a protected data structure with claims and an expiration. Once an API has learned about the key material, it can validate self-contained tokens without needing to communicate with the issuer. This makes JWTs hard to revoke. They will stay valid until they expire.

When using reference tokens - IdentityServer will store the contents of the token in a data store and will only issue a unique identifier for this token back to the client. The API receiving this reference must then open a back-channel communication to IdentityServer to validate the token.



You can switch the token type of a client using the following setting:

```
client.AccessTokenType = AccessTokenType.Reference;
```

IdentityServer provides an implementation of the OAuth 2.0 introspection specification which allows APIs to dereference the tokens. You can either use our dedicated [introspection handler](#) or use the [identity server authentication](#)

[handler](#) which can validate both JWTs and reference tokens.

The introspection endpoint requires authentication - since the client of an introspection endpoint is an API, you configure the secret on the `ApiResource`:

```
var api = new ApiResource("api1")
{
    ApiSecrets = { new Secret("secret".Sha256()) }
}
```

See [here](#) for more information on how to configure the IdentityServer authentication middleware for APIs.

Proof-of-Possession Access Tokens

By default, OAuth access tokens are so called *bearer* tokens. This means they are not bound to a client and anybody who possess the token can use it (compare to cash).

Proof-of-Possession (short PoP) tokens are bound to the client that requested the token. If that token leaks, it cannot be used by anyone else (compare to a credit card - well at least in an ideal world).

See [this](#) blog post for more history and motivation.

IdentityServer supports PoP tokens by using the *Mutual TLS mechanism*.

Mutual TLS support in IdentityServer allows for two features:

- Client authentication to IdentityServer endpoints using a TLS X.509 client certificate
- Binding of access tokens to clients using a TLS X.509 client certificate

Note: See the “[OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens](#)” spec for more information

Setting up MTLS involves a couple of steps.

38.1 Server setup

It’s the hosting layer’s responsibility to do the actual validation of the client certificate. IdentityServer will then use that information to associate the certificate with a client and embed the certificate information in the access tokens.

Depending which server you are using, those steps are different. See [this](#) blog post for more information.

Note: [mkcert](#) is a nice utility for creating certificates for development purposes.

38.2 ASP.NET Core setup

Depending on the server setup, there are different ways how the ASP.NET Core host will receive the client certificate. While for IIS and pure Kestrel hosting, there are no additional steps, typically you have a reverse proxy in front of the application server.

This means that in addition to the typical forwarded headers handling, you also need to process the header that contains the client certificate. Add a call to `app.UseCertificateForwarding()` ; in the beginning of your middleware pipeline for that.

The exact format how proxies transmit the certificates is not standardized, that's why you need to register a callback to do the actual header parsing. The Microsoft [docs](#) show how that would work for Azure Web Apps.

If you are using Nginx (which we found is the most flexible hosting option), you need to register the following service in `ConfigureServices`:

```
services.AddCertificateForwarding(options =>
{
    // header name might be different, based on your nginx config
    options.CertificateHeader = "X-SSL-CERT";

    options.HeaderConverter = (headerValue) =>
    {
        X509Certificate2 clientCertificate = null;

        if(!string.IsNullOrEmpty(headerValue))
        {
            var bytes = Encoding.UTF8.GetBytes(Uri.UnescapeDataString(headerValue));
            clientCertificate = new X509Certificate2(bytes);
        }

        return clientCertificate;
    };
});
```

Once, the certificate has been loaded, you also need to setup the authentication handler. In this scenario we want to support self-signed certificates, hence the `CertificateType.All` and no revocation checking. These settings might be different in your environment:

```
services.AddAuthentication()
    .AddCertificate(options =>
    {
        options.AllowedCertificateTypes = CertificateTypes.All;
        options.RevocationMode = X509RevocationMode.NoCheck;
    });
```

38.3 IdentityServer setup

Next step is to enable MTLS in IdentityServer. For that you need to specify the name of the certificate authentication handler you set-up in the last step (defaults to `Certificate`), and the MTLS hosting strategy.

In IdentityServer, the mutual TLS endpoints, can be configured in three ways (assuming IdentityServer is running on `https://identityserver.io`):

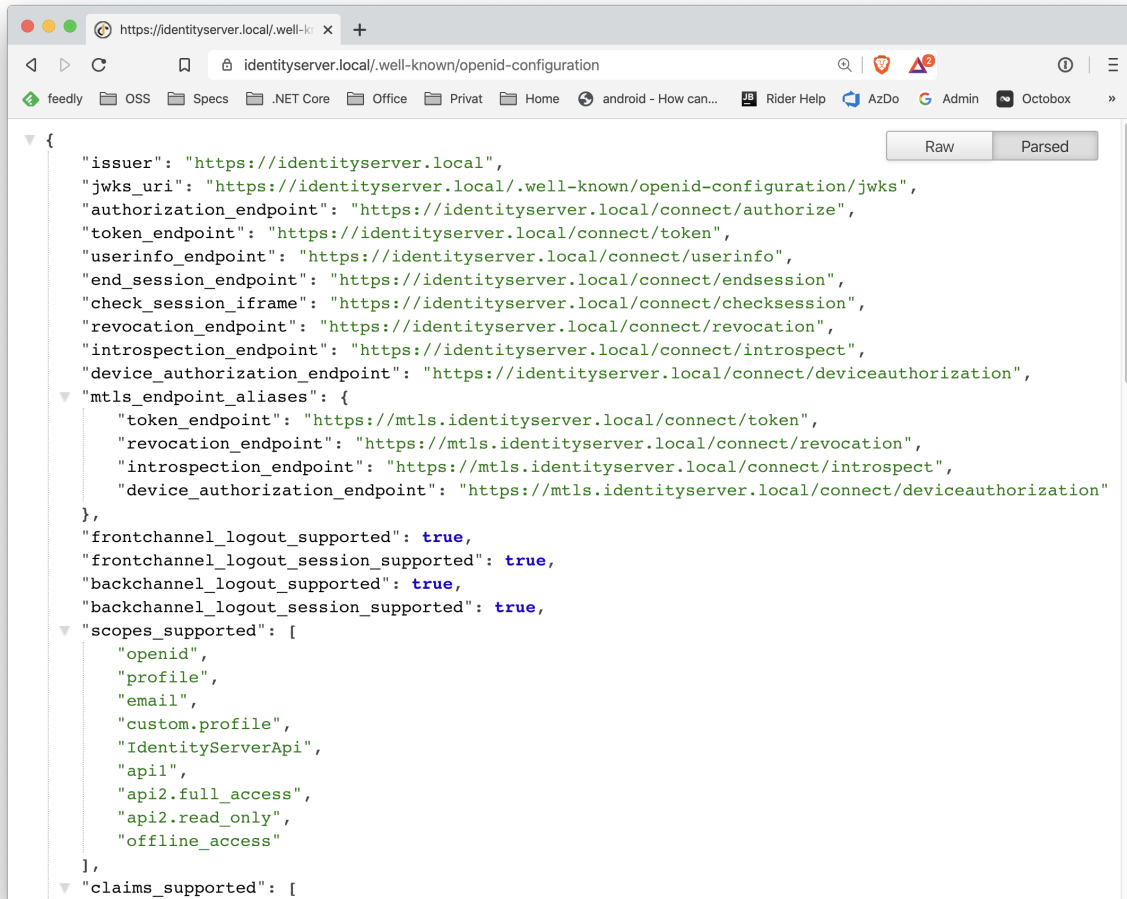
- path-based - endpoints located beneath the path `~/connect/mtls`, e.g. `https://identityserver.io/connect/mtls/token`.
- sub-domain based - endpoints are on a sub-domain of the main server, e.g. `https://mtls.identityserver.io/connect/token`.
- domain-based - endpoints are on a different domain, e.g. `https://identityserver-mtls.io`.

For example:

```
var builder = services.AddIdentityServer(options =>
{
    options.MutualTls.Enabled = true;
    options.MutualTls.ClientCertificateAuthenticationScheme = "Certificate";

    // uses sub-domain hosting
    options.MutualTls.DomainName = "mtls";
});
```

IdentityServer's discovery document reflects those endpoints:



38.4 Client authentication

Clients can use a X.509 client certificate as an authentication mechanism to endpoints in IdentityServer.

For this you need to associate a client certificate with a client in IdentityServer. Use the *IdentityServer builder* to add the services to DI which contain a default implementation to do that either thumbprint or common-name based:

```
builder.AddMutualTlsSecretValidators();
```

Finally, for the *client configuration* add to the `ClientSecrets` collection a secret type of either `SecretTypes.X509CertificateName` if you wish to authenticate the client from the certificate distinguished name or `SecretTypes.X509CertificateThumbprint` if you wish to authenticate the client by certificate thumbprint.

For example:

```
new Client
{
    ClientId = "mtls",
    AllowedGrantTypes = GrantTypes.ClientCredentials,
    AllowedScopes = { "api1" }
    ClientSecrets =
    {
        // name based
        new Secret(@"CN=mtls.test, OU=ROO\ballen@roo, O=mkcert development certificate
↪", "mtls.test")
        {
            Type = SecretTypes.X509CertificateName
        },
        // or thumbprint based
        //new Secret("bca0d040847f843c5ee0fa6eb494837470155868", "mtls.test")
        //{
        //    Type = SecretTypes.X509CertificateThumbprint
        //},
    },
}
```

38.4.1 Using a client certificate to authenticate to IdentityServer

When writing a client to connect to IdentityServer, the `SocketsHttpHandler` (or `HttpClientHandler` if you are on older .NET Framework versions) class provides a convenient mechanism to add a client certificate to outgoing requests.

And then HTTP calls (including using the various `IdentityModel` extension methods) with the `HttpClient` will perform client certificate authentication at the TLS channel.

For example:

```
static async Task<TokenResponse> RequestTokenAsync()
{
    var handler = new SocketsHttpHandler();
    var cert = new X509Certificate2("client.p12", "password");
    handler.SslOptions.ClientCertificates = new X509CertificateCollection { cert };

    var client = new HttpClient(handler);

    var disco = await client.GetDiscoveryDocumentAsync(Constants.Authority);
    if (disco.IsError) throw new Exception(disco.Error);

    var response = await client.RequestClientCredentialsTokenAsync(new
↪ClientCredentialsTokenRequest
    {
        Address = disco
            .TryGetValue(OidcConstants.Discovery.MtlsEndpointAliases)
            .Value<string>(OidcConstants.Discovery.TokenEndpoint)
            .ToString(),
    })
}
```

(continues on next page)

(continued from previous page)

```

        ClientId = "mtls",
        Scope = "api1"
    });

    if (response.IsError) throw new Exception(response.Error);
    return response;
}

```

38.5 Sender-constrained access tokens

Whenever a client authenticates to IdentityServer using a client certificate, the thumbprint of that certificate will be embedded in the access token.

Clients can use a X.509 client certificate as a mechanism for sender-constrained access tokens when authenticating to APIs. The use of these sender-constrained access tokens requires the client to use the same X.509 client certificate to authenticate to the API as the one used for IdentityServer.

38.5.1 Confirmation claim

When a client obtains an access token and has authenticated with mutual TLS, IdentityServer issues a confirmation claim (or `cnf`) in the access token. This value is a hash of the thumbprint of the client certificate used to authenticate with IdentityServer.

This value can be seen in this screen shot of a decoded access token:

```

C:\ballen\github\IdentityServer\IdentityServer4.Samples\Clients\src\ConsoleMTLSClient>dotnet run
Token response:
{
  "access_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6ImVjZTYzNGJmYTM5ODRhYTE2YTMyYzVmYjY5YWVjYjY5YWEwYjY5IiwiaWidHlwiOiJlbnRpdHlzcXJ2ZXIubG9jYyY2VzIiwiaHR0cHM6Ly9zdHMuaWRlbnRpdHlzcXJ2ZXIubG9jYyY2VzIiwiaXNjaW50ODgzM0EzMDYyYn0E1ozI0In19.mje5fxntJLUcDOaDmEwpXdkyzgl9J1015crJ8kwJ3RWLJ9yQyUZOeuaGa98u8oPy6Qz9H1X4fl3TSrvZ_jmIHvDgeMCLzIW5WlBnz8jVgxXKrSe2bhIkTCIT3CruEfvtu0HonEBu0VuDmqC8uPEStsTQmwUJ608Jx6RBOK63pvh6wrGa98_GcUb-Nv_PUQjwuGS7Rc3LY5q0ckM1sYVJrzz0PlEkPjeg6N2Re39y87tRmsk2iDrUf1ly2CrNGcMg3ENor60iz29KiyFwvWv_TVyiAVxk8-SBMFAxBVJR",
  "expires_in": 3600,
  "token_type": "Bearer"
}

Access Token (decoded):
{
  "alg": "RS256",
  "kid": "ece634bfa3984aa16a32c5fb69aa0b57",
  "typ": "JWT"
}
{
  "nbf": 1556465463,
  "exp": 1556469063,
  "iss": "https://sts.identityserver.local",
  "aud": [
    "https://sts.identityserver.local/resources",
    "api1"
  ],
  "client_id": "mtls",
  "scope": [
    "api1"
  ],
  "cnf": {
    "x5t#S256": "BF6E2CA4F07994430B86BF9D48833A33F27A5C24"
  }
}

```

The API will then use this value to ensure the client certificate being used at the API matches the confirmation value in the access token.

38.5.2 Validating and accepting a client certificate in APIs

As mentioned above for client authentication in IdentityServer, in the API the web server is expected to perform the client certificate validation at the TLS layer.

Additionally, the API hosting application will need a mechanism to accept the client certificate in order to obtain the thumbprint to perform the confirmation claim validation. Below is an example how an API in ASP.NET Core might be configured for both access tokens and client certificates:

```
services.AddAuthentication("token")
    .AddIdentityServerAuthentication("token", options =>
    {
        options.Authority = "https://identityserver.io";
        options.ApiName = "api1";

    })
    .AddCertificate(options =>
    {
        options.AllowedCertificateTypes = CertificateTypes.All;
    });
```

Finally, a mechanism is needed that runs after the authentication middleware to authenticate the client certificate and compare the thumbprint to the `cnf` from the access token.

Below is a simple middleware that checks the claims:

```
public class ConfirmationValidationMiddlewareOptions
{
    public string CertificateSchemeName { get; set; } =
    ↪CertificateAuthenticationDefaults.AuthenticationScheme;
    public string JwtBearerSchemeName { get; set; } = JwtBearerDefaults.
    ↪AuthenticationScheme;
}

// this middleware validate the cnf claim (if present) against the thumbprint of the
↪X.509 client certificate for the current client
public class ConfirmationValidationMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ConfirmationValidationMiddlewareOptions _options;

    public ConfirmationValidationMiddleware(RequestDelegate next,
    ↪ConfirmationValidationMiddlewareOptions options = null)
    {
        _next = next;
        _options = options ?? new ConfirmationValidationMiddlewareOptions();
    }

    public async Task Invoke(HttpContext ctx)
    {
        if (ctx.User.Identity.IsAuthenticated)
        {
            var cnfJson = ctx.User.FindFirst("cnf")?.Value;
            if (!String.IsNullOrEmpty(cnfJson))
            {
                var certResult = await ctx.AuthenticateAsync(_options.
                ↪CertificateSchemeName);
                if (!certResult.Succeeded)
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        {
            await ctx.ChallengeAsync(_options.CertificateSchemeName);
            return;
        }

        var certificate = await ctx.Connection.GetClientCertificateAsync();
        var thumbprint = Base64UrlTextEncoder.Encode(certificate.
↪GetCertHash(HashAlgorithmName.SHA256));

        var cnf = JObject.Parse(cnfJson);
        var sha256 = cnf.Value<string>("x5t#S256");

        if (String.IsNullOrEmpty(sha256) ||
            !thumbprint.Equals(sha256, StringComparison.Ordinal))
        {
            await ctx.ChallengeAsync(_options.JwtBearerSchemeName);
            return;
        }
    }

    await _next(ctx);
}

```

Below is an example pipeline for an API:

```

app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.
↪XForwardedProto
});

app.UseCertificateForwarding();
app.UseRouting();
app.UseAuthentication();

app.UseMiddleware<ConfirmationValidationMiddleware>(new
↪ConfirmationValidationMiddlewareOptions
{
    CertificateSchemeName = CertificateAuthenticationDefaults.AuthenticationScheme,
    JwtBearerSchemeName = "token"
});

app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});

```

Once the above middleware succeeds, then the caller has been authenticated with a sender-constrained access token.

38.5.3 Introspection and the confirmation claim

When the access token is a JWT, then the confirmation claim is contained in the token as a claim. When using reference tokens, the claims that the access token represents must be obtained via introspection. The introspection endpoint in

IdentityServer will return a `cnf` claim for reference tokens obtained via mutual TLS.

38.6 Ephemeral client certificates

You can use the IdentityServer MTLS support also to create sender-constrained access tokens without using the client certificate for client authentication. This is useful for situations where you already have client secrets in place that you don't want to change, e.g. shared secrets, or better private key JWTs.

Still, if a client certificate is present, the confirmation claim can be embedded in outgoing access tokens. And as long as the client is using the same client certificate to request the token and calling the API, this will give you the desired proof-of-possession properties.

For this enable the following setting in the options:

```
var builder = services.AddIdentityServer(options =>
{
    // other settings

    options.MutualTls.AlwaysEmitConfirmationClaim = true;
});
```

38.6.1 Using an ephemeral certificate to request a token

In this scenario, the client uses *some* client secret (a shared secret in the below sample), but attaches an additional client certificate to the token request. Since this certificate does not need to be associated with the client at the token services, it can be created on the fly:

```
static X509Certificate2 CreateClientCertificate(string name)
{
    X500DistinguishedName distinguishedName = new X500DistinguishedName($"CN={name}");

    using (RSA rsa = RSA.Create(2048))
    {
        var request = new CertificateRequest(distinguishedName, rsa,
        ↪ HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);

        request.CertificateExtensions.Add(
            new X509KeyUsageExtension(X509KeyUsageFlags.DataEncipherment |
        ↪ X509KeyUsageFlags.KeyEncipherment | X509KeyUsageFlags.DigitalSignature , false));

        request.CertificateExtensions.Add(
            new X509EnhancedKeyUsageExtension(
                new OidCollection { new Oid("1.3.6.1.5.5.7.3.2") }, false));

        return request.CreateSelfSigned(new DateTimeOffset(DateTime.UtcNow.AddDays(-
        ↪ 1)), new DateTimeOffset(DateTime.UtcNow.AddDays(10)));
    }
}
```

Then use this client certificate in addition to the already setup-up client secret:

```
static async Task<TokenResponse> RequestTokenAsync()
{
    var client = new HttpClient(GetHandler(ClientCertificate));
```

(continues on next page)

(continued from previous page)

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

var endpoint = disco
    .TryGetValue(OidcConstants.Discovery.MtlsEndpointAliases)
    .Value<string>(OidcConstants.Discovery.TokenEndpoint)
    .ToString();

var response = await client.RequestClientCredentialsTokenAsync(new
↪ClientCredentialsTokenRequest
{
    Address = endpoint,

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

if (response.IsError) throw new Exception(response.Error);
return response;
}

static SocketsHttpHandler GetHandler(X509Certificate2 certificate)
{
    var handler = new SocketsHttpHandler();
    handler.SslOptions.ClientCertificates = new X509CertificateCollection {
↪certificate };

    return handler;
}
```


Authorize Request Objects

Instead of providing the parameters for an authorize request as individual query string key/value pairs, you can package them up in signed JWTs. This makes the parameters tamper proof and you can authenticate the client already on the front-channel.

You can either transmit them by value or by reference to the authorize endpoint - see the [spec](#) for more details.

IdentityServer requires the request JWTs to be signed. We support X509 certificates and JSON web keys, e.g.:

```
var client = new Client
{
    ClientId = "foo",

    ClientSecrets =
    {
        new Secret
        {
            // X509 cert base64-encoded
            Type = IdentityServerConstants.SecretTypes.X509CertificateBase64,
            Value = Convert.ToBase64String(cert.Export(X509ContentType.Cert))
        },
        new Secret
        {
            // RSA key as JWK
            Type = IdentityServerConstants.SecretTypes.JsonWebKey,
            Value =
                "{ 'e': 'AQAB', 'kid': 'ZzAjSnraU3bkWGnnAqLapYGpTyNfLbjbzgAPbbW2GEA', 'kty'
↪ ': 'RSA', 'n': 'wWwQFtSzeRjjerpEM5Rmqz_
↪ DsNaZ9S1Bw6UbZkDLowuuTCjBWUax0vBMMxdy6XjEEK4Oq9lKMvx9JzjmeJflknoqSNrox3Ka0rnXpNAz6sATvme8p9mTXyp0
↪ S9NF5QWvpXvBeC4GAJx7QaSw4zrUkrc6XyaAiFnLhQEwKJCwUw4NOqIuYvYp_IXhw-5Ti_icDlZS-
↪ 282PcccnBeOcX7vc21pozibIdmZJKqXNsL1Ibx5Nkx1F1jLnkJAmdaACDjYRLL_
↪ 6n3W4wUp19UvzB1lGtXcJKLLkqB6YDiZNu16OSiSprfmrRXvYmvD8m6Fn15aetgKw' } "
        }
    }
}
```

Note: Microsoft.IdentityModel.Tokens.JsonWebKeyConverter has various helpers to convert keys to JWKs

39.1 Passing request JWTs by reference

If the `request_uri` parameter is used, IdentityServer will make an outgoing HTTP call to fetch the JWT from the specified URL.

You can customize the HTTP client used for this outgoing connection, e.g. to add caching or retry logic (e.g. via the Polly library):

```
builder.AddJwtRequestUriHttpClient(client =>
{
    client.Timeout = TimeSpan.FromSeconds(30);
})
.AddTransientHttpErrorPolicy(policy => policy.WaitAndRetryAsync(new[]
{
    TimeSpan.FromSeconds(1),
    TimeSpan.FromSeconds(2),
    TimeSpan.FromSeconds(3)
}));
```

Note: Request URI processing is disabled by default. Enable on the *IdentityServer Options* under Endpoints. Also see the security considerations from the JAR [specification](#).

39.2 Accessing the request object data

You can access the validated data from the request object in two ways

- wherever you have access to the `ValidatedAuthorizeRequest`, the `RequestObjectValues` dictionary holds the values
- in the UI code you can call `IIdentityServerInteractionService.GetAuthorizationContextAsync`, the resulting `AuthorizationRequest` object contains the `RequestObjectValues` dictionary as well

Custom Token Request Validation and Issuance

You can run custom code as part of the token issuance pipeline at the token endpoint. This allows e.g. for

- adding additional validation logic
- changing certain parameters (e.g. token lifetime) dynamically

For this purpose, implement (and register) the `ICustomTokenRequestValidator` interface:

```
/// <summary>
/// Allows inserting custom validation logic into token requests
/// </summary>
public interface ICustomTokenRequestValidator
{
    /// <summary>
    /// Custom validation logic for a token request.
    /// </summary>
    /// <param name="context">The context.</param>
    /// <returns>
    /// The validation result
    /// </returns>
    Task ValidateAsync(CustomTokenRequestValidationContext context);
}
```

The context object gives you access to:

- adding custom response parameters
- return an error and error description
- modifying the request parameters, e.g. access token lifetime and type, client claims, and the confirmation method

You can register your implementation of the validator using the `AddCustomTokenRequestValidator` extension method on the configuration builder.

Many endpoints in IdentityServer will be accessed via Ajax calls from JavaScript-based clients. Given that IdentityServer will most likely be hosted on a different origin than these clients, this implies that [Cross-Origin Resource Sharing](#) (CORS) will need to be configured.

41.1 Client-based CORS Configuration

One approach to configuring CORS is to use the `AllowedCorsOrigins` collection on the *client configuration*. Simply add the origin of the client to the collection and the default configuration in IdentityServer will consult these values to allow cross-origin calls from the origins.

Note: Be sure to use an origin (not a URL) when configuring CORS. For example: `https://foo:123/` is a URL, whereas `https://foo:123` is an origin.

This default CORS implementation will be in use if you are using either the “in-memory” or EF-based client configuration that we provide. If you define your own `IClientStore`, then you will need to implement your own custom CORS policy service (see below).

41.2 Custom Cors Policy Service

IdentityServer allows the hosting application to implement the `ICorsPolicyService` to completely control the CORS policy.

The single method to implement is: `Task<bool> IsOriginAllowedAsync(string origin)`. Return `true` if the *origin* is allowed, `false` otherwise.

Once implemented, simply register the implementation in DI and IdentityServer will then use your custom implementation.

DefaultCorsPolicyService

If you simply wish to hard-code a set of allowed origins, then there is a pre-built `ICorsPolicyService` implementation you can use called `DefaultCorsPolicyService`. This would be configured as a singleton in DI, and hard-coded with its `AllowedOrigins` collection, or setting the flag `AllowAll` to `true` to allow all origins. For example, in `ConfigureServices`:

```
var cors = new DefaultCorsPolicyService(_loggerFactory.CreateLogger
    <<DefaultCorsPolicyService>())
{
    AllowedOrigins = { "https://foo", "https://bar" }
};
services.AddSingleton<ICorsPolicyService>(cors);
```

Note: Use `AllowAll` with caution.

41.3 Mixing IdentityServer's CORS policy with ASP.NET Core's CORS policies

IdentityServer uses the CORS middleware from ASP.NET Core to provide its CORS implementation. It is possible that your application that hosts IdentityServer might also require CORS for its own custom endpoints. In general, both should work together in the same application.

Your code should use the documented CORS features from ASP.NET Core without regard to IdentityServer. This means you should define policies and register the middleware as normal. If your application defines policies in `ConfigureServices`, then those should continue to work in the same places you are using them (either where you configure the CORS middleware or where you use the MVC `EnableCors` attributes in your controller code). If instead you define an inline policy in the use of the CORS middleware (via the policy builder callback), then that too should continue to work normally.

The one scenario where there might be a conflict between your use of the ASP.NET Core CORS services and IdentityServer is if you decide to create a custom `ICorsPolicyProvider`. Given the design of the ASP.NET Core's CORS services and middleware, IdentityServer implements its own custom `ICorsPolicyProvider` and registers it in the DI system. Fortunately, the IdentityServer implementation is designed to use the decorator pattern to wrap any existing `ICorsPolicyProvider` that is already registered in DI. What this means is that you can also implement the `ICorsPolicyProvider`, but it simply needs to be registered prior to IdentityServer in DI (e.g. in `ConfigureServices`).

The discovery document can be found at <https://baseaddress/.well-known/openid-configuration>. It contains information about the endpoints, key material and features of your IdentityServer.

By default all information is included in the discovery document, but by using configuration options, you can hide individual sections, e.g.:

```
services.AddIdentityServer(options =>
{
    options.Discovery.ShowIdentityScopes = false;
    options.Discovery.ShowApiScopes = false;
    options.Discovery.ShowClaims = false;
    options.Discovery.ShowExtensionGrantTypes = false;
});
```

42.1 Extending discovery

You can add custom entries to the discovery document, e.g:

```
services.AddIdentityServer(options =>
{
    options.Discovery.CustomEntries.Add("my_setting", "foo");
    options.Discovery.CustomEntries.Add("my_complex_setting",
        new
        {
            foo = "foo",
            bar = "bar"
        });
});
```

When you add a custom value that starts with ~/ it will be expanded to an absolute path below the IdentityServer base address, e.g.:

```
options.Discovery.CustomEntries.Add("my_custom_endpoint", "~/custom");
```

If you want to take full control over the rendering of the discovery (and jwks) document, you can implement the `IDiscoveryResponseGenerator` interface (or derive from our default implementation).

Adding more API Endpoints

It's a common scenario to add additional API endpoints to the application hosting IdentityServer. These endpoints are typically protected by IdentityServer itself.

For simple scenarios, we give you some helpers. See the advanced section to understand more of the internal plumbing.

Note: You could achieve the same by using either our `IdentityServerAuthentication` handler or Microsoft's `JwtBearer` handler. But this is not recommended since it requires more configuration and creates dependencies on external libraries that might lead to conflicts in future updates.

Start by registering your API as an `ApiResource`, e.g.:

```
public static IEnumerable<ApiResource> Apis = new List<ApiResource>
{
    // local API
    new ApiResource(IdentityServerConstants.LocalApi.ScopeName),
};
```

..and give your clients access to this API, e.g.:

```
new Client
{
    // rest omitted
    AllowedScopes = { IdentityServerConstants.LocalApi.ScopeName },
}
```

Note: The value of `IdentityServerConstants.LocalApi.ScopeName` is `IdentityServerApi`.

To enable token validation for local APIs, add the following to your IdentityServer startup:

```
services.AddLocalApiAuthentication();
```

To protect an API controller, decorate it with an `Authorize` attribute using the `LocalApi.PolicyName` policy:

```
[Route("localApi")]
[Authorize(LocalApi.PolicyName)]
public class LocalApiController : ControllerBase
{
    public IActionResult Get()
    {
        // omitted
    }
}
```

Authorized clients can then request a token for the `IdentityServerApi` scope and use it to call the API.

43.1 Discovery

You can also add your endpoints to the discovery document if you want, e.g like this:

```
services.AddIdentityServer(options =>
{
    options.Discovery.CustomEntries.Add("local_api", "~/localapi");
})
```

43.2 Advanced

Under the covers, the `AddLocalApiAuthentication` helper does a couple of things:

- adds an authentication handler that validates incoming tokens using IdentityServer's built-in token validation engine (the name of this handler is `IdentityServerAccessToken` or `IdentityServerConstants.LocalApi.AuthenticationScheme`)
- configures the authentication handler to require a scope claim inside the access token of value `IdentityServerApi`
- sets up an authorization policy that checks for a scope claim of value `IdentityServerApi`

This covers the most common scenarios. You can customize this behavior in the following ways:

- **Add the authentication handler yourself by calling `services.AddAuthentication().AddLocalApi(...)`**
 - this way you can specify the required scope name yourself, or (by specifying no scope at all) accept any token from the current IdentityServer instance
- Do your own scope validation/authorization in your controllers using custom policies or code, e.g.:

```
services.AddAuthorization(options =>
{
    options.AddPolicy(IdentityServerConstants.LocalApi.PolicyName, policy =>
    {
        policy.AddAuthenticationSchemes(IdentityServerConstants.LocalApi.
↪AuthenticationScheme);
        policy.RequireAuthenticatedUser();
        // custom requirements
    });
});
```

43.3 Claims Transformation

You can provide a callback to transform the claims of the incoming token after validation. Either use the helper method, e.g.:

```
services.AddLocalApiAuthentication(principal =>
{
    principal.Identities.First().AddClaim(new Claim("additional_claim", "additional_
↪value"));

    return Task.FromResult(principal);
});
```

...or implement the event on the options if you add the authentication handler manually.

Adding new Protocols

IdentityServer4 allows adding support for other protocols besides the built-in support for OpenID Connect and OAuth 2.0.

You can add those additional protocol endpoints either as middleware or using e.g. MVC controllers. In both cases you have access to the ASP.NET Core DI system which allows re-using our internal services like access to client definitions or key material.

A sample for adding WS-Federation support can be found [here](#).

44.1 Typical authentication workflow

An authentication request typically works like this:

- authentication request arrives at protocol endpoint
- protocol endpoint does input validation
- **redirection to login page with a return URL set back to protocol endpoint (if user is anonymous)**
 - access to current request details via the `IIdentityServerInteractionService`
 - authentication of user (either locally or via external authentication middleware)
 - signing in the user
 - redirect back to protocol endpoint
- creation of protocol response (token creation and redirect back to client)

44.2 Useful IdentityServer services

To achieve the above workflow, some interaction points with IdentityServer are needed.

Access to configuration and redirecting to the login page

You can get access to the IdentityServer configuration by injecting the `IdentityServerOptions` class into your code. This, e.g. has the configured path to the login page:

```
var returnUrl = Url.Action("Index");
returnUrl = returnUrl.AddQueryString(Request.QueryString.Value);

var loginUrl = _options.UserInteraction.LoginUrl;
var url = loginUrl.AddQueryString(_options.UserInteraction.LoginReturnUrlParameter,
    returnUrl);

return Redirect(url);
```

Interaction between the login page and current protocol request

The `IIdentityServerInteractionService` supports turning a protocol return URL into a parsed and validated context object:

```
var context = await _interaction.GetAuthorizationContextAsync(returnUrl);
```

By default the interaction service only understands OpenID Connect protocol messages. To extend support, you can write your own `IReturnUrlParser`:

```
public interface IReturnUrlParser
{
    bool IsValidReturnUrl(string returnUrl);
    Task<AuthorizationRequest> ParseAsync(string returnUrl);
}
```

..and then register the parser in DI:

```
builder.Services.AddTransient<IReturnUrlParser, WsFederationReturnUrlParser>();
```

This allows the login page to get to information like the client configuration and other protocol parameters.

Access to configuration and key material for creating the protocol response

By injecting the `IKeyMaterialService` into your code, you get access to the configured signing credential and validation keys:

```
var credential = await _keys.GetSigningCredentialsAsync();
var key = credential.Key as Microsoft.IdentityModel.Tokens.X509SecurityKey;

var descriptor = new SecurityTokenDescriptor
{
    AppliesToAddress = result.Client.ClientId,
    Lifetime = new Lifetime(DateTime.UtcNow, DateTime.UtcNow.AddSeconds(result.Client.
    IdentityTokenLifetime)),
    ReplyToAddress = result.Client.RedirectUri.First(),
    SigningCredentials = new X509SigningCredentials(key.Certificate, result.
    RelyingParty.SignatureAlgorithm, result.RelyingParty.DigestAlgorithm),
    Subject = outgoingSubject,
    TokenIssuerName = _contextAccessor.HttpContext.GetIdentityServerIssuerUri(),
    TokenType = result.RelyingParty.TokenType
};
```

The `IdentityServerTools` class is a collection of useful internal tools that you might need when writing extensibility code for `IdentityServer`. To use it, inject it into your code, e.g. a controller:

```
public MyController(IdentityServerTools tools)
{
    _tools = tools;
}
```

The `IssueJwtAsync` method allows creating JWT tokens using the `IdentityServer` token creation engine. The `IssueClientJwtAsync` is an easier version of that for creating tokens for server-to-server communication (e.g. when you have to call an `IdentityServer` protected API from your code):

```
public async Task<IActionResult> MyAction()
{
    var token = await _tools.IssueClientJwtAsync(
        clientId: "client_id",
        lifetime: 3600,
        audiences: new[] { "backend.api" });

    // more code
}
```


CHAPTER 46

Discovery Endpoint

The discovery endpoint can be used to retrieve metadata about your IdentityServer - it returns information like the issuer name, key material, supported scopes etc. See the [spec](#) for more details.

The discovery endpoint is available via */.well-known/openid-configuration* relative to the base address, e.g.:

```
https://demo.identityserver.io/.well-known/openid-configuration
```

Note: You can use the [IdentityModel](#) client library to programmatically access the discovery endpoint from .NET code. For more information check the IdentityModel [docs](#).

Authorize Endpoint

The authorize endpoint can be used to request tokens or authorization codes via the browser. This process typically involves authentication of the end-user and optionally consent.

Note: IdentityServer supports a subset of the OpenID Connect and OAuth 2.0 authorize request parameters. For a full list, see [here](#).

client_id identifier of the client (required).

request instead of providing all parameters as individual query string parameters, you can provide a subset or all of them as a JWT

request_uri URL of a pre-packaged JWT containing request parameters

scope one or more registered scopes (required)

redirect_uri must exactly match one of the allowed redirect URIs for that client (required)

response_type `id_token` requests an identity token (only identity scopes are allowed)

`token` requests an access token (only resource scopes are allowed)

`id_token token` requests an identity token and an access token

`code` requests an authorization code

`code id_token` requests an authorization code and identity token

`code id_token token` requests an authorization code, identity token and access token

response_mode `form_post` sends the token response as a form post instead of a fragment encoded redirect (optional)

state identityserver will echo back the state value on the token response, this is for round tripping state between client and provider, correlating request and response and CSRF/replay protection. (recommended)

nonce identityserver will echo back the nonce value in the identity token, this is for replay protection)

Required for identity tokens via implicit grant.

prompt `none` no UI will be shown during the request. If this is not possible (e.g. because the user has to sign in or consent) an error is returned

`login` the login UI will be shown, even if the user is already signed-in and has a valid session

code_challenge sends the code challenge for PKCE

code_challenge_method `plain` indicates that the challenge is using plain text (not recommended) `S256` indicates the challenge is hashed with SHA256

login_hint can be used to pre-fill the username field on the login page

ui_locales gives a hint about the desired display language of the login UI

max_age if the user's logon session exceeds the max age (in seconds), the login UI will be shown

acr_values allows passing in additional authentication related information - identityserver special cases the following proprietary `acr_values`:

`idp:name_of_idp` bypasses the login/home realm screen and forwards the user directly to the selected identity provider (if allowed per client configuration)

`tenant:name_of_tenant` can be used to pass a tenant name to the login UI

Example

```
GET /connect/authorize?
  client_id=client1&
  scope=openid email api1&
  response_type=id_token token&
  redirect_uri=https://myapp/callback&
  state=abc&
  nonce=xyz
```

(URL encoding removed, and line breaks added for readability)

Note: You can use the [IdentityModel](#) client library to programmatically create authorize requests .NET code. For more information check the [IdentityModel docs](#).

CHAPTER 48

Token Endpoint

The token endpoint can be used to programmatically request tokens. It supports the `password`, `authorization_code`, `client_credentials`, `refresh_token` and `urn:ietf:params:oauth:grant-type:device_code` grant types. Furthermore the token endpoint can be extended to support extension grant types.

Note: IdentityServer supports a subset of the OpenID Connect and OAuth 2.0 token request parameters. For a full list, see [here](#).

client_id client identifier (required)

client_secret client secret either in the post body, or as a basic authentication header. Optional.

grant_type `authorization_code`, `client_credentials`, `password`, `refresh_token`, `urn:ietf:params:oauth:grant-type:device_code` or `custom`

scope one or more registered scopes. If not specified, a token for all explicitly allowed scopes will be issued.

redirect_uri required for the `authorization_code` grant type

code the authorization code (required for `authorization_code` grant type)

code_verifier PKCE proof key

username resource owner username (required for `password` grant type)

password resource owner password (required for `password` grant type)

acr_values allows passing in additional authentication related information for the `password` grant type - identityserver special cases the following proprietary `acr_values`:

`idp:name_of_idp` bypasses the login/home realm screen and forwards the user directly to the selected identity provider (if allowed per client configuration)

`tenant:name_of_tenant` can be used to pass a tenant name to the token endpoint

refresh_token the refresh token (required for `refresh_token` grant type)

device_code the device code (required for `urn:ietf:params:oauth:grant-type:device_code` grant type)

48.1 Example

```
POST /connect/token
```

```
client_id=client1&
client_secret=secret&
grant_type=authorization_code&
code=hdh922&
redirect_uri=https://myapp.com/callback
```

(Form-encoding removed and line breaks added for readability)

Note: You can use the [IdentityModel](#) client library to programmatically access the token endpoint from .NET code. For more information check the IdentityModel [docs](#).

UserInfo Endpoint

The UserInfo endpoint can be used to retrieve identity information about a user (see [spec](#)).

The caller needs to send a valid access token representing the user. Depending on the granted scopes, the UserInfo endpoint will return the mapped claims (at least the *openid* scope is required).

49.1 Example

```
GET /connect/userinfo
Authorization: Bearer <access_token>
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "sub": "248289761001",
  "name": "Bob Smith",
  "given_name": "Bob",
  "family_name": "Smith",
  "role": [
    "user",
    "admin"
  ]
}
```

Note: You can use the [IdentityModel](#) client library to programmatically access the userinfo endpoint from .NET code. For more information check the [IdentityModel docs](#).

Device Authorization Endpoint

The device authorization endpoint can be used to request device and user codes. This endpoint is used to start the device flow authorization process.

Note: The URL for the end session endpoint is available via the *discovery endpoint*.

client_id client identifier (required)

client_secret client secret either in the post body, or as a basic authentication header. Optional.

scope one or more registered scopes. If not specified, a token for all explicitly allowed scopes will be issued.

50.1 Example

```
POST /connect/deviceauthorization

  client_id=client1&
  client_secret=secret&
  scope=openid api1
```

(Form-encoding removed and line breaks added for readability)

Note: You can use the [IdentityModel](#) client library to programmatically access the device authorization endpoint from .NET code. For more information check the [IdentityModel docs](#).

Introspection Endpoint

The introspection endpoint is an implementation of [RFC 7662](#).

It can be used to validate reference tokens (or JWTs if the consumer does not have support for appropriate JWT or cryptographic libraries). The introspection endpoint requires authentication - since the client of an introspection endpoint is an API, you configure the secret on the `ApiResource`.

51.1 Example

```
POST /connect/introspect
Authorization: Basic xxxyyy

token=<token>
```

A successful response will return a status code of 200 and either an active or inactive token:

```
{
  "active": true,
  "sub": "123"
}
```

Unknown or expired tokens will be marked as inactive:

```
{
  "active": false,
}
```

An invalid request will return a 400, an unauthorized request 401.

Note: You can use the [IdentityModel](#) client library to programmatically access the introspection endpoint from .NET code. For more information check the [IdentityModel docs](#).

Revocation Endpoint

This endpoint allows revoking access tokens (reference tokens only) and refresh token. It implements the token revocation specification ([RFC 7009](#)).

token the token to revoke (required)

token_type_hint either `access_token` or `refresh_token` (optional)

52.1 Example

```
POST /connect/revocation HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

token=45ghiukldjahdnhzdauz&token_type_hint=refresh_token
```

Note: You can use the [IdentityModel](#) client library to programmatically access the revocation endpoint from .NET code. For more information check the [IdentityModel docs](#).

End Session Endpoint

The end session endpoint can be used to trigger single sign-out (see [spec](#)).

To use the end session endpoint a client application will redirect the user's browser to the end session URL. All applications that the user has logged into via the browser during the user's session can participate in the sign-out.

Note: The URL for the end session endpoint is available via the *discovery endpoint*.

53.1 Parameters

id_token_hint

When the user is redirected to the endpoint, they will be prompted if they really want to sign-out. This prompt can be bypassed by a client sending the original *id_token* received from authentication. This is passed as a query string parameter called *id_token_hint*.

post_logout_redirect_uri

If a valid *id_token_hint* is passed, then the client may also send a *post_logout_redirect_uri* parameter. This can be used to allow the user to redirect back to the client after sign-out. The value must match one of the client's pre-configured *PostLogoutRedirectUris* (*client docs*).

state

If a valid *post_logout_redirect_uri* is passed, then the client may also send a *state* parameter. This will be returned back to the client as a query string parameter after the user redirects back to the client. This is typically used by clients to round-trip state across the redirect.

53.2 Example

```
GET /connect/endsession?id_token_
→ hint=eyJhbGciOiJSUzI1NiIsImtpZCI6IjdlOGFkZmMzMjU1OTEyNzI0ZDY4NWZmYmIwOThjNDEyIiwidHlwIjoiSldUIIn0.
→ eyJuYmYiOiJlOTU3NjUzMjEsImV4cCI6MTQ5MTc2NTYyMSwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo1MDAwIiwiaXVkiJjoian
→ STzOWoeVYmtZdRAERT95cMYEmClixWkmGwVH2Yyiks9BETotbSZiSfgE5kRh72kghN78N3-
→ RgCTUmM2edB3bZx4H5ut3wWsBnZtQ2JLfhtwJAjaLE9Ykt68ovNJySbm8hjZhHzPWKh55jzshivQvTX0GdtlbcDoEA1oNONxHkp
→ rAALhFPkyKnVc-uB8IHtGNSyRWLFhwVqAdS3fRNO7iIs5hYRxeFSU7a5ZuUqZ6RRi-bcDhI-
→ djKO5uAwiyhfpbpYcaY_TxXWoCmq8N8uAw9zqFsQUwcXymfOAi2UF3eFZt02hBu-shKA&post_logout_
→ redirect_uri=http%3A%2F%2Flocalhost%3A7017%2Findex.html
```

Note: You can use the [IdentityModel](#) client library to programmatically create end_session requests .NET code. For more information check the [IdentityModel docs](#).

IdentityServer Options

- **IssuerUri** Set the issuer name that will appear in the discovery document and the issued JWT tokens. It is recommended to not set this property, which infers the issuer name from the host name that is used by the clients.
- **LowerCaseIssuerUri** Set to `false` to preserve the original casing of the `IssuerUri`. Defaults to `true`.
- **AccessTokenJwtType** Specifies the value used for the JWT typ header for access tokens (defaults to `at+jwt`).
- **EmitScopesAsSpaceDelimitedStringInJwt** Specifies whether scopes in JWTs are emitted as array or string
- **EmitStaticAudienceClaim** Emits an `aud` claim with the format `issuer/resources`. Defaults to `false`.

54.1 Endpoints

Allows enabling/disabling individual endpoints, e.g. token, authorize, userinfo etc.

By default all endpoints are enabled, but you can lock down your server by disabling endpoint that you don't need.

- **EnableJwtRequestUri** JWT `request_uri` processing is enabled on the authorize endpoint. Defaults to `false`.

54.2 Discovery

Allows enabling/disabling various sections of the discovery document, e.g. endpoints, scopes, claims, grant types etc.

The `CustomEntries` dictionary allows adding custom elements to the discovery document.

54.3 Authentication

- **CookieAuthenticationScheme** Sets the cookie authentication scheme configured by the host used for interactive users. If not set, the scheme will be inferred from the host's default authentication scheme. This setting is typically used when `AddPolicyScheme` is used in the host as the default scheme.
- **CookieLifetime** The authentication cookie lifetime (only effective if the IdentityServer-provided cookie handler is used).
- **CookieSlidingExpiration** Specifies if the cookie should be sliding or not (only effective if the IdentityServer-provided cookie handler is used).
- **CookieSameSiteMode** Specifies the SameSite mode for the internal cookies.
- **RequireAuthenticatedUserForSignOutMessage** Indicates if user must be authenticated to accept parameters to end session endpoint. Defaults to false.
- **CheckSessionCookieName** The name of the cookie used for the check session endpoint.
- **RequireCspFrameSrcForSignout** If set, will require frame-src CSP headers being emitting on the end session callback endpoint which renders iframes to clients for front-channel signout notification. Defaults to true.

54.4 Events

Allows configuring if and which events should be submitted to a registered event sink. See [here](#) for more information on events.

54.5 InputLengthRestrictions

Allows setting length restrictions on various protocol parameters like client id, scope, redirect URI etc.

54.6 UserInteraction

- **LoginUrl, LogoutUrl, ConsentUrl, ErrorUrl, DeviceVerificationUrl** Sets the URLs for the login, logout, consent, error and device verification pages.
- **LoginReturnUrlParameter** Sets the name of the return URL parameter passed to the login page. Defaults to *returnUrl*.
- **LogoutIdParameter** Sets the name of the logout message id parameter passed to the logout page. Defaults to *logoutId*.
- **ConsentReturnUrlParameter** Sets the name of the return URL parameter passed to the consent page. Defaults to *returnUrl*.
- **ErrorIdParameter** Sets the name of the error message id parameter passed to the error page. Defaults to *errorId*.
- **CustomRedirectReturnUrlParameter** Sets the name of the return URL parameter passed to a custom redirect from the authorization endpoint. Defaults to *returnUrl*.
- **DeviceVerificationUserCodeParameter** Sets the name of the user code parameter passed to the device verification page. Defaults to *userCode*.

- **CookieMessageThreshold** Certain interactions between IdentityServer and some UI pages require a cookie to pass state and context (any of the pages above that have a configurable “message id” parameter). Since browsers have limits on the number of cookies and their size, this setting is used to prevent too many cookies being created. The value sets the maximum number of message cookies of any type that will be created. The oldest message cookies will be purged once the limit has been reached. This effectively indicates how many tabs can be opened by a user when using IdentityServer.

54.7 Caching

These settings only apply if the respective caching has been enabled in the services configuration in startup.

- **ClientStoreExpiration** Cache duration of client configuration loaded from the client store.
- **ResourceStoreExpiration** Cache duration of identity and API resource configuration loaded from the resource store.

54.8 CORS

IdentityServer supports CORS for some of its endpoints. The underlying CORS implementation is provided from ASP.NET Core, and as such it is automatically registered in the dependency injection system.

- **CorsPolicyName** Name of the CORS policy that will be evaluated for CORS requests into IdentityServer (defaults to "IdentityServer4"). The policy provider that handles this is implemented in terms of the `ICorsPolicyService` registered in the dependency injection system. If you wish to customize the set of CORS origins allowed to connect, then it is recommended that you provide a custom implementation of `ICorsPolicyService`.
- **CorsPaths** The endpoints within IdentityServer where CORS is supported. Defaults to the discovery, user info, token, and revocation endpoints.
- **PreflightCacheDuration** *Nullable<TimeSpan>* indicating the value to be used in the preflight *Access-Control-Max-Age* response header. Defaults to *null* indicating no caching header is set on the response.

54.9 CSP (Content Security Policy)

IdentityServer emits CSP headers for some responses, where appropriate.

- **Level** The level of CSP to use. CSP Level 2 is used by default, but if older browsers must be supported then this be changed to `CspLevel1` to accomodate them.
- **AddDeprecatedHeader** Indicates if the older `X-Content-Security-Policy` CSP header should also be emitted (in addition to the standards-based header value). Defaults to true.

54.10 Device Flow

- **DefaultUserCodeType** The user code type to use, unless set at the client level. Defaults to *Numeric*, a 9-digit code.
- **Interval** Defines the minimum allowed polling interval on the token endpoint. Defaults to 5.

54.11 Mutual TLS

- **Enabled** Specifies if MTLS support should be enabled. Defaults to *false*.
- **ClientCertificateAuthenticationScheme** Specifies the name of the authentication handler for X.509 client certificates. Defaults to "Certificate".
- **DomainName** Specifies either the name of the sub-domain or full domain for running the MTLS endpoints (will use path-based endpoints if not set). Use a simple string (e.g. "mtls") to set a sub-domain, use a full domain name (e.g. "identityserver-mtls.io") to set a full domain name. When a full domain name is used, you also need to set the *IssuerName* to a fixed value.
- **AlwaysEmitConfirmationClaim** Specifies whether a *cnf* claim gets emitted for access tokens if a client certificate was present. Normally the *cnf* claims only gets emitted if the client used the client certificate for authentication, setting this to true, will set the claim regardless of the authentication method. (defaults to false).

Identity Resource

This class models an identity resource.

Enabled Indicates if this resource is enabled and can be requested. Defaults to true.

Name The unique name of the identity resource. This is the value a client will use for the scope parameter in the authorize request.

DisplayName This value will be used e.g. on the consent screen.

Description This value will be used e.g. on the consent screen.

Required Specifies whether the user can de-select the scope on the consent screen (if the consent screen wants to implement such a feature). Defaults to false.

Emphasize Specifies whether the consent screen will emphasize this scope (if the consent screen wants to implement such a feature). Use this setting for sensitive or important scopes. Defaults to false.

ShowInDiscoveryDocument Specifies whether this scope is shown in the discovery document. Defaults to true.

UserClaims List of associated user claim types that should be included in the identity token.

This class models an OAuth scope.

Enabled Indicates if this resource is enabled and can be requested. Defaults to true.

Name The unique name of the API. This value is used for authentication with introspection and will be added to the audience of the outgoing access token.

DisplayName This value can be used e.g. on the consent screen.

Description This value can be used e.g. on the consent screen.

UserClaims List of associated user claim types that should be included in the access token.

56.1 Defining API scope in appsettings.json

The `AddInMemoryApiResource` extensions method also supports adding clients from the ASP.NET Core configuration file:

```
"IdentityServer": {  
  "IssuerUri": "urn:sso.company.com",  
  "ApiScopes": [  
    {  
      "Name": "IdentityServerApi"  
    },  
    {  
      "Name": "resource1.scope1"  
    },  
    {  
      "Name": "resource2.scope1"  
    },  
    {  
      "Name": "scope3"  
    },  
    {  

```

(continues on next page)

(continued from previous page)

```
        "Name": "shared.scope"
    },
    {
        "Name": "transaction",
        "DisplayName": "Transaction",
        "Description": "A transaction"
    }
]
```

Then pass the configuration section to the `AddInMemoryApiScopes` method:

```
AddInMemoryApiResources(configuration.GetSection("IdentityServer:ApiScopes"))
```

This class model an API resource.

Enabled Indicates if this resource is enabled and can be requested. Defaults to true.

Name The unique name of the API. This value is used for authentication with introspection and will be added to the audience of the outgoing access token.

DisplayName This value can be used e.g. on the consent screen.

Description This value can be used e.g. on the consent screen.

ApiSecrets The API secret is used for the introspection endpoint. The API can authenticate with introspection using the API name and secret.

AllowedAccessTokenSigningAlgorithms List of allowed signing algorithms for access token. If empty, will use the server default signing algorithm.

UserClaims List of associated user claim types that should be included in the access token.

Scopes List of API scope names.

57.1 Defining API resources in appsettings.json

The `AddInMemoryApiResource` extensions method also supports adding API resources from the ASP.NET Core configuration file:

```
"IdentityServer": {  
  "IssuerUri": "urn:sso.company.com",  
  "ApiResources": [  
    {  
      "Name": "resource1",  
      "DisplayName": "Resource #1",  
  
      "Scopes": [  
        "resource1.scope1",
```

(continues on next page)

(continued from previous page)

```
        "shared.scope"
      ]
    },
    {
      "Name": "resource2",
      "DisplayName": "Resource #2",

      "UserClaims": [
        "name",
        "email"
      ],

      "Scopes": [
        "resource2.scope1",
        "shared.scope"
      ]
    }
  ]
}
```

Then pass the configuration section to the `AddInMemoryApiResource` method:

```
AddInMemoryApiResources(configuration.GetSection("IdentityServer:ApiResources"))
```

The `Client` class models an OpenID Connect or OAuth 2.0 client - e.g. a native application, a web application or a JS-based application.

58.1 Basics

Enabled Specifies if client is enabled. Defaults to *true*.

ClientId Unique ID of the client

ClientSecrets List of client secrets - credentials to access the token endpoint.

RequireClientSecret Specifies whether this client needs a secret to request tokens from the token endpoint (defaults to *true*)

RequireRequestObject Specifies whether this client needs to wrap the authorize request parameters in a JWT (defaults to *false*)

AllowedGrantTypes Specifies the grant types the client is allowed to use. Use the `GrantTypes` class for common combinations.

RequirePkce Specifies whether clients using an authorization code based grant type must send a proof key (defaults to *true*).

AllowPlainTextPkce Specifies whether clients using PKCE can use a plain text code challenge (not recommended - and default to *false*)

RedirectUri Specifies the allowed URIs to return tokens or authorization codes to

AllowedScopes By default a client has no access to any resources - specify the allowed resources by adding the corresponding scopes names

AllowOfflineAccess Specifies whether this client can request refresh tokens (be requesting the `offline_access` scope)

AllowAccessTokensViaBrowser Specifies whether this client is allowed to receive access tokens via the browser. This is useful to harden flows that allow multiple response types (e.g. by disallowing a hybrid flow)

client that is supposed to use *code id_token* to add the *token* response type and thus leaking the token to the browser.

Properties Dictionary to hold any custom client-specific values as needed.

58.2 Authentication/Logout

PostLogoutRedirectUri Specifies allowed URIs to redirect to after logout. See the [OIDC Connect Session Management spec](#) for more details.

FrontChannelLogoutUri Specifies logout URI at client for HTTP based front-channel logout. See the [OIDC Front-Channel spec](#) for more details.

FrontChannelLogoutSessionRequired Specifies if the user's session id should be sent to the FrontChannelLogoutUri. Defaults to true.

BackChannelLogoutUri Specifies logout URI at client for HTTP based back-channel logout. See the [OIDC Back-Channel spec](#) for more details.

BackChannelLogoutSessionRequired Specifies if the user's session id should be sent in the request to the BackChannelLogoutUri. Defaults to true.

EnableLocalLogin Specifies if this client can use local accounts, or external IdPs only. Defaults to *true*.

IdentityProviderRestrictions Specifies which external IdPs can be used with this client (if list is empty all IdPs are allowed). Defaults to empty.

UserSsoLifetime *added in 2.3* The maximum duration (in seconds) since the last time the user authenticated. Defaults to *null*. You can adjust the lifetime of a session token to control when and how often a user is required to reenter credentials instead of being silently authenticated, when using a web application.

58.3 Token

IdentityTokenLifetime Lifetime to identity token in seconds (defaults to 300 seconds / 5 minutes)

AllowedIdentityTokenSigningAlgorithms List of allowed signing algorithms for identity token. If empty, will use the server default signing algorithm.

AccessTokenLifetime Lifetime of access token in seconds (defaults to 3600 seconds / 1 hour)

AuthorizationCodeLifetime Lifetime of authorization code in seconds (defaults to 300 seconds / 5 minutes)

AbsoluteRefreshTokenLifetime Maximum lifetime of a refresh token in seconds. Defaults to 2592000 seconds / 30 days

SlidingRefreshTokenLifetime Sliding lifetime of a refresh token in seconds. Defaults to 1296000 seconds / 15 days

RefreshTokenUsage *ReUse* the refresh token handle will stay the same when refreshing tokens

OneTime the refresh token handle will be updated when refreshing tokens. This is the default.

RefreshTokenExpiration *Absolute* the refresh token will expire on a fixed point in time (specified by the *AbsoluteRefreshTokenLifetime*)

Sliding when refreshing the token, the lifetime of the refresh token will be renewed (by the amount specified in *SlidingRefreshTokenLifetime*). The lifetime will not exceed *AbsoluteRefreshTokenLifetime*.

UpdateAccessTokenClaimsOnRefresh Gets or sets a value indicating whether the access token (and its claims) should be updated on a refresh token request.

AccessTokenType Specifies whether the access token is a reference token or a self contained JWT token (defaults to *Jwt*).

IncludeJwtId Specifies whether JWT access tokens should have an embedded unique ID (via the *jti* claim). Defaults to `true`.

AllowedCorsOrigins If specified, will be used by the default CORS policy service implementations (In-Memory and EF) to build a CORS policy for JavaScript clients.

Claims Allows settings claims for the client (will be included in the access token).

AlwaysSendClientClaims If set, the client claims will be sent for every flow. If not, only for client credentials flow (default is *false*)

AlwaysIncludeUserClaimsInIdToken When requesting both an id token and access token, should the user claims always be added to the id token instead of requiring the client to use the *userinfo* endpoint. Default is *false*.

ClientClaimsPrefix If set, the prefix client claim types will be prefixed with. Defaults to *client_*. The intent is to make sure they don't accidentally collide with user claims.

PairWiseSubjectSalt Salt value used in pair-wise *subjectId* generation for users of this client.

58.4 Consent Screen

RequireConsent Specifies whether a consent screen is required. Defaults to *false*.

AllowRememberConsent Specifies whether user can choose to store consent decisions. Defaults to `true`.

ConsentLifetime Lifetime of a user consent in seconds. Defaults to null (no expiration).

ClientName Client display name (used for logging and consent screen)

ClientUri URI to further information about client (used on consent screen)

LogoUri URI to client logo (used on consent screen)

58.5 Device flow

UserCodeType Specifies the type of user code to use for the client. Otherwise falls back to default.

DeviceCodeLifetime Lifetime to device code in seconds (defaults to 300 seconds / 5 minutes)

GrantValidationResult

The `GrantValidationResult` class models the outcome of grant validation for extensions grants and resource owner password grants.

The most common usage is to either new it up using an identity (success case):

```
context.Result = new GrantValidationResult(  
    subject: "818727",  
    authenticationMethod: "custom",  
    claims: optionalClaims);
```

...or using an error and description (failure case):

```
context.Result = new GrantValidationResult(  
    TokenRequestErrors.InvalidGrant,  
    "invalid custom credential");
```

In both case you can pass additional custom values that will be included in the token response.

Often `IdentityServer` requires identity information about users when creating tokens or when handling requests to the `userinfo` or `introspection` endpoints. By default, `IdentityServer` only has the claims in the authentication cookie to draw upon for this identity data.

It is impractical to put all of the possible claims needed for users into the cookie, so `IdentityServer` defines an extensibility point for allowing claims to be dynamically loaded as needed for a user. This extensibility point is the `IProfileService` and it is common for a developer to implement this interface to access a custom database or API that contains the identity data for users.

60.1 `IProfileService` APIs

GetProfileDataAsync The API that is expected to load claims for a user. It is passed an instance of `ProfileDataRequestContext`.

IsActiveAsync The API that is expected to indicate if a user is currently allowed to obtain tokens. It is passed an instance of `IsActiveContext`.

60.2 `ProfileDataRequestContext`

Models the request for user claims and is the vehicle to return those claims. It contains these properties:

Subject The `ClaimsPrincipal` modeling the user.

Client The `Client` for which the claims are being requested.

RequestedClaimTypes The collection of claim types being requested.

Caller An identifier for the context in which the claims are being requested (e.g. an identity token, an access token, or the user info endpoint). The constant `IdentityServerConstants.ProfileDataCallers` contains the different constant values.

IssuedClaims The list of `Claims` that will be returned. This is expected to be populated by the custom `IProfileService` implementation.

AddRequestedClaims Extension method on the `ProfileDataRequestContext` to populate the `IssuedClaims`, but first filters the claims based on `RequestedClaimTypes`.

60.3 Requested scopes and claims mapping

The scopes requested by the client control what user claims are returned in the tokens to the client. The `GetProfileDataAsync` method is responsible for dynamically obtaining those claims based on the `RequestedClaimTypes` collection on the `ProfileDataRequestContext`.

The `RequestedClaimTypes` collection is populated based on the user claims defined on the *resources* that model the scopes. If the scopes requested are an *identity resources*, then the claims in the `RequestedClaimTypes` will be populated based on the user claim types defined in the `IdentityResource`. If the scopes requested are an *API resources*, then the claims in the `RequestedClaimTypes` will be populated based on the user claim types defined in the `ApiResource` and/or the `Scope`.

60.4 IsActiveContext

Models the request to determine if the user is currently allowed to obtain tokens. It contains these properties:

Subject The `ClaimsPrincipal` modeling the user.

Client The `Client` for which the claims are being requested.

Caller An identifier for the context in which the claims are being requested (e.g. an identity token, an access token, or the user info endpoint). The constant `IdentityServerConstants.ProfileDataCallers` contains the different constant values.

IsActive The flag indicating if the user is allowed to obtain tokens. This is expected to be assigned by the custom `IProfileService` implementation.

IdentityServer Interaction Service

The `IIdentityServerInteractionService` interface is intended to provide services to be used by the user interface to communicate with IdentityServer, mainly pertaining to user interaction. It is available from the dependency injection system and would normally be injected as a constructor parameter into your MVC controllers for the user interface of IdentityServer.

61.1 IIdentityServerInteractionService APIs

GetAuthorizationContextAsync Returns the `AuthorizationRequest` based on the `returnUrl` passed to the login or consent pages.

IsValidReturnUrl Indicates if the `returnUrl` is a valid URL for redirect after login or consent.

GetErrorContextAsync Returns the `ErrorMessage` based on the `errorId` passed to the error page.

GetLogoutContextAsync Returns the `LogoutRequest` based on the `logoutId` passed to the logout page.

CreateLogoutContextAsync Used to create a `logoutId` if there is not one presently. This creates a cookie capturing all the current state needed for signout and the `logoutId` identifies that cookie. This is typically used when there is no current `logoutId` and the logout page must capture the current user's state needed for sign-out prior to redirecting to an external identity provider for signout. The newly created `logoutId` would need to be round-tripped to the external identity provider at signout time, and then used on the signout callback page in the same way it would be on the normal logout page.

GrantConsentAsync Accepts a `ConsentResponse` to inform IdentityServer of the user's consent to a particular `AuthorizationRequest`.

GetAllUserConsentsAsync Returns a collection of `Consent` for the user.

RevokeUserConsentAsync Revokes all of a user's consents and grants for a client.

RevokeTokensForCurrentSessionAsync Revokes all of a user's consents and grants for clients the user has signed into during their current session.

61.2 AuthorizationRequest

ClientId The client identifier that initiated the request.

RedirectUri The URI to redirect the user to after successful authorization.

DisplayMode The display mode passed from the authorization request.

UiLocales The UI locales passed from the authorization request.

IdP The external identity provider requested. This is used to bypass home realm discovery (HRD). This is provided via the “idp:” prefix to the `acr_values` parameter on the authorize request.

Tenant The tenant requested. This is provided via the “tenant:” prefix to the `acr_values` parameter on the authorize request.

LoginHint The expected username the user will use to login. This is requested from the client via the `login_hint` parameter on the authorize request.

PromptMode The prompt mode requested from the authorization request.

AcrValues The acr values passed from the authorization request.

ScopesRequested The scopes requested from the authorization request.

Parameters The entire parameter collection passed to the authorization request.

61.3 ErrorMessage

DisplayMode The display mode passed from the authorization request.

UiLocales The UI locales passed from the authorization request.

Error The error code.

RequestId The per-request identifier. This can be used to display to the end user and can be used in diagnostics.

61.4 LogoutRequest

ClientId The client identifier that initiated the request.

PostLogoutRedirectUri The URL to redirect the user to after they have logged out.

SessionId The user’s current session id.

SignOutIFrameUrl The URL to render in an `<iframe>` on the logged out page to enable single sign-out.

Parameters The entire parameter collection passed to the end session endpoint.

ShowSignoutPrompt Indicates if the user should be prompted for signout based upon the parameters passed to the end session endpoint.

61.5 ConsentResponse

ScopesConsented The collection of scopes the user consented to.

RememberConsent Flag indicating if the user’s consent is to be persisted.

61.6 Consent

SubjectId The subject id that granted the consent.

ClientId The client identifier for the consent.

Scopes The collection of scopes consented to.

CreationTime The date and time when the consent was granted.

Expiration The date and time when the consent will expire.

Device Flow Interaction Service

The `IDeviceFlowInteractionService` interface is intended to provide services to be used by the user interface to communicate with `IdentityServer` during device flow authorization. It is available from the dependency injection system and would normally be injected as a constructor parameter into your MVC controllers for the user interface of `IdentityServer`.

62.1 `IDeviceFlowInteractionService` APIs

`GetAuthorizationContextAsync` Returns the `DeviceFlowAuthorizationRequest` based on the `userCode` passed to the login or consent pages.

`DeviceFlowInteractionResult` Completes device authorization for the given `userCode`.

62.2 `DeviceFlowAuthorizationRequest`

`ClientId` The client identifier that initiated the request.

`ScopesRequested` The scopes requested from the authorization request.

62.3 `DeviceFlowInteractionResult`

`IsError` Specifies if the authorization request errored.

`ErrorDescription` Error description upon failure.

Entity Framework Support

An EntityFramework-based implementation is provided for the configuration and operational data extensibility points in IdentityServer. The use of EntityFramework allows any EF-supported database to be used with this library.

The code for this library is located [here](#) (with the underlying storage code [here](#)) and the NuGet package is [here](#).

The features provided by this library are broken down into two main areas: configuration store and operational store support. These two different areas can be used independently or together, based upon the needs of the hosting application.

63.1 Configuration Store support for Clients, Resources, and CORS settings

If client, identity resource, API resource, or CORS data is desired to be loaded from a EF-supported database (rather than use in-memory configuration), then the configuration store can be used. This support provides implementations of the `IClientStore`, `IResourceStore`, and the `ICorsPolicyService` extensibility points. These implementations use a `DbContext`-derived class called `ConfigurationDbContext` to model the tables in the database.

To use the configuration store support, use the `AddConfigurationStore` extension method after the call to `AddIdentityServer`:

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    const string connectionString = @"Data Source=(LocalDb)\MSSQLLocalDB;
    ↪database=IdentityServer4.EntityFramework-2.0.0;trusted_connection=yes;";
    var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

    services.AddIdentityServer()
        // this adds the config data from DB (clients, resources, CORS)
        .AddConfigurationStore(options =>
        {
            options.ConfigureDbContext = builder =>
```

(continues on next page)

(continued from previous page)

```

        builder.UseSqlServer(connectionString,
            sql => sql.MigrationsAssembly(migrationsAssembly));
    });
}

```

To configure the configuration store, use the `ConfigurationStoreOptions` options object passed to the configuration callback.

63.2 ConfigurationStoreOptions

This options class contains properties to control the configuration store and `ConfigurationDbContext`.

ConfigureDbContext Delegate of type `Action<DbContextOptionsBuilder>` used as a callback to configure the underlying `ConfigurationDbContext`. The delegate can configure the `ConfigurationDbContext` in the same way if EF were being used directly with `AddDbContext`, which allows any EF-supported database to be used.

DefaultSchema Allows setting the default database schema name for all the tables in the `ConfigurationDbContext`

```
options.DefaultSchema = "myConfigurationSchema";
```

If you need to change the schema for the Migration History Table, you can chain another action to the `UseSqlServer`:

```

options.ConfigureDbContext = b =>
    b.UseSqlServer(connectionString,
        sql => sql.MigrationsAssembly(migrationsAssembly).MigrationsHistoryTable(
            ↪ "MyConfigurationMigrationTable", "myConfigurationSchema"));

```

63.3 Operational Store support for authorization grants, consents, and tokens (refresh and reference)

If authorization grants, consents, and tokens (refresh and reference) are desired to be loaded from a EF-supported database (rather than the default in-memory database), then the operational store can be used. This support provides implementations of the `IPersistedGrantStore` extensibility point. The implementation uses a `DbContext`-derived class called `PersistedGrantDbContext` to model the table in the database.

To use the operational store support, use the `AddOperationalStore` extension method after the call to `AddIdentityServer`:

```

public IServiceCollection ConfigureServices(IServiceCollection services)
{
    const string connectionString = @"Data Source=(LocalDb)\MSSQLLocalDB;
    ↪ database=IdentityServer4.EntityFramework-2.0.0;trusted_connection=yes;";
    var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

    services.AddIdentityServer()
        // this adds the operational data from DB (codes, tokens, consents)
        .AddOperationalStore(options =>
        {

```

(continues on next page)

(continued from previous page)

```

options.ConfigureDbContext = builder =>
    builder.UseSqlServer(connectionString,
        sql => sql.MigrationsAssembly(migrationsAssembly));

    // this enables automatic token cleanup. this is optional.
options.EnableTokenCleanup = true;
options.TokenCleanupInterval = 3600; // interval in seconds (default is 3600)
    });
}

```

To configure the operational store, use the `OperationalStoreOptions` options object passed to the configuration callback.

63.4 OperationalStoreOptions

This options class contains properties to control the operational store and `PersistedGrantDbContext`.

ConfigureDbContext Delegate of type `Action<DbContextOptionsBuilder>` used as a callback to configure the underlying `PersistedGrantDbContext`. The delegate can configure the `PersistedGrantDbContext` in the same way if EF were being used directly with `AddDbContext`, which allows any EF-supported database to be used.

DefaultSchema Allows setting the default database schema name for all the tables in the `PersistedGrantDbContext`.

EnableTokenCleanup Indicates whether stale entries will be automatically cleaned up from the database. The default is `false`.

TokenCleanupInterval The token cleanup interval (in seconds). The default is 3600 (1 hour).

63.5 Database creation and schema changes across different versions of IdentityServer

It is very likely that across different versions of IdentityServer (and the EF support) that the database schema will change to accommodate new and changing features.

We do not provide any support for creating your database or migrating your data from one version to another. You are expected to manage the database creation, schema changes, and data migration in any way your organization sees fit.

Using EF migrations is one possible approach to this. If you do wish to use migrations, then see the *EF quickstart* for samples on how to get started, or consult the Microsoft [documentation on EF migrations](#).

We also publish [sample SQL scripts](#) for the current version of the database schema.

ASP.NET Identity Support

An ASP.NET Identity-based implementation is provided for managing the identity database for users of IdentityServer. This implementation implements the extensibility points in IdentityServer needed to load identity data for your users to emit claims into tokens.

The repo for this support is located [here](#) and the NuGet package is [here](#).

To use this library, configure ASP.NET Identity normally. Then use the `AddAspNetIdentity` extension method after the call to `AddIdentityServer`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddIdentityServer()
        .AddAspNetIdentity<ApplicationUser>();
}
```

`AddAspNetIdentity` requires as a generic parameter the class that models your user for ASP.NET Identity (and the same one passed to `AddIdentity` to configure ASP.NET Identity). This configures IdentityServer to use the ASP.NET Identity implementations of `IUserClaimsPrincipalFactory`, `IResourceOwnerPasswordValidator`, and `IProfileService`. It also configures some of ASP.NET Identity's options for use with IdentityServer (such as claim types to use and authentication cookie settings).

When using your own implementation of `IUserClaimsPrincipalFactory`, make sure that you register it before calling the IdentityServer `AddAspNetIdentity` extension method.

Here are some online, remote and classroom training options to learn more about ASP.NET Core identity & IdentityServer4.

65.1 Identity & Access Control for modern Applications (using ASP.NET Core 2 and IdentityServer4)

That's our own three day flagship course (including extensive hands-on labs) that we deliver as part of conferences, on-sites and remote.

The agenda and dates for public training can be found [here](#), [contact](#) us for private workshops.

65.2 PluralSight courses

There are some good courses on PluralSight around identity, ASP.NET Core and IdentityServer.

new

- [Securing Angular Apps with OpenID and OAuth2](#)
- [ASP.NET Core Identity Management Playbook](#)
- [Getting Started with ASP.NET Core and OAuth](#)
- [Securing ASP.NET Core with OAuth2 and OpenID Connect](#)
- [Understanding ASP.NET Core Security \(Centralized Authentication with a Token Service\)](#)

older

- [Introduction to OAuth2, OpenID Connect and JSON Web Tokens \(JWT\)](#)
- [Web API v2 Security](#)
- [Using OAuth to Secure Your ASP.NET API](#)

- [OAuth2 and OpenID Connect Strategies for Angular and ASP.NET](#)

66.1 Team posts

66.1.1 2019

- Scope and claims design in IdentityServer
- Try Device Flow with IdentityServer4
- The State of the Implicit Flow in OAuth2
- An alternative way to secure SPAs (with ASP.NET Core, OpenID Connect, OAuth 2.0 and ProxyKit)
- Automatic OAuth 2.0 Token Management in ASP.NET Core
- Encrypting Identity Tokens in IdentityServer4

66.1.2 2018

- IdentityServer4 Update
- IdentityServer and Swagger
- <https://www.scottbrady91.com/OAuth/Removing-Shared-Secrets-for-OAuth-Client-Authentication>
- Creating Your Own IdentityServer4 Storage Library

66.1.3 2017

- Platforms where you can run IdentityServer4
- Optimizing Tokens for size
- Identity vs Permissions
- Bootstrapping OpenID Connect: Discovery

- [Extending IdentityServer4 with WS-Federation Support](#)
- [Announcing IdentityServer4 RC1](#)
- [Getting Started with IdentityServer 4](#)
- [IdentityServer 4 SharePoint Integration using WS-Federation](#)

66.2 What's new posts

- [New in IdentityServer4 v2: Simplified Configuration behind Load-balancers or Reverse-Proxies](#)
- [New in IdentityServer4: Clients without Secrets](#)
- [New in IdentityServer4: Default Scopes](#)
- [New in IdentityServer4: Support for Extension Grants](#)
- [New in IdentityServer4: Resource Owner Password Validation](#)
- [New in IdentityServer4: Resource-based Configuration](#)
- [New in IdentityServer4: Events](#)

66.3 Community posts

- [OAuth 2.0 - OpenID Connect & IdentityServer](#)
- [Running IdentityServer4 in a Docker Container](#)
- [Connecting Zendesk and IdentityServer 4 SAML 2.0 Identity Provider](#)
- [IdentityServer localization using ui_locales](#)
- [Self-issuing an IdentityServer4 token in an IdentityServer4 service](#)
- [IdentityServer4 on the ASP.NET Team Blog](#)
- [Angular2 OpenID Connect Implicit Flow with IdentityServer4](#)
- [Full Server Logout with IdentityServer4 and OpenID Connect Implicit Flow](#)
- [IdentityServer4, ASP.NET Identity, Web API and Angular in a single Project](#)
- [Secure your .NETCore web applications using IdentityServer 4](#)
- [ASP.NET Core IdentityServer4 Resource Owner Password Flow with custom UserRepository](#)
- [Secure ASP.NET Core MVC with Angular using IdentityServer4 OpenID Connect Hybrid Flow](#)
- [Adding an external Microsoft login to IdentityServer4](#)
- [Implementing Two-factor authentication with IdentityServer4 and Twilio](#)
- [Security Experiments with gRPC and ASP.NET Core 3.0](#)
- [ASP.NET Core OAuth Device Flow Client with IdentityServer4](#)
- [Securing a Vue.js app using OpenID Connect Code Flow with PKCE and IdentityServer4](#)
- [Using an OData Client with an ASP.NET Core API](#)
- [OpenID Connect back-channel logout using Azure Redis Cache and IdentityServer4](#)
- [Single Sign Out in IdentityServer4 with Back Channel Logout](#)

67.1 2020

- January [NDC London] – Implementing OpenID Connect and OAuth 2.0 – Tips from the Trenches
- January [NDC London] – OpenID Connect & OAuth 2.0 – Security Best Practices

67.2 2019

- October [TDC] – Securing Web Applications and APIs with ASP.NET Core 3.0
- January [NDC] – Securing Web Applications and APIs with ASP.NET Core 2.2 and 3.0
- January [NDC] – Building Clients for OpenID Connect/OAuth 2-based Systems

67.3 2018

- 26/09 [DevConf] – Authorization for modern Applications
- 17/01 [NDC London] – IdentityServer v2 on ASP.NET Core v2 - an Update
- 17/01 [NDC London] – Implementing authorization for web apps and APIs (aka PolicyServer announcement)
- 17/01 [DotNetRocks] – IdentityServer and PolicyServer on DotNetRocks

67.4 2017

- 14/09 [Microsoft Learning] – Introduction to IdentityServer for ASP.NET Core - Brock Allen
- 14/06 [NDC Oslo] – Implementing Authorization for Web Applications and APIs

- 22/02 [NDC Mini Copenhagen] – IdentityServer4: New & Improved for ASP.NET Core - Dominick Baier
- 02/02 [DotNetRocks] – IdentityServer4 on DotNetRocks
- 16/01 [NDC London] – IdentityServer4: New and Improved for ASP.NET Core
- 16/01 [NDC London] – Building JavaScript and mobile/native Clients for Token-based Architectures

67.5 2016

- The history of .NET identity and IdentityServer Channel9 interview
- Authentication & secure API access for native & mobile Applications - Dominick Baier
- ASP.NET Identity 3 - Brock Allen
- Introduction to IdentityServer3 - Brock Allen

67.6 2015

- Securing Web APIs – Patterns & Anti-Patterns - Dominick Baier
- Authentication and authorization in modern JavaScript web applications – how hard can it be? - Brock Allen

67.7 2014

- Unifying Authentication & Delegated API Access for Mobile, Web and the Desktop with OpenID Connect and OAuth 2 - Dominick Baier