<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.3</td>
<td>Key material</td>
<td>90</td>
</tr>
<tr>
<td>25.4</td>
<td>Operational data</td>
<td>90</td>
</tr>
<tr>
<td>25.5</td>
<td>ASP.NET Core data protection</td>
<td>90</td>
</tr>
<tr>
<td>26</td>
<td>Logging</td>
<td>91</td>
</tr>
<tr>
<td>26.1</td>
<td>Setup for Serilog</td>
<td>91</td>
</tr>
<tr>
<td>26.1.1</td>
<td>ASP.NET Core 2.0+</td>
<td>91</td>
</tr>
<tr>
<td>26.1.2</td>
<td>.NET Core 1.0, 1.1</td>
<td>92</td>
</tr>
<tr>
<td>27</td>
<td>Events</td>
<td>95</td>
</tr>
<tr>
<td>27.1</td>
<td>Emitting events</td>
<td>95</td>
</tr>
<tr>
<td>27.2</td>
<td>Custom sinks</td>
<td>96</td>
</tr>
<tr>
<td>27.3</td>
<td>Built-in events</td>
<td>96</td>
</tr>
<tr>
<td>27.4</td>
<td>Custom events</td>
<td>97</td>
</tr>
<tr>
<td>28</td>
<td>Cryptography, Keys and HTTPS</td>
<td>99</td>
</tr>
<tr>
<td>28.1</td>
<td>Token signing and validation</td>
<td>99</td>
</tr>
<tr>
<td>28.2</td>
<td>Signing key rollover</td>
<td>99</td>
</tr>
<tr>
<td>28.3</td>
<td>Data protection</td>
<td>100</td>
</tr>
<tr>
<td>28.4</td>
<td>HTTPS</td>
<td>100</td>
</tr>
<tr>
<td>29</td>
<td>Grant Types</td>
<td>101</td>
</tr>
<tr>
<td>29.1</td>
<td>Machine to Machine Communication</td>
<td>101</td>
</tr>
<tr>
<td>29.2</td>
<td>Interactive Clients</td>
<td>102</td>
</tr>
<tr>
<td>29.3</td>
<td>Interactive clients without browsers or with constrained input devices</td>
<td>103</td>
</tr>
<tr>
<td>29.4</td>
<td>Custom scenarios</td>
<td>103</td>
</tr>
<tr>
<td>30</td>
<td>Secrets</td>
<td>105</td>
</tr>
<tr>
<td>30.1</td>
<td>Creating a shared secret</td>
<td>105</td>
</tr>
<tr>
<td>30.2</td>
<td>Authentication using a shared secret</td>
<td>106</td>
</tr>
<tr>
<td>30.3</td>
<td>Beyond shared secrets</td>
<td>106</td>
</tr>
<tr>
<td>31</td>
<td>Extension Grants</td>
<td>109</td>
</tr>
<tr>
<td>31.1</td>
<td>Example: Simple delegation using an extension grant</td>
<td>110</td>
</tr>
<tr>
<td>32</td>
<td>Resource Owner Password Validation</td>
<td>113</td>
</tr>
<tr>
<td>33</td>
<td>Refresh Tokens</td>
<td>115</td>
</tr>
<tr>
<td>33.1</td>
<td>Additional client settings</td>
<td>115</td>
</tr>
<tr>
<td>34</td>
<td>Reference Tokens</td>
<td>117</td>
</tr>
<tr>
<td>35</td>
<td>Mutual TLS</td>
<td>119</td>
</tr>
<tr>
<td>35.1</td>
<td>Client authentication</td>
<td>119</td>
</tr>
<tr>
<td>35.1.1</td>
<td>Validating the X.509 client certificate in IdentityServer</td>
<td>119</td>
</tr>
<tr>
<td>35.1.2</td>
<td>Endpoints in IdentityServer for mutual TLS</td>
<td>119</td>
</tr>
<tr>
<td>35.1.3</td>
<td>Mutual TLS configuration in IdentityServer</td>
<td>120</td>
</tr>
<tr>
<td>35.1.4</td>
<td>Using a client certificate to authenticate to IdentityServer</td>
<td>121</td>
</tr>
<tr>
<td>35.2</td>
<td>Sender-constrained access tokens</td>
<td>122</td>
</tr>
<tr>
<td>35.2.1</td>
<td>Confirmation claim</td>
<td>122</td>
</tr>
<tr>
<td>35.2.2</td>
<td>Validating and accepting a client certificate in APIs</td>
<td>123</td>
</tr>
<tr>
<td>35.2.3</td>
<td>Introspection and the confirmation claim</td>
<td>125</td>
</tr>
<tr>
<td>36</td>
<td>Authorize Request Objects</td>
<td>127</td>
</tr>
<tr>
<td>36.1</td>
<td>Passing request JWTs by reference</td>
<td>128</td>
</tr>
<tr>
<td>36.2</td>
<td>Accessing the request object data</td>
<td>128</td>
</tr>
</tbody>
</table>
### 37 Custom Token Request Validation and Issuance 129

### 38 CORS 131
- 38.1 Client-based CORS Configuration ........................................ 131
- 38.2 Custom Cors Policy Service ........................................ 131
- 38.3 Mixing IdentityServer’s CORS policy with ASP.NET Core’s CORS policies ........................................ 132

### 39 Discovery 133
- 39.1 Extending discovery ........................................ 133

### 40 Adding more API Endpoints 135
- 40.1 Discovery ........................................ 136
- 40.2 Advanced ........................................ 136
- 40.3 Claims Transformation ........................................ 137

### 41 Adding new Protocols 139
- 41.1 Typical authentication workflow ........................................ 139
- 41.2 Useful IdentityServer services ........................................ 139

### 42 Tools 141

### 43 Discovery Endpoint 143

### 44 Authorize Endpoint 145

### 45 Token Endpoint 147
- 45.1 Example ........................................ 148

### 46 UserInfo Endpoint 149
- 46.1 Example ........................................ 149

### 47 Device Authorization Endpoint 151
- 47.1 Example ........................................ 151

### 48 Introspection Endpoint 153
- 48.1 Example ........................................ 153

### 49 Revocation Endpoint 155
- 49.1 Example ........................................ 155

### 50 End Session Endpoint 157
- 50.1 Parameters ........................................ 157
- 50.2 Example ........................................ 158

### 51 Identity Resource 159

### 52 API Resource 161
- 52.1 Scopes ........................................ 161
- 52.2 Convenience Constructor Behavior ........................................ 162
- 52.3 Defining API resources in appsettings.json ........................................ 163

### 53 Client 165
- 53.1 Basics ........................................ 165
- 53.2 Authentication/Logout ........................................ 166
- 53.3 Token ........................................ 166
- 53.4 Consent Screen ........................................ 167
- 53.5 Device flow ........................................ 167
54  GrantValidationResult 169

55  Profile Service 171
   55.1 IProfileService APIs ................................................. 171
   55.2 ProfileDataRequestContext ........................................... 171
   55.3 Requested scopes and claims mapping ................................ 172
   55.4 IsActiveContext ....................................................... 172

56  IdentityServer Interaction Service 173
   56.1 IIdentityServerInteractionService APIs ................................ 173
   56.2 AuthorizationRequest .................................................. 174
   56.3 ErrorMessage ............................................................ 174
   56.4 LogoutRequest ........................................................... 174
   56.5 ConsentResponse ......................................................... 174
   56.6 Consent ................................................................. 175

57  Device Flow Interaction Service 177
   57.1 IDeviceFlowInteractionService APIs .................................... 177
   57.2 DeviceFlowAuthorizationRequest ...................................... 177
   57.3 DeviceFlowInteractionResult ........................................... 177

58  IdentityServer Options 179
   58.1 Endpoints ............................................................... 179
   58.2 Discovery ............................................................... 179
   58.3 Authentication ......................................................... 180
   58.4 Events ................................................................. 180
   58.5 InputLengthRestrictions ................................................. 180
   58.6 UserInteraction ......................................................... 180
   58.7 Caching ............................................................... 181
   58.8 CORS ................................................................. 181
   58.9 CSP (Content Security Policy) ........................................ 181
   58.10 Device Flow ............................................................ 181
   58.11 Mutual TLS .............................................................. 182

59  Entity Framework Support 183
   59.1 Configuration Store support for Clients, Resources, and CORS settings ........................................ 183
   59.2 ConfigurationStoreOptions ............................................. 184
   59.3 Operational Store support for authorization grants, consents, and tokens (refresh and reference) ........ 184
   59.4 OperationalStoreOptions ............................................... 185
   59.5 Database creation and schema changes across different versions of IdentityServer .......................... 185

60  ASP.NET Identity Support 187

61  Training 189
   61.1 Identity & Access Control for modern Applications (using ASP.NET Core 2 and IdentityServer4) ............. 189
   61.2 PluralSight courses ....................................................... 189

62  Blog posts 191
   62.1 Team posts ................................................................ 191
   62.1.1 2019 ................................................................ 191
   62.1.2 2018 ................................................................ 191
   62.1.3 2017 ................................................................ 191
   62.2 What’s new posts .......................................................... 192
   62.3 Community posts .......................................................... 192
<table>
<thead>
<tr>
<th></th>
<th>Videos</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>2019</td>
<td>193</td>
</tr>
<tr>
<td>63.1</td>
<td>2019</td>
<td>193</td>
</tr>
<tr>
<td>63.2</td>
<td>2018</td>
<td>193</td>
</tr>
<tr>
<td>63.3</td>
<td>2017</td>
<td>193</td>
</tr>
<tr>
<td>63.4</td>
<td>2016</td>
<td>194</td>
</tr>
<tr>
<td>63.5</td>
<td>2015</td>
<td>194</td>
</tr>
<tr>
<td>63.6</td>
<td>2014</td>
<td>194</td>
</tr>
</tbody>
</table>
IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core.

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.
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.
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 (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 dev/interim builds to the GitHub package repository. Add the following feed if you want to give them a try:

https://github.com/orgs/IdentityServer/packages
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.
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.

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.
CHAPTER 9

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.:
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:

```bash
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):

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

**Note:** The protocol used in this Template is http and the port is set to 5000 when running on Kestrel or a random one on IISExpress. You can change that in the Properties\launchSettings.json file. However, all of the quickstart instructions will assume you use the default port on Kestrel as well as the http protocol, which is sufficient for local development. For production scenarios you should switch to 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.

For this add a new class to your project and call it Config - add the following code to it:

```csharp
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 though 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:

```csharp
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 - the template already does this for you:

```csharp
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 http://localhost:5000/.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.
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 http://localhost:5001 only. You can do this by editing the launch-
Settings.json file inside the Properties folder. Change the application URL setting to be:

"applicationUrl": "http://localhost:5001"

9.7.1 The controller

Add a new class called IdentityController:

```csharp
[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:

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

(continues on next page)
services.AddAuthentication("Bearer")
    .AddJwtBearer("Bearer", options =>
    {
        options.Authority = "http://localhost:5000";
        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 http://localhost:5001/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:

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:

```csharp
// discover endpoints from metadata
var client = new HttpClient();
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:

```csharp
// 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:

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

if (!response.IsSuccessStatusCode)
{
    Console.WriteLine(response.StatusCode);
}
else
{
    var content = await response.Content.ReadAsStringAsync();
}
```

(continues on next page)
The output should look like this:

```json
{
    "access_token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJmdW5zaXMiLCJleHRlcnMiOjE2LCJpYXQiOjE2NjMzMTk4MzIsImlhdCI6MTY2NzE5Njk2MCwiZXhwIjoxNjY4NTU3ODE5fQ.5t5JF9p9nW3v1sJmdfoTq9HxvGJ4qo73KJGccetmLry",
    "token_type": "Bearer"
}
```

```json

```

**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
Interactive Applications with ASP.NET Core

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
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. 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:

```csharp
JwtSecurityTokenHandler.DefaultMapInboundClaims = false;

services.AddAuthentication(options =>
{
    options.DefaultScheme = "Cookies";
    options.DefaultChallengeScheme = "oidc";
})
.AddCookie("Cookies")
.AddOpenIdConnect("oidc", options =>
{
    options.Authority = "http://localhost:5000";
    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:

```csharp
@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 ammending the `Ids` property in `Config.cs`:

```csharp
public static IEnumerable<IdentityResource> Ids =>
    new List<IdentityResource>
```
Register the identity resources with IdentityServer in startup.cs:

```csharp
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:

```csharp
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:

```csharp
public static IEnumerable<Client> Clients =>
    new List<Client>
    {
        // machine to machine client (from quickstart 1)
        new Client
        {
            ClientId = "client",
            ClientSecrets = { new Secret("secret".Sha256()) },
            AllowedGrantTypes = GrantTypes.ClientCredentials,
            AllowedScopes = { "api1" }
        },
```

(continues on next page)
```csharp
// 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
    RedirectUris = { "http://localhost:5002/signin-oidc" },

    // where to redirect to after logout
    PostLogoutRedirectUris = { "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.
As you can see, the cookie has two parts, the claims of the user, and some metadata. This metadata also contains the original token that was issued by IdentityServer. Feel free to copy this token to jwt.ms to inspect its content.

### 10.7 Adding sign-out

The very last step is to add sign-out to the MVC client.

With an authentication service like IdentityServer, it is not enough to clear the local application cookies. In addition you also need to make a roundtrip to IdentityServer to clear the central single sign-on session.
The exact protocol steps are implemented inside the OpenID Connect handler, simply add the following code to some controller to trigger the sign-out:

```csharp
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. `http://localhost:5000/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`:

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

(continues on next page)
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:

![Login Page Screenshot]

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

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

```csharp
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.
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:

```csharp
new Client
{
    ClientId = "mvc",
    ClientSecrets = { new Secret("secret".Sha256()) },
    AllowedGrantTypes = GrantTypes.Code,
    RequireConsent = false,
    RequirePkce = true,

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

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

    AllowedScopes = new List<string>
    {
```
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:

```csharp
services.AddAuthentication(options =>
{
    options.DefaultScheme = "Cookies";
    options.DefaultChallengeScheme = "oidc";
})
    .AddCookie("Cookies")
    .AddOpenIdConnect("oidc", options =>
    {
        options.Authority = "http://localhost:5000";
        options.RequireHttpsMetadata = false;
        options.ClientId = "mvc";
        options.ClientSecret = "secret";
        options.ResponseType = "code";
        options.SaveTokens = true;
        options.Scope.Add("api1");
        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:

```csharp
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`:

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

### 12.1 IdentityServer4.EntityFramework

IdentityServer4.EntityFramework implements the required stores and services using the following DbContexts:

- 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`
12.2 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
```

12.3 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.

12.4 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:

```csharp
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:

```csharp
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.

### 12.5 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:

```bash
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:

```bash
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.

### 12.6 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:

```csharp
private void InitializeDatabase(IApplicationBuilder app)
{
    using (var serviceScope = app.ApplicationServices.GetService<IServiceScopeFactory>())
    {
        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)
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.

12.7 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 12. Using EntityFramework Core for configuration and operational data
IdentityServer is a combination of middleware and services. All configuration is done in your startup class.

### 13.1 Configuring services

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

```csharp
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.

### 13.2 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 service 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.
13.3 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.

13.4 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.

13.5 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 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.

### 13.6 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.

### 13.7 Configuring the pipeline

You need to add IdentityServer to the pipeline by calling:

```csharp
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 IdentityServer before the UI framework that implements the login screen.
The first thing you will typically define in your system are the resources that you want to protect. That could be identity information of your users, like profile data or email addresses, or access to APIs.

**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.

### 14.1 Defining identity resources

Identity resources are data like user ID, name, or email address of a user. An identity resource has a unique name, and you can assign arbitrary claim types to it. These claims will then be included in the identity token for the user. The client will use the **scope** parameter to request access to an identity resource.

The OpenID Connect specification specifies a couple of **standard** identity resources. The minimum requirement is, that you provide support for emitting a unique ID for your users - also called the subject id. This is done by exposing the standard identity resource called **openid**:

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

The **IdentityResources** class supports all scopes defined in the specification (openid, email, profile, telephone, and address). If you want to support them all, you can add them to your list of supported identity resources:

```csharp
public static IEnumerable<IdentityResource> GetIdentityResources()
{
    // (continues on next page)
}
```
14.2 Defining custom identity resources

You can also define custom identity resources. Create a new IdentityResource class, give it a name and optionally a display name and description and define which user claims should be included in the identity token when this resource gets requested:

```csharp
public static IEnumerable<IdentityResource> GetIdentityResources()
{
    var customProfile = new IdentityResource(
        name: "custom.profile",
        displayName: "Custom profile",
        claimTypes: new[] { "name", "email", "status" });

    return new List<IdentityResource>
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Email(),
        new IdentityResources.Profile(),
        new IdentityResources.Phone(),
        new IdentityResources.Address()
        customProfile
    };
}
```

See the reference section for more information on identity resource settings.

14.3 Defining API resources

To allow clients to request access tokens for APIs, you need to define API resources, e.g.:

To get access tokens for APIs, you also need to register them as a scope. This time the scope type is of type Resource:

```csharp
public static IEnumerable<ApiResource> GetApis()
{
    return new[]
    {
        // simple API with a single scope (in this case the scope name is the same as
        // the api name)
        new ApiResource("api1", "Some API 1"),

        // expanded version if more control is needed
        new ApiResource
        {
            Name = "api2",
```
// secret for using introspection endpoint
ApiSecrets =
{
    new Secret("secret".Sha256())
},

// include the following using claims in access token (in addition to subject id)
UserClaims = { JwtClaimTypes.Name, JwtClaimTypes.Email },

// this API defines two scopes
Scopes =
{
    new Scope()
    {
        Name = "api2.full_access",
        DisplayName = "Full access to API 2",
    },
    new Scope
    {
        Name = "api2.read_only",
        DisplayName = "Read only access to API 2"
    }
};

See the reference section for more information on API resource settings.

**Note:** The user claims defined by resources are used to tell the IProfileService extensibility point which claims to include in tokens.
Clients represent applications that can request tokens from your identity server.

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.

### 15.1 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):

```csharp
public class Clients
{
    public static IEnumerable<Client> Get()
    {
        return new List<Client>
        {
            new Client
            {
                ClientId = "service.client",
```
15.2 Defining browser-based JavaScript client (e.g. SPA) for user authentication and delegated access and API

This client uses the so called implicit flow to request an identity and access token from JavaScript:

```csharp
var jsClient = new Client
{
    ClientId = "js",
    ClientName = "JavaScript Client",
    ClientUri = "http://identityserver.io",
    AllowedGrantTypes = GrantTypes.Implicit,
    AllowAccessTokensViaBrowser = true,
    RedirectUris = { "http://localhost:7017/index.html" },
    PostLogoutRedirectUris = { "http://localhost:7017/index.html" },
    AllowedCorsOrigins = { "http://localhost:7017" },
    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        IdentityServerConstants.StandardScopes.Email,
        "api1", "api2.read_only"
    }
};
```

15.3 Defining a server-side web application (e.g. MVC) for use authentication and delegated API access

Interactive server side (or native desktop/mobile) applications use the hybrid 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):

```csharp
var mvcClient = new Client
{
    ClientId = "mvc",
    ClientName = "MVC Client",
    ClientUri = "http://identityserver.io",
    AllowedGrantTypes = GrantTypes.Hybrid,
    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        IdentityServerConstants.StandardScopes.Email,
        "api1", "api2.read_only"
    }
};
```
AllowOfflineAccess = true,
ClientSecrets = { new Secret("secret").Sha256() },

RedirectUris = { "http://localhost:21402/signin-oidc" },
PostLogoutRedirectUris = { "http://localhost:21402/" },
FrontChannelLogoutUri = "http://localhost:21402/signout-oidc",

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

    "api1", "api2.read_only"
};

15.4 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": [
        {
            "Enabled": true,
            "ClientId": "local-dev",
            "ClientName": "Local Development",
            "ClientSecrets": [ { "Value": "<Insert Sha256 hash of the secret encoded as Base64 string>" } ],
            "AllowedGrantTypes": [ "implicit" ],
            "AllowedScopes": [ "openid", "profile" ],
            "RedirectUris": [ "https://localhost:5001/signin-oidc" ],
            "RequireConsent": false
        }
    ]
}

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.

### 16.1 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.

### 16.2 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:

```csharp
services.AddIdentityServer()
    .AddInMemoryClients(Clients.Get())
    .AddInMemoryIdentityResources(Resource.GetIdentityResources())
    .AddInMemoryApiResources(Resource.GetApiResources())
    .AddDeveloperSigningCredential()
    .AddTestUsers(TestUsers.Users);

services.AddAuthentication("MyCookie")
    .AddCookie("MyCookie", options =>
```

(continues on next 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`.

### 16.3 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.

### 16.4 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.

[Diagram showing the login workflow]

1) `https://server/authorize?...

2) `/login?returnUrl=/authorize?...`
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.

16.5 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.

16.6 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.
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.

### 17.1 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.

### 17.2 The role of cookies

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

```csharp
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:

```csharp
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:

```csharp
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.

### 17.3 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.:

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

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

return Challenge(provider, props);
```
17.4 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:

```csharp
// 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("~/");

17.5 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:

```csharp
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 =>
    {
      // ...
    })
```
If only particular schemes are to be configured, then pass those schemes as parameters:

```csharp
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 =>
    {
        // ...
    });
}
```
Chapter 17. Sign-in with External Identity Providers
On supported platforms, you can use IdentityServer to authenticate users using Windows authentication (e.g. against Active Directory). Currently Windows authentication is available when you host IdentityServer using:

- Kestrel on Windows using IIS and the IIS integration package
- HTTP.sys server on Windows

In both cases, Windows authentication is triggered by using the `ChallengeAsync` API on the `HttpContext` using the scheme "Windows". The account controller in our quickstart UI implements the necessary logic.

### 18.1 Using Kestrel

When using Kestrel, you must run “behind” IIS and use the IIS integration:

```csharp
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://localhost:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

Kestrel is automatically configured when using the `WebHost.CreateDefaultBuilder` approach for setting up the `WebHostBuilder`.

Also the virtual directory in IIS (or IIS Express) must have Windows and anonymous authentication enabled.

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 this automatic behavior. This is done in `ConfigureServices`:

```csharp
services.Configure<IISOptions>(iis =>
{

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

**Note:** By default, the display name is empty, and the Windows authentication button will not show up in the quickstart UI. You need to set a display name if you rely on automatic discovery of external providers.
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.

### 19.1 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):

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

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

```csharp
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.

### 19.2 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.

### 19.3 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 `ShowSignoutPrompt` 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.
CHAPTER 20

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-logon redirect. The post-logon 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:

```csharp
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Logout(LogoutInputModel 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();
    }

    await HttpContext.RedirectAsync(model.RedirectUri);
}
```

(continues on next 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 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.
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.

### 22.1 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.

### 23.1 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.

### 23.2 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.
23.3 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.

23.4 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 24

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:

```csharp
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, ILogggerFactory loggerFactory)
    {
        app.UseAuthentication();
    }
}
```

(continues on next page)
24.1 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:

```csharp
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.

24.2 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.:
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:

```csharp
.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`).

### 24.3 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:

```csharp
services
    .AddMvcCore(options =>
    {
        // require scope1 or scope2
        var policy = ScopePolicy.Create("scope1", "scope2");
        options.Filters.Add(new AuthorizeFilter(policy));
    });
.AddJsonFormatters()
.AddAuthorization();
```

#### Composing a scope policy:

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

(continues on next 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).

### 25.1 Typical architecture

Typically you will design your IdentityServer deployment for high availability:

![Diagram of typical architecture](#)
IdentityServer itself is stateless and does not require server affinity - but there is data that needs to be shared between the instances.

### 25.2 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`.

### 25.3 Key material

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

### 25.4 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.

### 25.5 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.
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...

### 26.1 Setup for Serilog

We personally like Serilog a lot. Give it a try.

#### 26.1.1 ASP.NET Core 2.0+

For the following configuration you need the Serilog.AspNetCore and Serilog.Sinks.Console packages:
public class Program
{
    public static void Main(string[] args)
    {
        Console.Title = "IdentityServer4";

        Log.Logger = new LoggerConfiguration()
            .MinimumLevel.Debug()
            .Enrich.FromLogContext()
            .CreateLogger();

        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args)
    {
        return WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .UseSerilog()
            .Build();
    }
}

26.1.2 .NET Core 1.0, 1.1

For the following configuration you need the Serilog.Extensions.Logging and Serilog.Sinks.Console packages:

public class Program
{
    public static void Main(string[] args)
    {
        Console.Title = "IdentityServer4";

        Log.Logger = new LoggerConfiguration()
            .MinimumLevel.Debug()
            .Enrich.FromLogContext()
            .CreateLogger();

        BuildWebHost(args).Run();
    }
}
public static IWebHost BuildWebHost(string[] args)
{
    return WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging(builder =>
        {
            builder.ClearProviders();
            builder.AddSerilog();
        })
        .Build();
}
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.

27.1 Emitting events

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

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

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

```csharp
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 credentials"));
    }
}
```

(continues on next page)
27.2 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:

```csharp
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.

27.3 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.

## 27.4 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:

```csharp
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; }
}
```
IdentityServer relies on a couple of crypto mechanisms to do its job.

### 28.1 Token signing and validation

IdentityServer needs an asymmetric key pair to sign and validate JWTs. This keypair can be a certificate/private key combination or raw RSA keys. In any case it must support RSA with SHA256.

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.

### 28.2 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.

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 AddValidationKeys 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.

### 28.3 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.

### 28.4 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:

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

You can also specify the grant types list manually:

```csharp
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.

### 29.1 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.
29.2 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 substition 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:
```csharp
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
};
```

### 29.3 Interactive clients without browsers or with constrained input devices

This grant type is detailed [RFC 8628](https://tools.ietf.org/html/rfc8628).

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.

### 29.4 Custom scenarios

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

### 30.1 Creating a shared secret

The following code sets up a hashed shared secret:

```csharp
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:

```csharp
var client = new Client
{
    ClientId = "client",
    ClientSecrets = new List<Secret> { secret },
    AllowedGrantTypes = GrantTypes.ClientCredentials,
    AllowedScopes = new List<string>
    {
        "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)};

30.2 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.

30.3 Beyond shared secrets

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).

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>()
Our default private key JWT secret validator expects the full (leaf) certificate as base64 on the secret definition. This certificate will then be used to validate the signature on the self-signed JWT, e.g.:

```csharp
var client = new Client
{
    ClientId = "client.jwt",
    ClientSecrets =
    {
        new Secret
        {
            Type = IdentityServerConstants.SecretTypes.X509CertificateBase64,
            Value = ...
        }
    },
    AllowedGrantTypes = GrantTypes.ClientCredentials,
    AllowedScopes = { "api1", "api2" }
};
```

You could implement your own secret validator (or extend ours) to implement e.g. chain trust validation instead.
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:

```csharp
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:

```csharp
builder.AddExtensionGrantValidator<MyExtensionsGrantValidator>();
```
31.1 Example: Simple delegation using an extension grant

Imagine the following scenario - a front end client calls a middle tier API 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:

```csharp
public class DelegationGrantValidator : IExtensionGrantValidator
{
}
```
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.:

```csharp
var client = new client
{
    ClientId = "api1.client",
    ClientSecrets = new List<Secret>
    {
        new Secret("secret".Sha256()),
    },
    AllowedGrantTypes = { "delegation" },
    AllowedScopes = new List<string>
    {
        "api2"
    }
};
```

31.1. Example: Simple delegation using an extension grant 111
Calling the token endpoint

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

```csharp
public async Task<TokenResponse> DelegateAsync(string userToken)
{
    var payload = new {
        token = userToken
    };

    // create token client
    var client = new TokenClient(disco.TokenEndpoint, "api1.client", "secret");

    // send custom grant to token endpoint, return response
    return await client.RequestCustomGrantAsync("delegation", "api2", payload);
}
```

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:

```csharp
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.
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 clients needs to be explicitly authorized to request refresh tokens by setting `AllowOfflineAccess` to `true`.

### 33.1 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.
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:

```csharp
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`:

```csharp
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.
Mutual TLS support in IdentityServer allows for two features:

- Client authentication to endpoints within IdentityServer using a X.509 client certificate
- Use of sender-constrained access tokens from a client to APIs using a X.509 client certificate certificate

Note: See this spec for more information

35.1 Client authentication

Clients can use a X.509 client certificate as an authentication mechanism to endpoints in IdentityServer.

35.1.1 Validating the X.509 client certificate in IdentityServer

It’s the hosting layer’s responsibility to do the actual validation of the client certificate, and then IdentityServer uses (and trusts) this result as part of the client authentication and validation at the application level. This means to use this feature within IdentityServer you must first configure your web server (IIS, Kestrel, etc.) to accept and validate client certificates.

Consult your web server documentation to learn how to do this.

Note: mkcert is a nice utility for creating certificates for development purposes.

35.1.2 Endpoints in IdentityServer for mutual TLS

Given that mutual TLS is performed at the TLS channel, it tends to be more convenient to configure distinct endpoints where mutual TLS is expected and/or required. This allows the existing endpoints in IdentityServer to operate normally
without mutual TLS. In IdentityServer, the mutual TLS endpoints are expected to be located beneath the path `~/connect/mtls`. This means your web server can be configured to require mutual TLS for all requests at and below that path.

As such, IdentityServer’s discovery document reflects those paths:

```json
{
  issuer: http://localhost:5000,
  jwks_uri: http://localhost:5000/well-known/openid-configuration/jwks,
  authorization_endpoint: http://localhost:5000/connect/authorize,
  token_endpoint: http://localhost:5000/connect/token,
  userinfo_endpoint: http://localhost:5000/connect/userinfo,
  end_session_endpoint: http://localhost:5000/connect/end_session,
  check_session_iframe: http://localhost:5000/connect/check_session,
  revocation_endpoint: http://localhost:5000/connect/revocation,
  introspection_endpoint: http://localhost:5000/connect/introspect,
  device_authorization_endpoint: http://localhost:5000/connect/device_authorization,
  mtls_endpoints_aliases: {
    token_endpoint: http://localhost:5000/connect/mtls/token,
    revocation_endpoint: http://localhost:5000/connect/mtls/revocation,
    introspection_endpoint: http://localhost:5000/connect/mtls/introspect,
    device_authorization_endpoint: http://localhost:5000/connect/mtls/device_authorization,
  },
  frontchannel_logout_supported: true,
  frontchannel_logout_session_supported: true,
  backchannel_logout_supported: true,
  backchannel_logout_session_supported: true,
  scopes_supported: [
    "openid",
    "profile",
    "email",
    "custom.profile",
    "IdentityServerApi",
    "api1",
    "api2.full_access",
    "api2.read_only",
    "offline_access",
  ],
  claims_supported: [
    "sub",
    "name",
    "family_name",
    "given_name",
    "middle_name",
    "nickname",
    "preferred_username",
    "profile",
    "picture",
    "website",
    "gender",
    "birthdate",
    "zoneinfo",
    "locale",
    "updated_at",
    "time_zone",
  ],
}
```

### 35.1.3 Mutual TLS configuration in IdentityServer

IdentityServer requires additional configuration to utilize the result of the mutual TLS authentication.

First, there must be an authentication handler configured in the hosting application that will surface the result of the client certificate authentication in the web server. There will be one built into ASP.NET Core 3.0, but prior to that the idunno.Authentication.Certificate handler is recommended. For example:

```csharp
services.AddAuthentication()
    .AddCertificate("x509", options =>
    {
        options.Events = new CertificateAuthenticationEvents
        {
            OnValidateCertificate = context =>
```

(continues on next page)
Next, in the IdentityServer options, enable mutual TLS and configure the scheme of the authentication handler registered in the previous step.

For example:

```csharp
var builder = services.AddIdentityServer(options =>
{
    options.MutualTls.Enabled = true;
    options.MutualTls.ClientCertificateAuthenticationScheme = "x509";
});
```

Next, on the IdentityServer builder add the services to DI which will allow for validating the client certificate by calling AddMutualTlsSecretValidators:

```csharp
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:

```csharp
new Client
{
    ClientId = "mTLS",
    AllowedGrantTypes = GrantTypes.ClientCredentials,
    AllowedScopes = { "api1" }
    ClientSecrets = {
        new Secret(@"CN=mtls.test, OU=ROO\ballen@roo, O=mkcert development certificate --", "mtls.test")
        {
            Type = SecretTypes.X509CertificateName
        },
        // or:
        //new Secret("bca0d040847f843c5ee0fa6eb494837470155868", "mtls.test")
        //{
        //    Type = SecretTypes.X509CertificateThumbprint
        //},
    }
}
}
```

### 35.1.4 Using a client certificate to authenticate to IdentityServer

When writing a client to connect to IdentityServer, the HttpClientHandler class provides a convenient mechanism to use a client certificate by adding to the ClientCertificates collection. The HttpClientHandler
can then be used as the message handler in HttpClient. 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:

```csharp
static async Task<TokenResponse> RequestTokenAsync()
{
    var handler = new HttpClientHandler();
    var cert = X509.CurrentUser.My.Thumbprint.Find("bca0d040847f843c5ee0fa6eb494837470155868").Single();
    handler.ClientCertificates.Add(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(),
        ClientId = "mtls",
        Scope = "api1"
    });
    if (response.IsError) throw new Exception(response.Error);
    return response;
}
```

## 35.2 Sender-constrained access tokens

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.

### 35.2.1 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:
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.

### 35.2.2 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:

```csharp
services.AddAuthentication("token")
    .AddIdentityServerAuthentication("token", options =>
    {
        options.Authority = Constants.Authority;
        options.RequireHttpsMetadata = false;

        options.ApiName = "api1";
        options.ApiSecret = "secret";
    })
    .AddCertificate("x509", options =>
    {

        options.Events = new CertificateAuthenticationEvents
        {
            OnValidateCertificate = context =>
            {
                context.Principal = Principal.CreateFromCertificate(context.ClientCertificate, includeAllClaims: true);
            }
        };
    })
```

(continues on next page)
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 an example implemented in middleware:

```csharp
app.UseAuthentication();
app.Use(async (ctxt, next) =>
{
    if (ctxt.User.Identity.IsAuthenticated)
    {
        var cnfJson = ctxt.User.FindFirst("cnf")?.Value;
        if (!String.IsNullOrWhiteSpace(cnfJson))
        {
            var certResult = await ctxt.AuthenticateAsync("x509");
            if (!certResult.Succeeded)
            {
                await ctxt.ChallengeAsync("x509");
                return;
            }
            var cert = ctxt.Connection.ClientCertificate;
            if (cert == null)
            {
                await ctxt.ChallengeAsync("x509");
                return;
            }
            var thumbprint = cert.Thumbprint;
            var cnf = JObject.Parse(cnfJson);
            var sha256 = cnf.Value<string>("x5t#S256");
            if (String.IsNullOrEmpty(sha256) || !thumbprint.Equals(sha256, StringComparison.OrdinalIgnoreCase))
            {
                await ctxt.ChallengeAsync("token");
                return;
            }
        }
    }
    await next();
});
app.UseMvc();
```

Once the above middleware succeeds, then the caller has been authenticated with a sender-constrained access token.
35.2.3 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.
Authorize Request Objects

Instead of providing all parameters for an authorize request as individual query string parameters, you can package them up in signed JWTs. 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, symmetric and RSA keys. For symmetric and RSA you need to add a JWK secret to the corresponding client, e.g.:

```csharp
private readonly string _symmetricJwk = @"{ 'kty': 'oct', 'use': 'sig', 'kid': '1', 'k' :
 'nYA-If8xTsdBHe9hunvizcp3dt7f6qGquad18kZHntvqEgjJ9uD-9x3kbQ-
LYfLHS3xmM2MpFQFg1JzT_0U_F8DI4oby4TvBdGaSzP664UgA8_
5Gb7f1nrlsapl1LiItVnpgQX31pyTvCzVvQ-
UVsxbDaaS6Us1nw7SE4Im8Ye2WY2rdCCXL8yAX9vIR7v77yvNTEcBcI6y4J1v2aqMB4YKVSfygs8XqGCHjpE5bvI-
A4ESbUXC6cVFvCeDg9pR6HK78mPM1096xrgtKZcXEJtUEIYPys6-rbwAIdmxJxKxpgRpt0Frv_
9fm6YpwG7QivYBXvRwaodL1TA', 'alg': 'HS256'}";

var client = new Client
{
    ClientId = "foo",
    ClientSecrets =
    {
        new Secret
        {
            Type = IdentityServerConstants.SecretTypes.JsonWebKey,
            Value = _symmetricJwk
        }
    }
};
```

**Note:** Microsoft.IdentityModel.Tokens.JsonWebKeyConverter has various helpers to convert keys to JWKs

Because of a bug in Microsoft’s JWT library, X509 certificates cannot be formatted as JWKs right now. You can use the base64 representation instead:
ClientSecrets =
{
    new Secret
    {
        Type = IdentityServerConstants.SecretTypes.X509CertificateBase64,
        Value = Convert.ToBase64String(cert.Export(X509ContentType.Cert))
    }
}

36.1 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):

```csharp
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.

36.2 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:

```csharp
/// <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.

### 38.1 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` 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).

### 38.2 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`:

```csharp
var cors = new DefaultCorsPolicyService(_loggerFactory.CreateLogger<DefaultCorsPolicyService>())
{
    AllowedOrigins = { "https://foo", "https://bar" }
};
services.AddSingleton<ICorsPolicyService>(cors);
```

Note: Use `AllowAll` with caution.

### 38.3 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`).
CHAPTER 39

Discovery

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.:

```csharp
services.AddIdentityServer(options =>
{
    options.Discovery.ShowIdentityScopes = false;
    options.Discovery.ShowApiScopes = false;
    options.Discovery.ShowClaims = false;
    options.Discovery.ShowExtensionGrantTypes = false;
});
```

39.1 Extending discovery

You can add custom entries to the discovery document, e.g.

```csharp
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.:

```csharp
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.:

```csharp
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:

```csharp
services.AddLocalApiAuthentication();
```

To protect an API controller, decorate it with an `Authorize` attribute using the `LocalApi.PolicyName` policy:
Authorized clients can then request a token for the `IdentityServerApi` scope and use it to call the API.

### 40.1 Discovery

You can also add your endpoints to the discovery document if you want, e.g. like this:

```csharp
services.AddIdentityServer(options =>
{
   options.Discovery.CustomEntries.Add("local_api", "~/localapi");
});
```

### 40.2 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.:

  ```csharp
  services.AddAuthorization(options =>
  {
      options.AddPolicy(IdentityServerConstants.LocalApi.PolicyName, policy =>
      {
          policy.AddAuthenticationSchemes(IdentityServerConstants.LocalApi.AuthenticationScheme);
          policy.RequireAuthenticatedUser();
          // custom requirements
      });
  });
  ```
40.3 Claims Transformation

You can provide a callback to transform the claims of the incoming token after validation. Either use the helper method, e.g.:

```csharp
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.

41.1 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)

41.2 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:

```csharp
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:

```csharp
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`:

```csharp
public interface IReturnUrlParser
{
    bool IsValidReturnUrl(string returnUrl);
    Task<AuthorizationRequest> ParseAsync(string returnUrl);
}
```

..and then register the parser in DI:

```csharp
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:

```csharp
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.RedirectUris.First(),
    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:

```csharp
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):

```csharp
public async Task<IActionResult> MyAction()
{
    var token = await _tools.IssueClientJwtAsync(
        clientId: "client_id",
        lifetime: 3600,
        audiences: new[] { "backend.api" });

    // more code
}
```
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 45

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)

45.1 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.
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).

### 46.1 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.
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.

### 47.1 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.
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.

### 48.1 Example

```plaintext
POST /connect/introspect
Authorization: Basic xxxxyyy
token=<token>
```

A successful response will return a status code of 200 and either an active or inactive token:

```json
{
   "active": true,
   "sub": "123"
}
```

Unknown or expired tokens will be marked as inactive:

```json
{
   "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.
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)

### 49.1 Example

```plaintext
POST /connect/revocation HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

token=45ghiukldjahndhzdauz&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.

50.1 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.
50.2 Example

```
GET /connect/endsession?id_token=_
  → hint=eyJhbGciOiJSUzI1NiIsImtpZCI6Ijd1OGFkZmMjU1OTY4NWZmYi02NThjNDEyIiwidHlwIjoiSlidUI0.
  → eyJuYmYiOiJ0TE3NjUzMjEsmVcCI6MTQ5MTCt2NTyyMSwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo1MDAwIiwiYXVkIjoianbSI
  → STzOWweVYMt2dRAeRT95cMYEmC1lixWkmGwVH2Yyiks9BEtotbS2iSfgE5kRh72kgH78N3-
  → RgCTUmM2edB3bZx4H5ut3wWSb2tQ2JLfhTwJAjaLE9ykt68ovNJySbm8hj2hHzPWh55jzshivQvTX0Gdt1bdDoEAloNONxKp
  → rAALhFPkyKnvC-uB8IhtGNSyRWLFhvQAdS3fRNO7iIs5hYRxwFSU7a52uUqz6RRi-bcDi-
  → djkO5uAwihf6bYcaY_TxXWoCmq8N8uAw9zqFSQucXymfOAi2UF3eFZt02hBu-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`.  

---
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.
API Resource

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.

**UserClaims** List of associated user claim types that should be included in the access token.

**Scopes** An API must have at least one scope. Each scope can have different settings.

### 52.1 Scopes

In the simple case an API has exactly one scope. But there are cases where you might want to sub-divide the functionality of an API, and give different clients access to different parts.

**Name** The unique name of the scope. This is the value a client will use for the scope parameter in the authorize/token request.

**DisplayName** This value can be used e.g. on the consent screen.

**Description** This value can 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 access token. The claims specified here will be added to the list of claims specified for the API.

52.2 Convenience Constructor Behavior

Just a note about the constructors provided for the ApiResource class.

For full control over the data in the ApiResource, use the default constructor with no parameters. You would use this approach if you wanted to configure multiple scopes per API. For example:

```csharp
new ApiResource
{
    Name = "api2",
    Scopes =
    {
        new Scope()
        {
            Name = "api2.full_access",
            DisplayName = "Full access to API 2"
        },
        new Scope
        {
            Name = "api2.read_only",
            DisplayName = "Read only access to API 2"
        }
    }
}
```

For simpler scenarios where you only require one scope per API, then several convenience constructors which accept a name are provided. For example:

```csharp
new ApiResource("api1", "Some API 1")
```

Using the convenience constructor is equivalent to this:

```csharp
new ApiResource
{
    Name = "api1",
    DisplayName = "Some API 1",
    Scopes =
    {
        new Scope()
        {
            Name = "api1",
            DisplayName = "Some API 1"
        }
    }
}
```
52.3 Defining API resources in appsettings.json

The `AddInMemoryApiResource` 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:

```json
"IdentityServer": {
    "IssuerUri": "urn:sso.company.com",
    "ApiResources": [
        {
            "Name": "api1",
            "DisplayName": "My API",
            "Scopes": [
                {
                    "Name": "api1",
                    "DisplayName": "My API"
                }
            ]
        }
    ]
}
```

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

```csharp
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.

### 53.1 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.  
  *(defaults to `true`)*
- **RequireClientSecret** Specifies whether this client needs a secret to request tokens from the token endpoint  
  *(defaults to `true`)*
- **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
- **AllowPlainTextPkce** Specifies whether clients using PKCE can use a plain text code challenge (not recommended - and default to `false`)
- **RedirectUris** 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.
53.2 Authentication/Logout

**PostLogoutRedirectUris** 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.

53.3 Token

**IdentityTokenLifetime** Lifetime to identity token in seconds (defaults to 300 seconds / 5 minutes)

**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).

**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.

### 53.4 Consent Screen

**RequireConsent** Specifies whether a consent screen is required. Defaults to true.

**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)

### 53.5 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)
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):

```csharp
context.Result = new GrantValidationResult(
    subject: "818727",
    authenticationMethod: "custom",
    claims: optionalClaims);
```

...or using an error and description (failure case):

```csharp
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.
CHAPTER 55

Profile Service

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.

55.1 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.

55.2 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.

### 55.3 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.

### 55.4 IsActiveContext

Models the request to determine is 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.
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.

### 56.1 `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.


56.2 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.

56.3 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.

56.4 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.

56.5 ConsentResponse

ScopesConsented The collection of scopes the user consented to.
RememberConsent Flag indicating if the user’s consent is to be persisted.
56.6 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.
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.

### 57.1 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`.

### 57.2 DeviceFlowAuthorizationRequest

- **ClientId** The client identifier that initiated the request.
- **ScopesRequested** The scopes requested from the authorization request.

### 57.3 DeviceFlowInteractionResult

- **IsError** Specifies if the authorization request errored.
- **ErrorMessage** Error description upon failure.
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.

- **PublicOrigin** The origin of this server instance, e.g. https://myorigin.com. If not set, the origin name is inferred from the request.

- **AccessTokenJwtType** Specifies the value used for the JWT typ header for access tokens (defaults to at+jwt).

- **EmitLegacyResourceAudienceClaim** Emits an aud claim with the format issuer/resources. That’s needed for some older access token validation plumbing. Defaults to false.

### 58.1 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.

### 58.2 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.
58.3 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).

- **RequireAuthenticatedUserForSignOutMessage** Indicates if the 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 emitted on the end session callback endpoint which renders iframes to clients for front-channel signout notification. Defaults to true.

58.4 Events

Allows configuring if and which events should be submitted to a registered event sink. See [here](#) for more information on events.

58.5 InputLengthRestrictions

Allows setting length restrictions on various protocol parameters like client id, scope, redirect URI etc.

58.6 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.

### 58.7 Caching

These setting 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.

### 58.8 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.

### 58.9 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 `CspLevel.One` 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.

### 58.10 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.
58.11 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".
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 repo for this library is located 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.

### 59.1 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:

```csharp
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.AddAssembly(migrationsAssembly);
    services.AddIdentityServer()
        // this adds the config data from DB (clients, resources, CORS)
        .AddConfigurationStore(options =>
        {
            options.ConfigureDbContext = builder =>
```
To configure the configuration store, use the `ConfigurationStoreOptions` object passed to the configuration callback.

### 59.2 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`.

```csharp
options.DefaultSchema = "myConfigurationSchema";
```

If you need to change the schema for the Migration History Table, you can chain another action to the `UseSqlServer`:

```csharp
options.ConfigureDbContext = b =>
    b.UseSqlServer(connectionString,
    sql => sql.MigrationsAssembly(migrationsAssembly).
    MigrationsHistoryTable("MyConfigurationMigrationTable", "myConfigurationSchema"));
```

### 59.3 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 an 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`:

```csharp
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()
        .AddOperationalStore(options =>
        {
            // this adds the operational data from DB (codes, tokens, consents)
            .AddOperationalStore(options =>
            {
```

(continues on next page)
options.ConfigureDbContext = builder =>
    builder.UseSqlServer(connectionString,
    sql => sql.MigrationsAssembly(migrationsAssembly));

    // this enables automatic token cleanup. this is optional.
    options.EnableTokenCleanup = true;
    options.TokenCleanupInterval = 30; // interval in seconds

To configure the operational store, use the OperationalStoreOptions options object passed to the configuration callback.

### 59.4 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).

### 59.5 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.
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`:

```csharp
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).
Here are some online, remote and classroom training options to learn more about ASP.NET Core identity & IdentityServer4.

### 61.1 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.

### 61.2 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 OAuth 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
190 Chapter 61. Training

- OAuth2 and OpenID Connect Strategies for Angular and ASP.NET
62.1 Team posts

62.1.1 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

62.1.2 2018

- IdentityServer4 Update
- IdentityServer and Swagger
- https://www.scottbrady91.com/OAuth/Removing-Shared-Secrets-for-OAuth-Client-Authentication
- Creating Your Own IdentityServer4 Storage Library

62.1.3 2017

- Platforms where you can run IdentityServer4
- Optimizing Tokens for size
- Identity vs Permissions
- Bootstraping OpenID Connect: Discovery
62.2 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

62.3 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
CHAPTER 63

Videos

63.1 2019

- 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

63.2 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

63.3 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
63.4 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

63.5 2015

- Securing Web APIs – Patterns & Anti-Patterns - Dominick Baier
- Authentication and authorization in modern JavaScript web applications – how hard can it be? - Brock Allen

63.6 2014

- Unifying Authentication & Delegated API Access for Mobile, Web and the Desktop with OpenID Connect and OAuth 2 - Dominick Baier