IdentityServer4 Documentation

Release 1.0.0

Brock Allen, Dominick Baier

Dec 04, 2018

Introduction

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core 2.

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.

The Big Picture

Most modern applications look more or less like this:
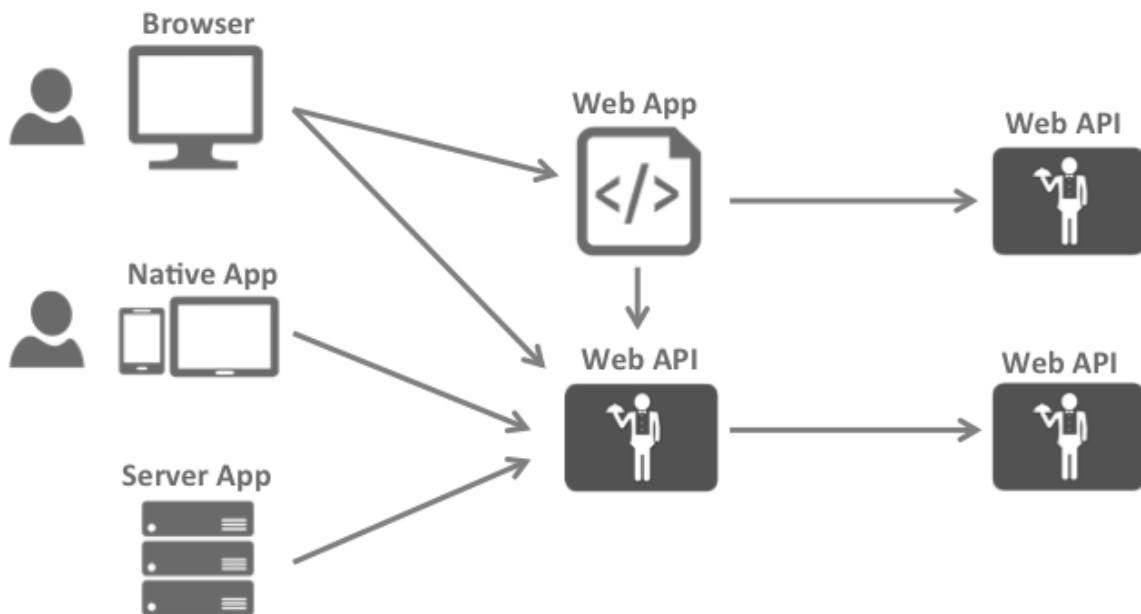
The most common interactions are:

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

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

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

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

Restructuring the application to support a security token service leads to the following architecture and protocols:
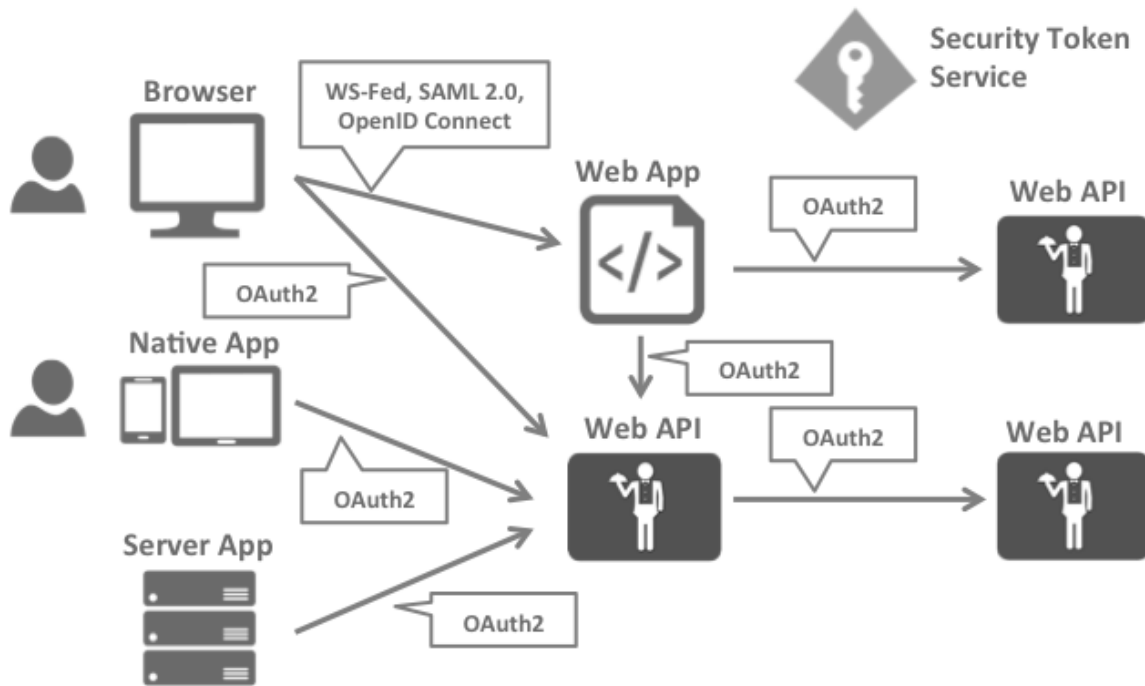
Such a design divides security concerns into two parts:

1.1 Authentication

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

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

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

1.2 API Access

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

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

1.3 OpenID Connect and OAuth 2.0 – better together

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

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

1.4 How IdentityServer4 can help

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

Typically, you build (or re-use) an application that contains a login and logout page (and maybe consent - depending on your needs), and the IdentityServer middleware adds the necessary protocol heads to it, so that client applications can talk to it using those standard protocols.
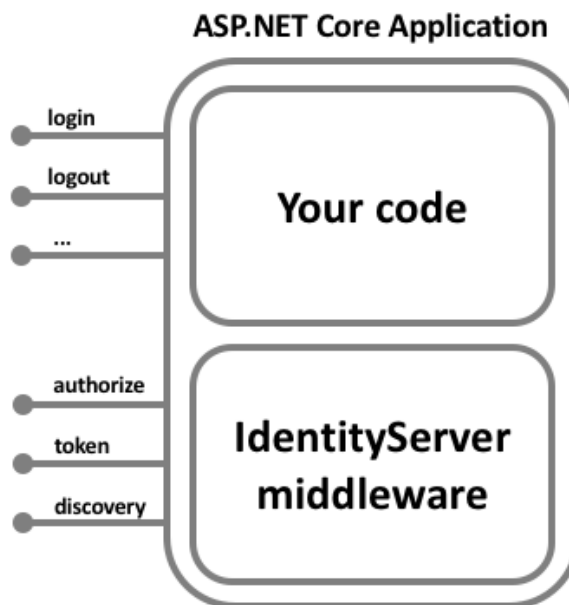
The hosting application can be as complex as you want, but we typically recommend to keep the attack surface as small as possible by including authentication related UI only.

Terminology

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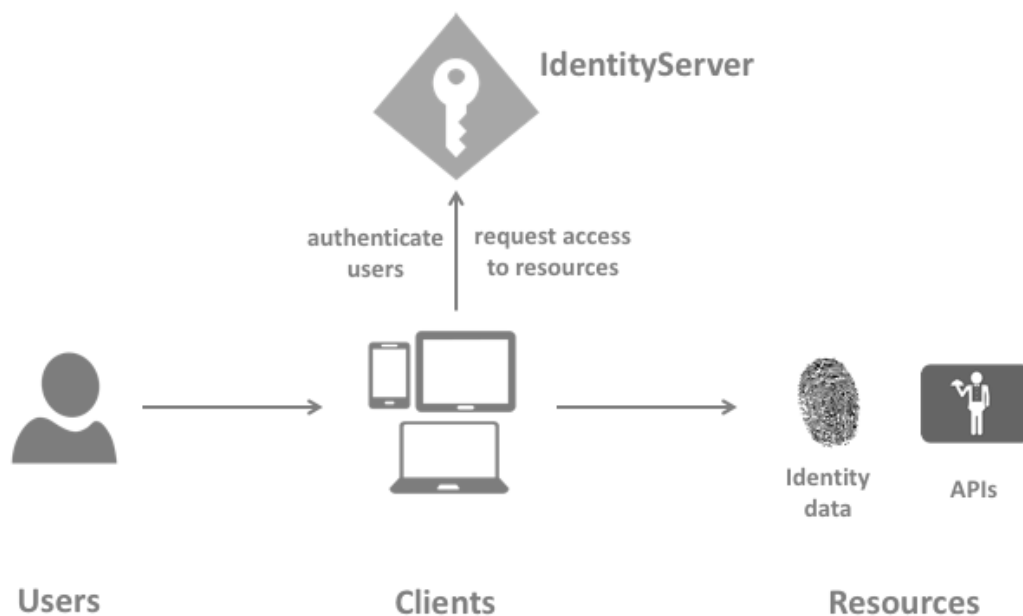
2.1 IdentityServer

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

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

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

IdentityServer has a number of jobs and features - including:

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

2.2 User

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

2.3 Client

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

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

2.4 Resources

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

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

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

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

2.5 Identity Token

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

2.6 Access Token

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

Supported Specifications

IdentityServer implements the following specifications:

3.1 OpenID Connect

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

3.2 OAuth 2.0

- OAuth 2.0 (RFC 6749)
- OAuth 2.0 Bearer Token Usage (RFC 6750)
- OAuth 2.0 Multiple Response Types (spec)
- OAuth 2.0 Form Post Response Mode (spec)
- OAuth 2.0 Token Revocation (RFC 7009)
- OAuth 2.0 Token Introspection (RFC 7662)
- Proof Key for Code Exchange (RFC 7636)
- JSON Web Tokens for Client Authentication (RFC 7523)
- OAuth 2.0 Device Flow for Browserless and Input Constrained Devices (draft)

Packaging and Builds

IdentityServer consists of a number of nuget packages.

4.1 IdentityServer4

nuget | github

Contains the core IdentityServer object model, services and middleware. Only contains support for in-memory configuration and user stores - but you can plug-in support for other stores via the configuration. This is what the other repos and packages are about.

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 ASP.NET Core Identity

nuget | github

ASP.NET Core Identity integration package for IdentityServer. This package provides a simple configuration API to use the ASP.NET Identity management library for your IdentityServer users.

4.5 EntityFramework Core

nuget | github

EntityFramework Core storage implementation for IdentityServer. This package provides an EntityFramework implementation for the configuration and operational stores in IdentityServer.

4.6 Dev builds

In addition we publish dev/interim builds to MyGet. Add the following feed to your Visual Studio if you want to give them a try:

https://www.myget.org/F/identity/

Support and Consulting Options

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

5.1 Free support

Free support is community-based and uses public forums

StackOverflow

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

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

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

Please use the `IdentityServer4` tag when asking new questions

Gitter

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

https://gitter.im/IdentityServer/IdentityServer4

Reporting a bug

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

Also check the contribution guidelines before posting.

5.2 Commercial support

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

Training

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

Admin UI, Identity Express and SAML2p support

There are a couple of commercial add-on products available from our partners, check https://www.identityserver.com/products/.

CHAPTER 6

Demo Server and Tests

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.

Furthermore we have a repo that exercises a variety of IdentityServer and Web API combinations (IdentityServer 3 and 4, ASP.NET Core and Katana). We use this test harness to make sure all permutations work. You can test it yourself by cloning this repo.

Contributing

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 Platform

IdentityServer is built against ASP.NET Core 2 and runs on .NET Framework 4.6.1 (and higher) and .NET Core 2 (and higher).

7.4 Bugs and feature requests?

Please log a new issue in the appropriate GitHub repo:

- Core
- Samples

- AccessTokenValidation

7.5 Other discussions

https://gitter.im/IdentityServer/IdentityServer4

7.6 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.

Note: **We only accept PRs to the dev branch.**

7.7 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

CHAPTER 8

Setup and Overview

There are two fundamental ways to start a new IdentityServer project:

- start from scratch
- start with the ASP.NET Identity template in Visual Studio

If you start from scratch, we provide a couple of helpers and in-memory stores, so you don't have to worry about persistence right from the start.

If you start with ASP.NET Identity, we provide an easy way to integrate with that as well.

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.

Every quickstart has a reference solution - you can find the code in the IdentityServer4.Samples repo in the quickstarts folder.

8.1 Basic setup

The screen shots show Visual Studio - but this is not a requirement.

Creating the quickstart IdentityServer

Start by creating a new ASP.NET Core project.
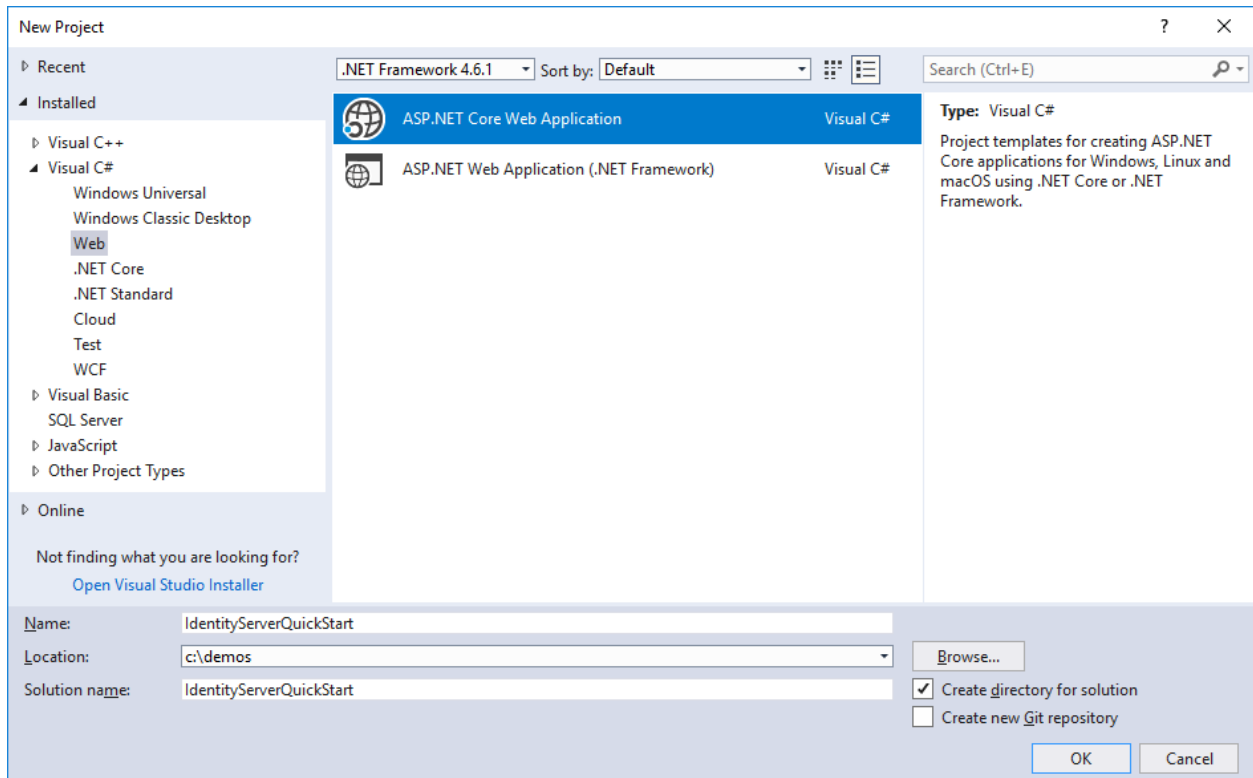
Then select the "Empty" option.
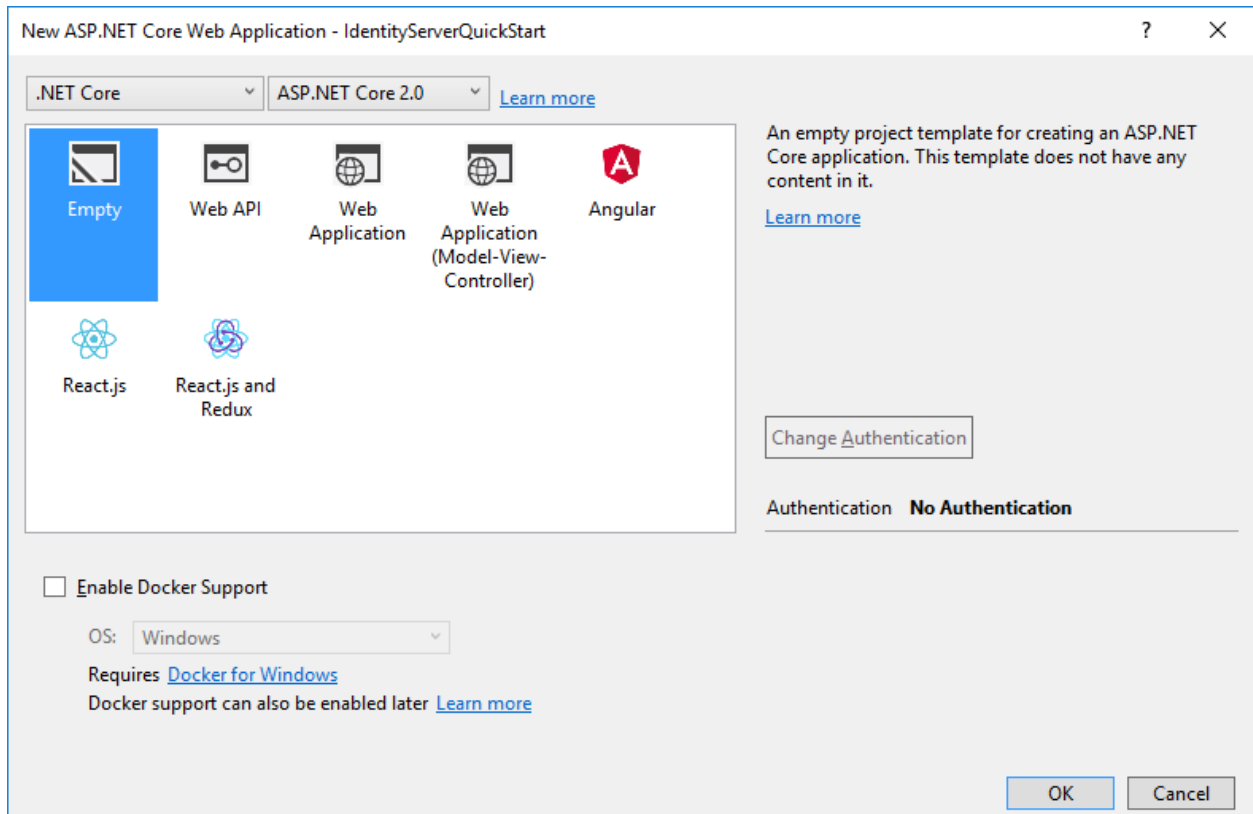
Next, add the *IdentityServer4* nuget package:
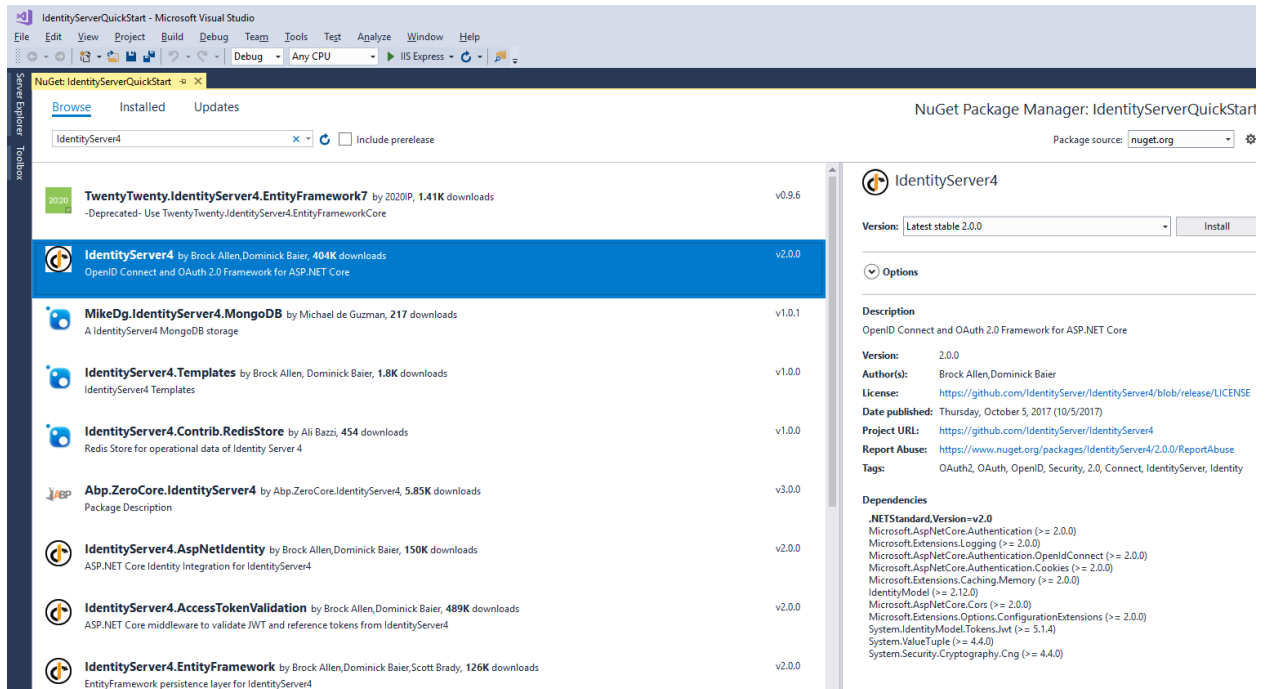
Alternatively you can use Package Manager Console to add the dependency by running the following command:

"Install-Package IdentityServer4"

Note: IdentityServer build numbers 1.x target ASP.NET Core 1.1, and IdentityServer build numbers 2.x targets ASP.NET Core 2.0 .

IdentityServer uses the usual pattern to configure and add services to an ASP.NET Core host. In `ConfigureServices` the required services are configured and added to the DI system. In `Configure` the middleware is added to the HTTP pipeline.

Modify your `Startup.cs` file to look like this:

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

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseIdentityServer();
    }
}
```

`AddIdentityServer` registers the IdentityServer services in DI. It also registers an in-memory store for runtime

state. This is useful for development scenarios. For production scenarios you need a persistent or shared store like a database or cache for that. See the *EntityFramework* quickstart for more information.

The `AddDeveloperSigningCredential` extension creates temporary key material for signing tokens. Again this might be useful to get started, but needs to be replaced by some persistent key material for production scenarios. See the *cryptography docs* for more information.

Note: IdentityServer is not yet ready to be launched. We will add the required services in the following quickstarts.

8.2 Modify hosting

By default Visual Studio uses IIS Express to host your web project. This is totally fine, except that you won't be able to see the real time log output to the console.

IdentityServer makes extensive use of logging whereas the "visible" error message in the UI or returned to clients are deliberately vague.

We recommend to run IdentityServer in the console host. You can do this by switching the launch profile in Visual Studio. You also don't need to launch a browser every time you start IdentityServer - you can turn that off as well:
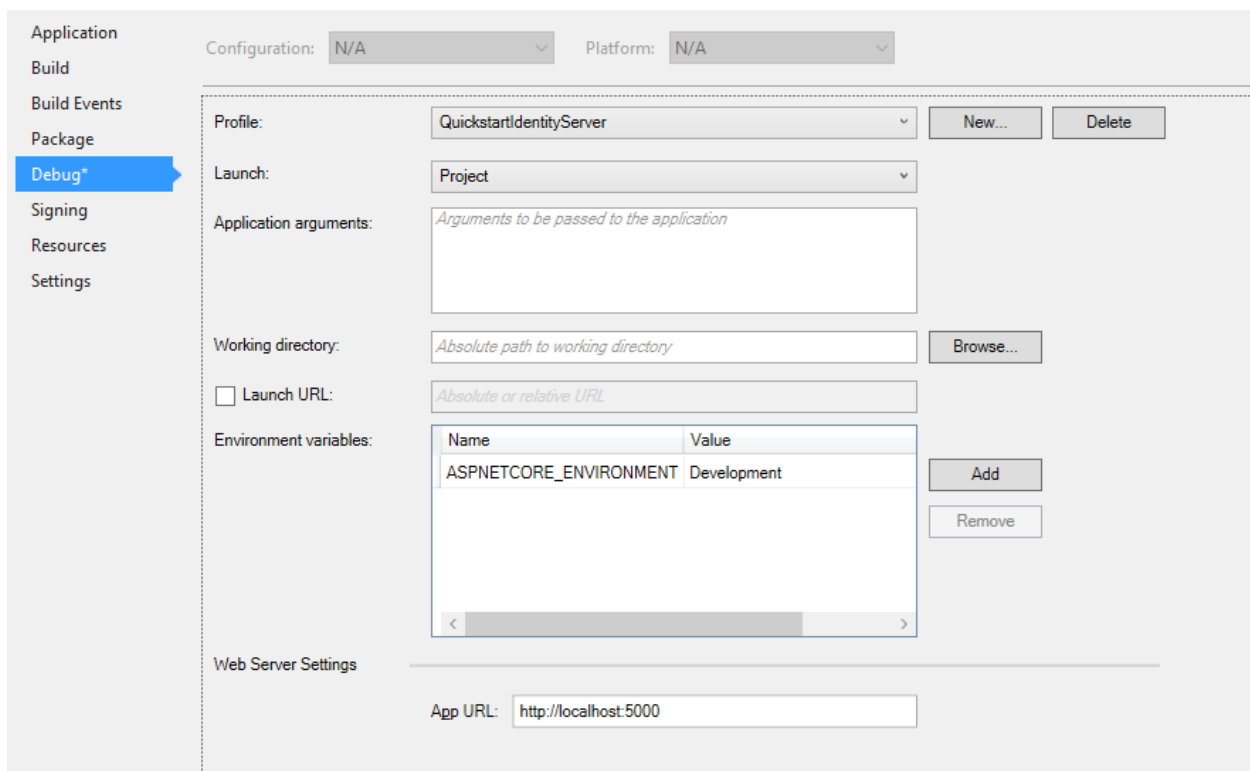
In addition, it will be helpful to run IdentityServer on a consistent URL for these quickstarts. You should also configure this URL in the launch profile dialog above, and use `http://localhost:5000/`. In the above screenshot you can see this URL has been configured.

Note: We recommend to configure the same port for IIS Express and self-hosting. This way you can switch between the two without having to modify any configuration in your clients.

To then choose the console host when you launch, you must select it in the launch menu from Visual Studio:
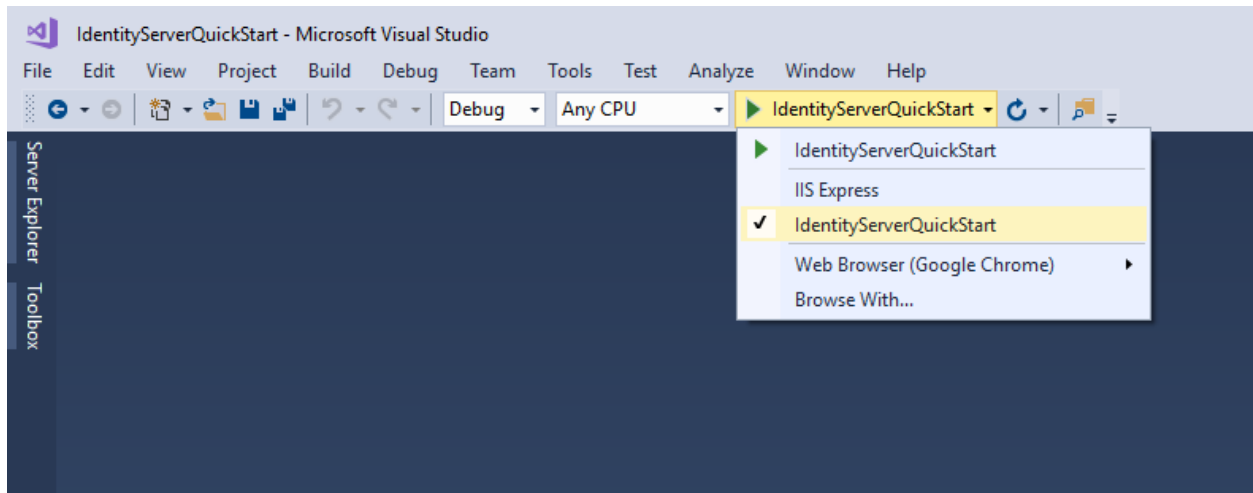
8.3 How to run the quickstart

As mentioned above every quickstart has a reference solution - you can find the code in the IdentityServer4.Samples repo in the quickstarts folder.

The easiest way to run the individual parts of a quickstart solution is to set the startup mode to "current selection". Right click the solution and select "Set Startup Projects":
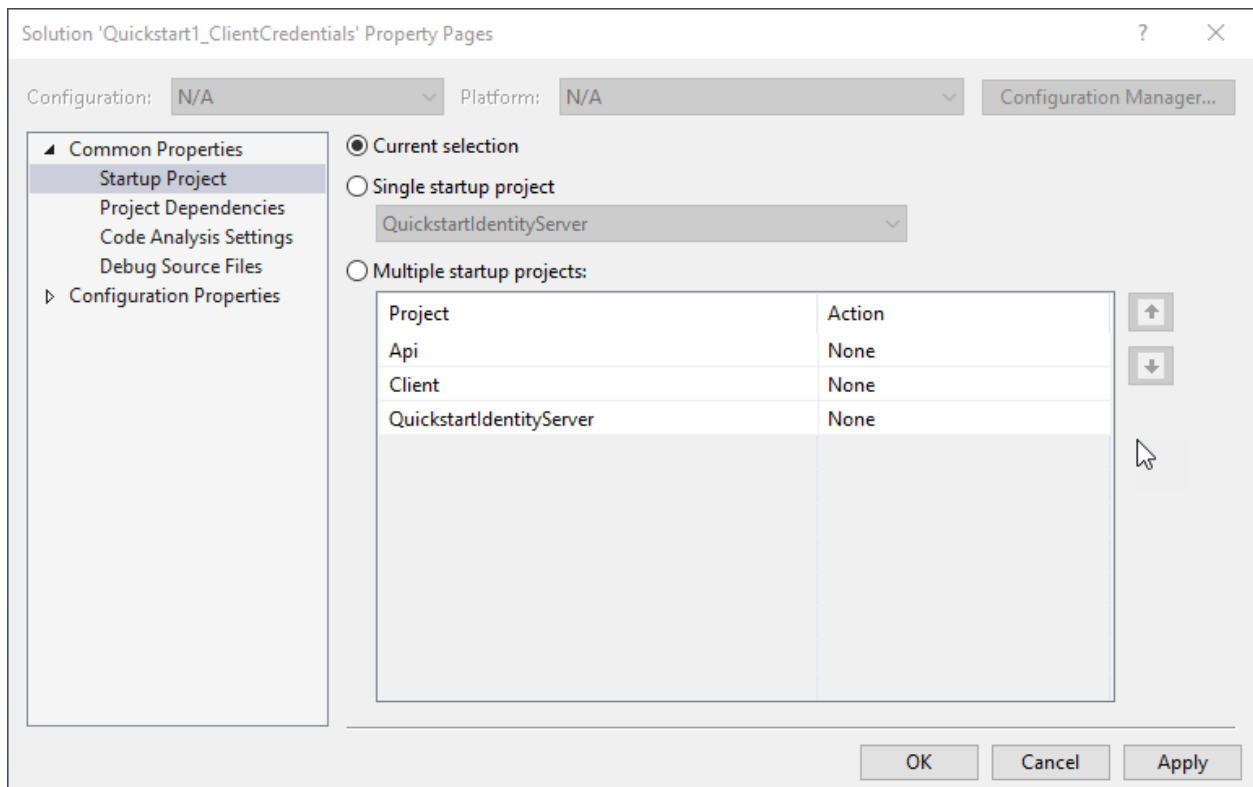
Typically you start IdentityServer first, then the API, and then the client. Only run in the debugger if you actually want to debug. Otherwise Ctrl+F5 is the best way to run the projects.

Protecting an API using Client Credentials

This quickstart presents the most basic scenario for protecting APIs using IdentityServer.

In this scenario we will define an API and a client that wants to access it. The client will request an access token at IdentityServer and use it to gain access to the API.

9.1 Defining the API

Scopes define the resources in your system that you want to protect, e.g. APIs.

Since we are using the in-memory configuration for this walkthrough - all you need to do to add an API, is to create an object of type `ApiResource` and set the appropriate properties.

Add a file (e.g. `Config.cs`) into your project and add the following code:

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

9.2 Defining the client

The next step is to define a client that can access this API.

For this scenario, the client will not have an interactive user, and will authenticate using the so called client secret with IdentityServer. Add the following code to your *Config.cs* file:

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

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

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

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

9.3 Configure IdentityServer

To configure IdentityServer to use your scopes and client definition, you need to add code to the `ConfigureServices` method. You can use convenient extension methods for that - under the covers these add the relevant stores and data into the DI system:

```
public void ConfigureServices(IServiceCollection services)
{
    // configure identity server with in-memory stores, keys, clients and resources
    services.AddIdentityServer()
        .AddDeveloperSigningCredential()
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients());
}
```

That's it - 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. This will be used by your clients and APIs to download the necessary configuration data.

Dominick (Dev)

localhost:5000/.well-know ×

localhost:5000/.well-known/openid-configuration

{ } [ ]

```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/endsession",
    "check_session_iframe": "http://localhost:5000/connect/checksession",
    "revocation_endpoint": "http://localhost:5000/connect/revocation",
    "introspection_endpoint": "http://localhost:5000/connect/introspect",
    "frontchannel_logout_supported": true,
    "frontchannel_logout_session_supported": true,
    "scopes_supported": [
        "api1"
    ],
    "claims_supported": [
    ],
    "response_types_supported": [
        "code",
        "token",
        "id_token",
        "id_token token",
        "code id_token",
        "code token",
        "code id_token token"
    ],
    "response_modes_supported": [
        "form_post",
        "query",
        "fragment"
    ],
    "grant_types_supported": [
        "authorization_code",
        "client_credentials",
        "refresh_token",
        "implicit"
    ],
    "subject_types_supported": [
        "public"
    ],
    "id_token_signing_alg_values_supported": [
        "RS256"
    ],
    "token_endpoint_auth_methods_supported": [
        "client_secret_basic",
        "client_secret_post"
    ],
    "code_challenge_methods_supported": [
        "plain",
        "S256"
    ]
}
```

9.4 Adding an API

Next, add an API to your solution.

You can use the ASP.NET Core Web API template. Again, we recommend you take control over the ports and use the same technique as you used to configure Kestrel and the launch profile as before. This walkthrough assumes you have configured your API to run on `http://localhost:5001`.

The controller

Add a new controller to your API project:

```
[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.

Configuration

The last step is to add the authentication services to DI 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 scope)

Add the *IdentityServer4.AccessTokenValidation* NuGet package to your project.
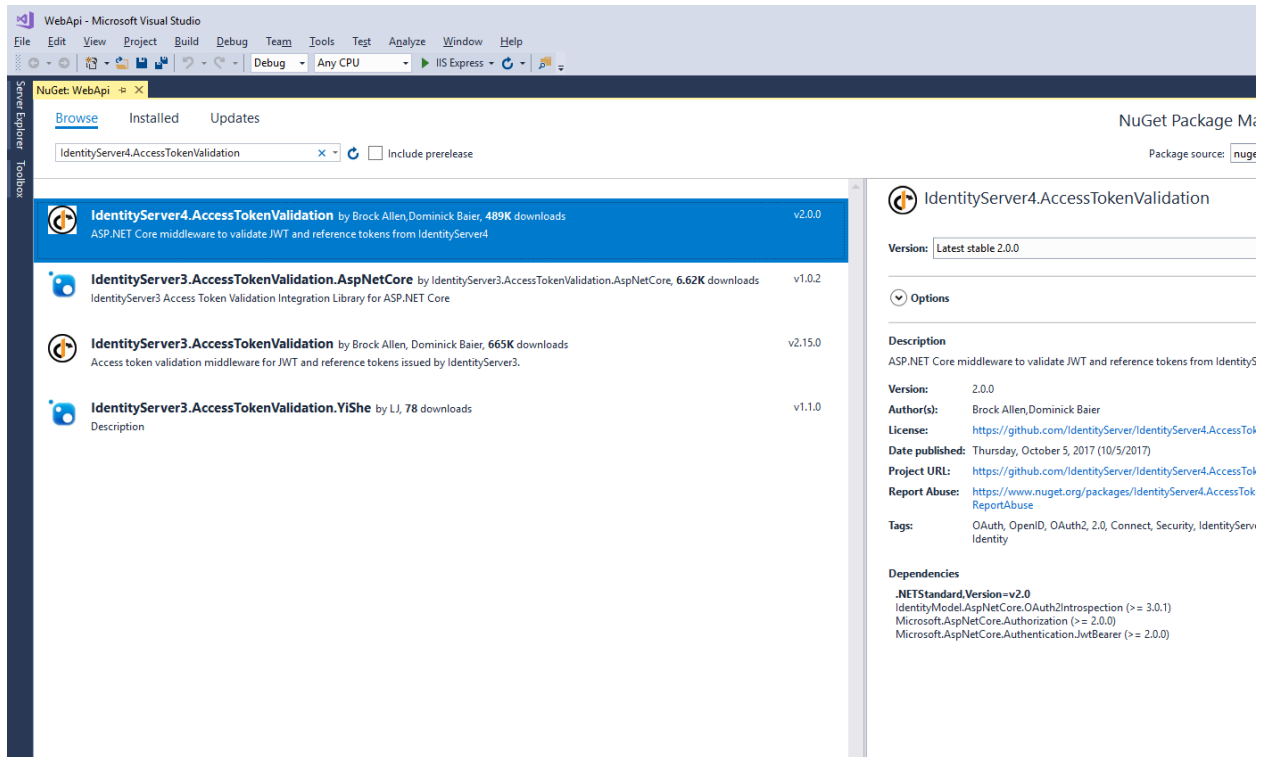
Update *Startup* to look like this:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvcCore()
            .AddAuthorization()
            .AddJsonFormatters();

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

                options.ApiName = "api1";
            });
    }

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

        app.UseMvc();
    }
}
```

`AddAuthentication` adds the authentication services to DI and configures `"Bearer"` as the default scheme.
`AddIdentityServerAuthentication` adds the IdentityServer access token validation handler into DI for use
by the authentication services. `UseAuthentication` adds the authentication middleware to the pipeline so authen-
tication will be performed automatically on every call into the host.

If you use the browser to navigate to the controller (`http://localhost:5001/identity`), you should get a 401 status code in return. This means your API requires a credential.

That's it, the API is now protected by IdentityServer.

9.5 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 (see full code here).

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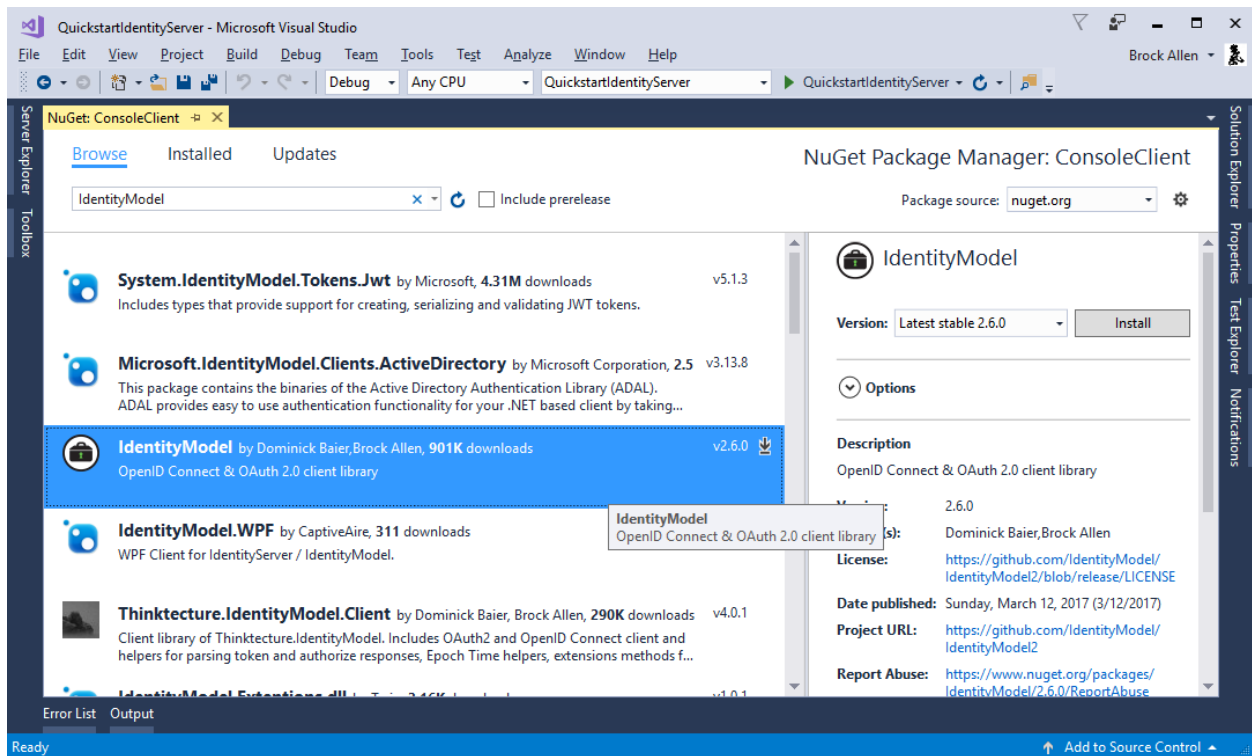
IdentityModel includes a client library to use with the discovery endpoint. This way you only need to know the base-address of IdentityServer - the actual endpoint addresses can be read from the metadata:

```
// discover endpoints from metadata
var disco = await DiscoveryClient.GetAsync("http://localhost:5000");
if (disco.IsError)
{
    Console.WriteLine(disco.Error);
    return;
}
```

Next you can use the `TokenClient` class to request the token. To create an instance you need to pass in the token endpoint address, client id and secret.

Next you can use the `RequestClientCredentialsAsync` method to request a token for your API:

```
// request token
var tokenClient = new TokenClient(disco.TokenEndpoint, "client", "secret");
var tokenResponse = await tokenClient.RequestClientCredentialsAsync("api1");

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

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

Note: Copy and paste the access token from the console to jwt.io to inspect the raw token.
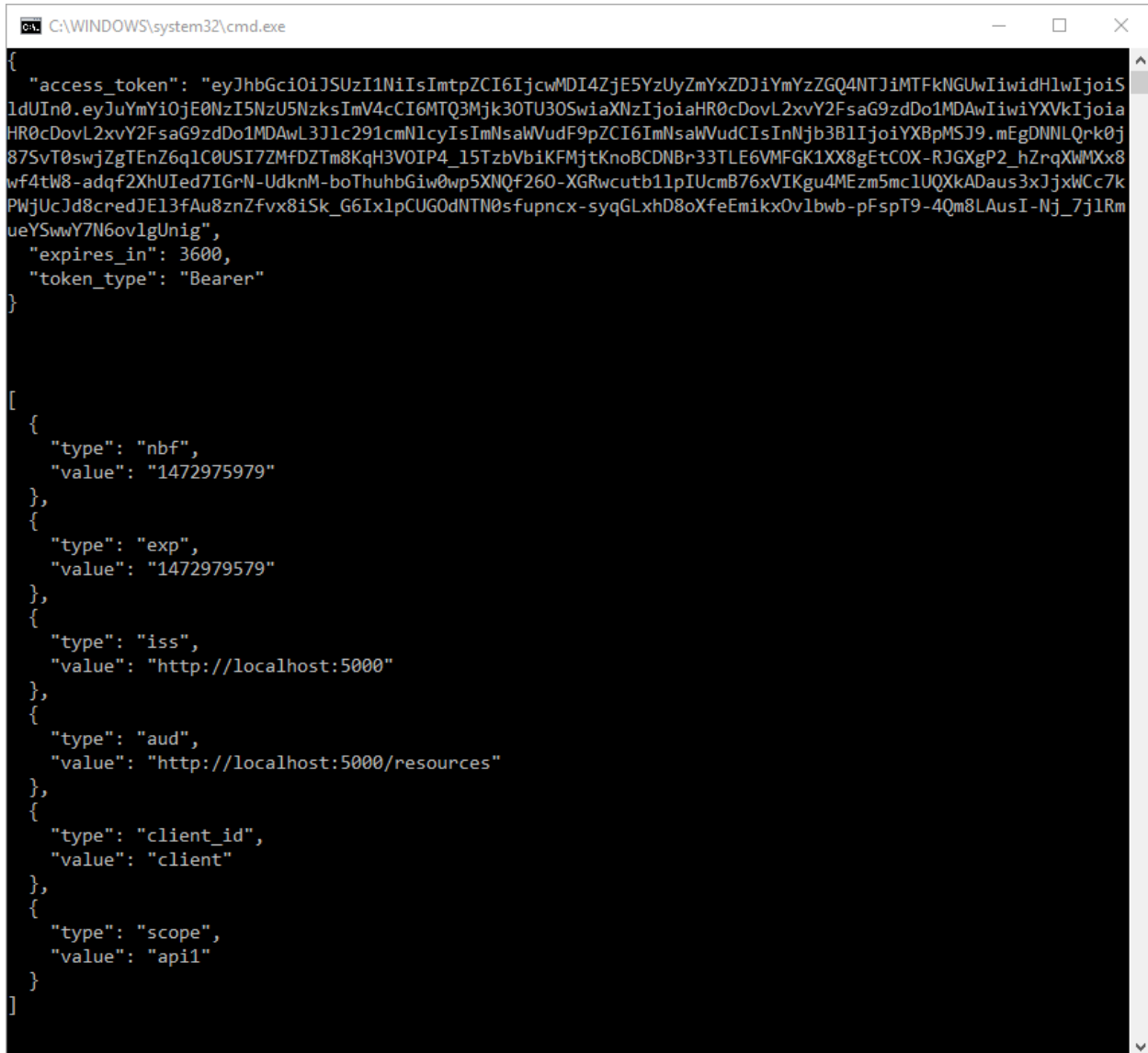
The last step is now to call the API.

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

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

var response = await client.GetAsync("http://localhost:5001/identity");
if (!response.IsSuccessStatusCode)
{
    Console.WriteLine(response.StatusCode);
}
else
{
    var content = await response.Content.ReadAsStringAsync();
    Console.WriteLine(JArray.Parse(content));
}
```

The output should look like this:

```
C:\WINDOWS\system32\cmd.exe                                         —    □    ✕
{
  "access_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjcwMDI4ZjE5YzUyZmYxZDJiYmYzZGQ4NTJiMTFkNGUwIiwidHlwIjoiS
ldUIn0.eyJuYmYiOjE0NzI5NzU5NzksImV4cCI6MTQ3Mjk3OTU3OSwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo1MDAwIiwiYXVkIjoia
HR0cDovL2xvY2FsaG9zdDo1MDAwL3Jlc291cmNlcyIsImNsaWVudF9pZCI6ImNsaWVudCIsInNjb3BlIjoiYXBpMSJ9.mEgDNNLQrk0j
87SvT0swjZgTEnZ6qlC0USI7ZMfDZTm8KqH3VOIP4_15TzbVbiKFMjtKnoBCDNBr33TLE6VMFGK1XX8gEtCOX-RJGXgP2_hZrqXWMXx8
wf4tW8-adqf2XhUIed7IGrN-UdknM-boThuhbGiw0wp5XNQf26O-XGRwcutb11pIUcmB76xVIKgu4MEzm5mclUQXkADaus3xJjxWCc7k
PWjUcJd8credJEl3fAu8znZfvx8iSk_G6IxlpCUGOdNTN0sfupncx-syqGLxhD8oXfeEmikxOvlbwb-pFspT9-4Qm8LAusI-Nj_7jlRm
ueYSwwY7N6ovlgUnig",
  "expires_in": 3600,
  "token_type": "Bearer"
}




[
  {
    "type": "nbf",
    "value": "1472975979"
  },
  {
    "type": "exp",
    "value": "1472979579"
  },
  {
    "type": "iss",
    "value": "http://localhost:5000"
  },
  {
    "type": "aud",
    "value": "http://localhost:5000/resources"
  },
  {
    "type": "client_id",
    "value": "client"
  },
  {
    "type": "scope",
    "value": "api1"
  }
]
```

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

9.6 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

Protecting an API using Passwords

The OAuth 2.0 resource owner password grant allows a client to send username and password to the token service and get an access token back that represents that user.

The spec recommends using the resource owner password grant only for "trusted" (or legacy) applications. Generally speaking you are typically far better off using one of the interactive OpenID Connect flows when you want to authenticate a user and request access tokens.

Nevertheless, this grant type allows us to introduce the concept of users to our quickstart IdentityServer, and that's why we show it.

10.1 Adding users

Just like there are in-memory stores for resources (aka scopes) and clients, there is also one for users.

Note: Check the ASP.NET Identity based quickstarts for more information on how to properly store and manage user accounts.

The class `TestUser` represents a test user and its claims. Let's create a couple of users by adding the following code to our config class:

First add the following using statement to the `Config.cs` file:

```
using IdentityServer4.Test;

public static List<TestUser> GetUsers()
{
    return new List<TestUser>
    {
        new TestUser
        {
            SubjectId = "1",
```

```
            Username = "alice",
            Password = "password"
        },
        new TestUser
        {
            SubjectId = "2",
            Username = "bob",
            Password = "password"
        }
    };
}
```

Then register the test users with IdentityServer:

```
public void ConfigureServices(IServiceCollection services)
{
    // configure identity server with in-memory stores, keys, clients and scopes
    services.AddIdentityServer()
        .AddDeveloperSigningCredential()
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients())
        .AddTestUsers(Config.GetUsers());
}
```

The `AddTestUsers` extension method does a couple of things under the hood

- adds support for the resource owner password grant

- adds support to user related services typically used by a login UI (we'll use that in the next quickstart)

- adds support for a profile service based on the test users (you'll learn more about that in the next quickstart)

10.2 Adding a client for the resource owner password grant

You could simply add support for the grant type to our existing client by changing the `AllowedGrantTypes` property. If you need your client to be able to use both grant types that is absolutely supported.

Typically you want to create a separate client for the resource owner use case, add the following to your clients configuration:

```
public static IEnumerable<Client> GetClients()
{
    return new List<Client>
    {
        // other clients omitted...

        // resource owner password grant client
        new Client
        {
            ClientId = "ro.client",
            AllowedGrantTypes = GrantTypes.ResourceOwnerPassword,

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

```
            AllowedScopes = { "api1" }
        }
    };
}
```

10.3 Requesting a token using the password grant

The client looks very similar to what we did for the client credentials grant. The main difference is now that the client would collect the user's password somehow, and send it to the token service during the token request.

Again IdentityModel's `TokenClient` can help out here:

```
// request token
var tokenClient = new TokenClient(disco.TokenEndpoint, "ro.client", "secret");
var tokenResponse = await tokenClient.RequestResourceOwnerPasswordAsync("alice",
→"password", "api1");

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

Console.WriteLine(tokenResponse.Json);
Console.WriteLine("\n\n");
```

When you send the token to the identity API endpoint, you will notice one small but important difference compared to the client credentials grant. The access token will now contain a `sub` claim which uniquely identifies the user. This "sub" claim can be seen by examining the content variable after the call to the API and also will be displayed on the screen by the console application.

The presence (or absence) of the `sub` claim lets the API distinguish between calls on behalf of clients and calls on behalf of users.

Adding User Authentication with OpenID Connect

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

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

11.1 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 either clone or download this repo and drop the controllers, views, models and CSS into your IdentityServer web application.

Alternatively you can run this command from the command line in the same directory as your IdentityServer web application to automate the download:

```
iex ((New-Object System.Net.WebClient).DownloadString('https://raw.githubusercontent.
→com/IdentityServer/IdentityServer4.Quickstart.UI/master/getmaster.ps1'))
```

For Unix/Linux:

```
\curl -L https://raw.githubusercontent.com/IdentityServer/IdentityServer4.Quickstart.
→UI/master/getmaster.sh | bash
```

Once you have added the MVC UI assets, you will also need to add MVC to the hosting application, both in the DI system and in the pipeline. Add MVC to `ConfigureServices` with the `AddMvc` extension method:

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

(continues on next page)

```
    // configure identity server with in-memory stores, keys, clients and scopes
    services.AddIdentityServer()
        .AddDeveloperSigningCredential()
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients())
        .AddTestUsers(Config.GetUsers());
}
```

Add MVC as the last middleware in the pipeline in `Configure` with the `UseMvc` extension method:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseIdentityServer();

    app.UseStaticFiles();
    app.UseMvcWithDefaultRoute();
}
```

See the readme for the quickstart UI for more information.

Note: The `release` branch of the UI repo has the UI that matches the latest stable release. The `dev` branch goes along with the current dev build of IdentityServer4. If you are looking for a specific version of the UI - check the tags.

Spend some time inspecting the controllers and models, 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.

11.2 Creating an MVC client

Next you will add an MVC application to your solution. Use the ASP.NET Core "Web Application" (i.e. MVC) template for that. Don't configure the "Authentication" settings in the wizard – you will do this manually in this quickstart. Once you've created the project, configure the application to use port 5002 (see the overview part for instructions on how to do that).

To add support for OpenID Connect authentication to the MVC application, add the following to `ConfigureServices` in `Startup`:

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

    JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

    services.AddAuthentication(options =>
        {
            options.DefaultScheme = "Cookies";
            options.DefaultChallengeScheme = "oidc";
```

```
        })
        .AddCookie("Cookies")
        .AddOpenIdConnect("oidc", options =>
        {
            options.SignInScheme = "Cookies";

            options.Authority = "http://localhost:5000";
            options.RequireHttpsMetadata = false;

            options.ClientId = "mvc";
            options.SaveTokens = true;
        });
}
```

`AddAuthentication` adds the authentication services to DI. We are using a cookie as the primary means to authenticate a user (via `"Cookies"` as the `DefaultScheme`). We set the `DefaultChallengeScheme` to `"oidc"` because when we need the user to login, we will be using the OpenID Connect scheme.

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 that we are trusting IdentityServer. We then identify this client via the `ClientId`. `SignInScheme` is used to issue a cookie using the cookie handler once the OpenID Connect protocol is complete. And `SaveTokens` is used to persist the tokens from IdentityServer in the cookie (as they will be needed later).

As well, we've turned off the JWT claim type mapping to allow well-known claims (e.g. 'sub' and 'idp') to flow through unmolested:

```
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();
```

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

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseAuthentication();

    app.UseStaticFiles();
    app.UseMvcWithDefaultRoute();
}
```

The authentication middleware should be added before the MVC in the pipeline.

The last step is to trigger the authentication handshake. For that go to the home controller and add the `[Authorize]` on one of the actions. Also modify the view of that action to display the claims of the user, e.g.:

```
<dl>
    @foreach (var claim in User.Claims)
```

```
    {
        <dt>@claim.Type</dt>
        <dd>@claim.Value</dd>
    }
</dl>
```

If you now navigate to that controller 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.

11.3 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 adding a new helper (in `Config.cs`) to create a collection of `IdentityResource` objects:

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

Note: All standard scopes and their corresponding claims can be found in the OpenID Connect specification

You will then need to add these identity resources to your IdentityServer configuration in `Startup.cs`. Use the `AddInMemoryIdentityResources` extension method where you call `AddIdentityServer()`:

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

    // configure identity server with in-memory stores, keys, clients and scopes
    services.AddIdentityServer()
        .AddDeveloperSigningCredential()
        .AddInMemoryIdentityResources(Config.GetIdentityResources())
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients())
        .AddTestUsers(Config.GetUsers());
}
```

11.4 Adding a client for OpenID Connect implicit flow

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.

Add the following to your clients configuration:

```
public static IEnumerable<Client> GetClients()
{
    return new List<Client>
    {
        // other clients omitted...

        // OpenID Connect implicit flow client (MVC)
        new Client
        {
            ClientId = "mvc",
            ClientName = "MVC Client",
            AllowedGrantTypes = GrantTypes.Implicit,

            // 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
            }
        }
    };
}
```

11.5 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.
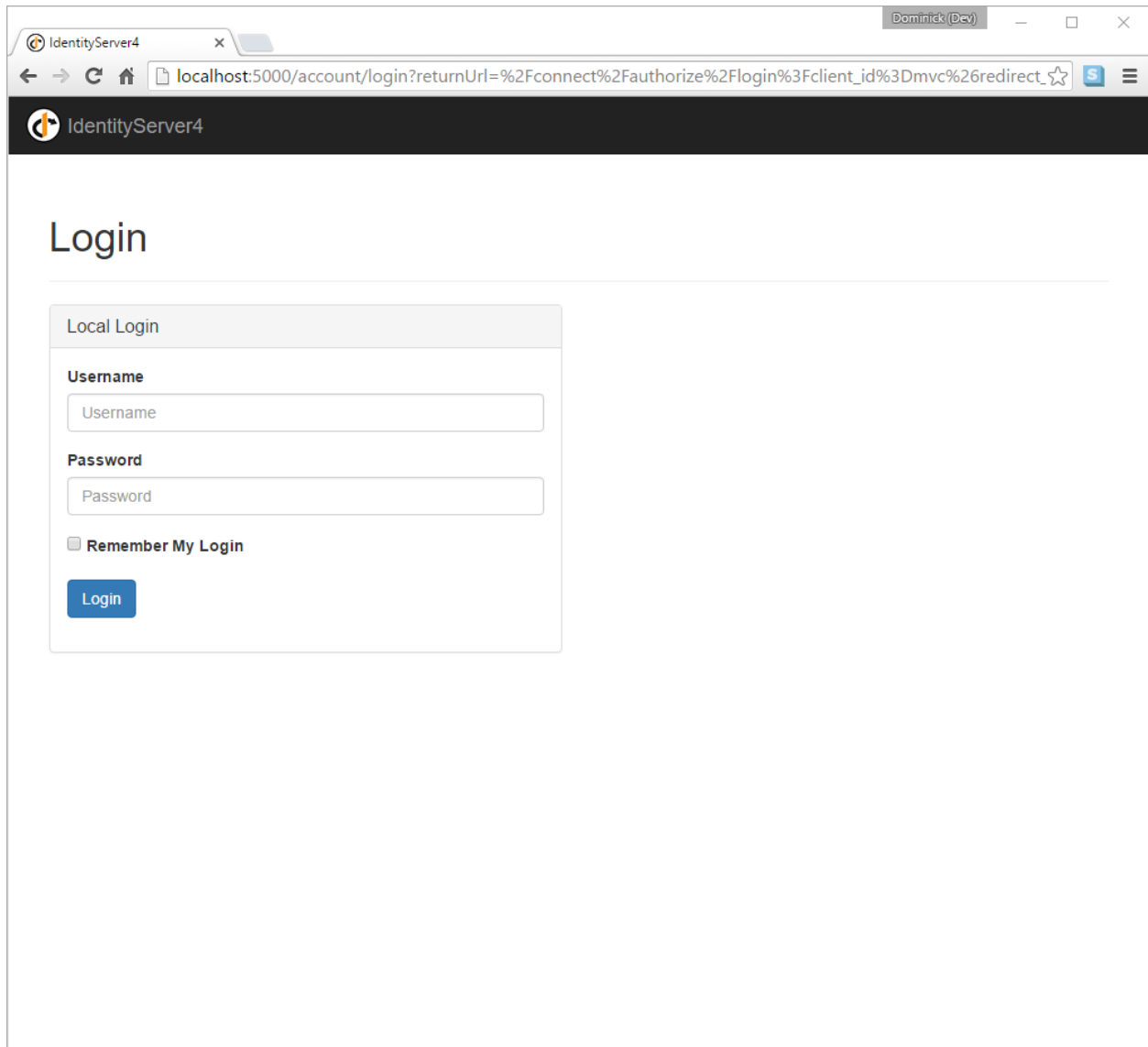
After successful login, the user is presented with the consent screen. Here the user can decide if he wants to release his identity information to the client application.

Note: Consent can be turned off on a per client basis using the `RequireConsent` property on the client object.
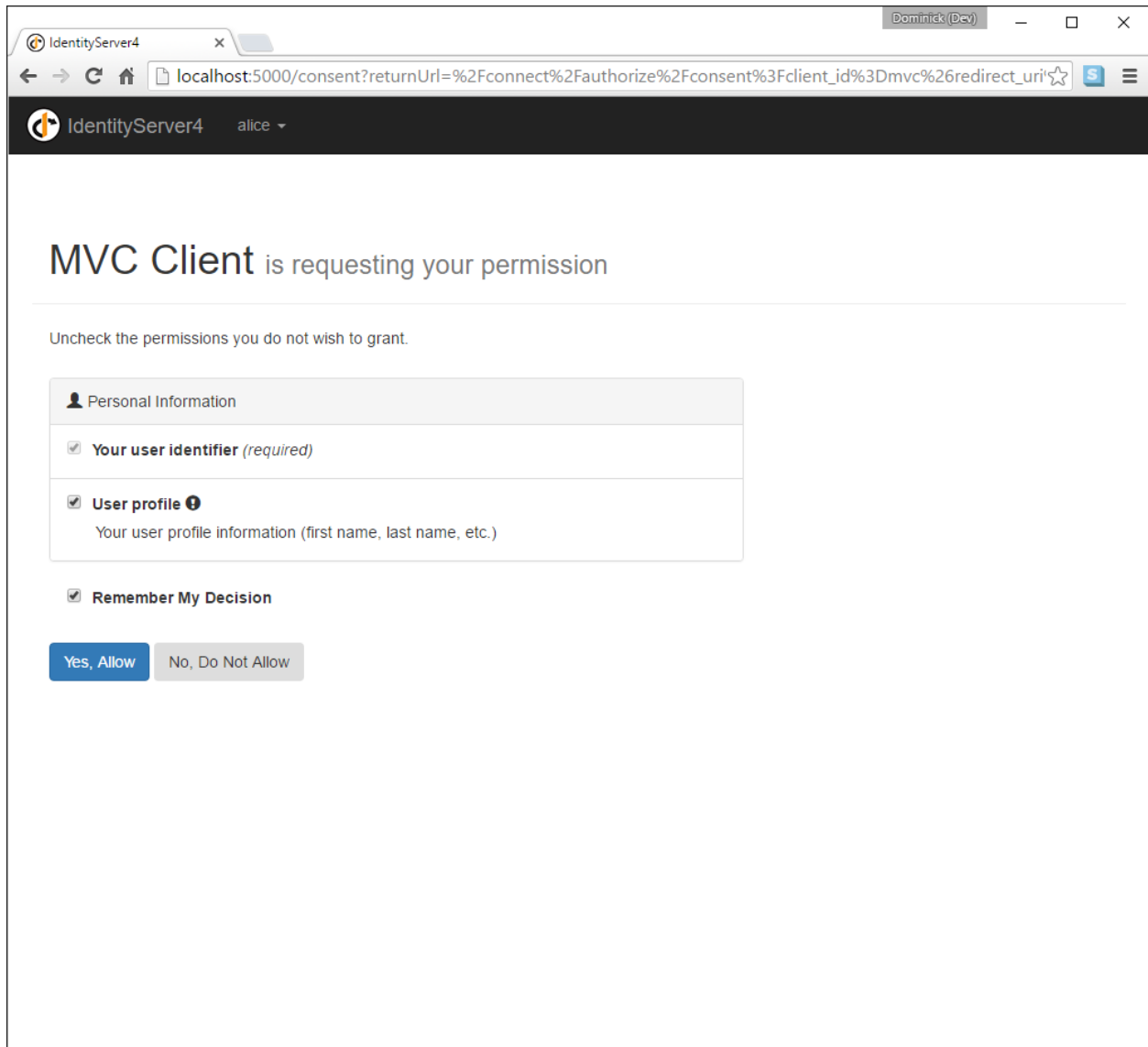
..and finally the browser redirects back to the client application, which shows the claims of the user.
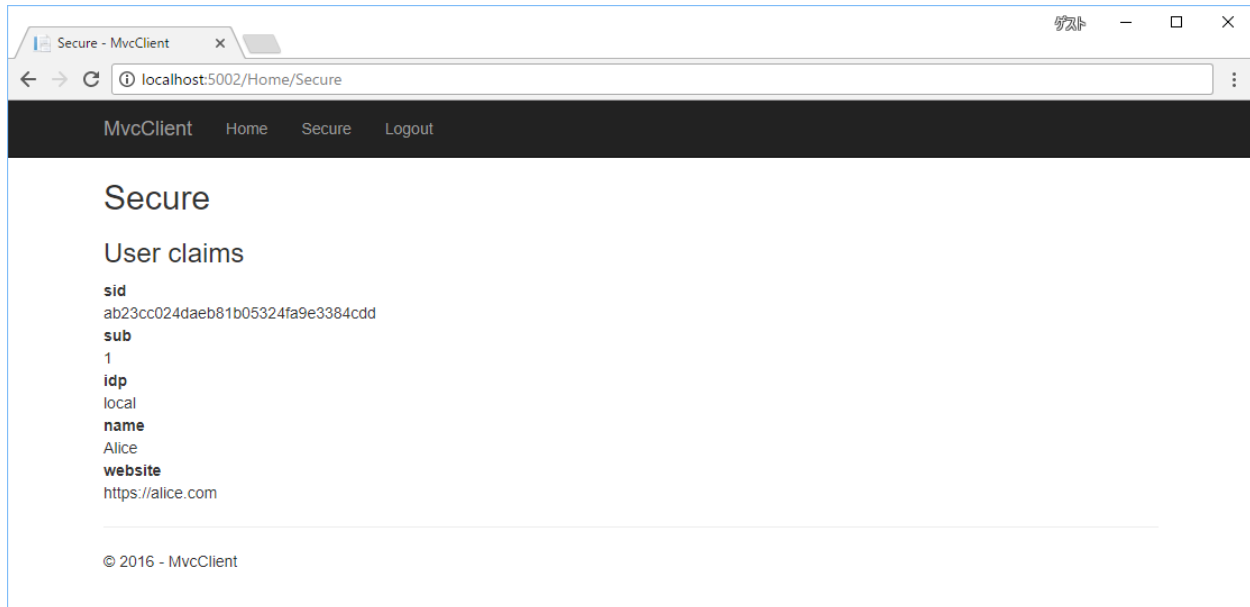
Note: During development you might sometimes see an exception stating that the token could not be validated. This is due to the fact that the signing key material is created on the fly and kept in-memory only. This exception happens when the client and IdentityServer get out of sync. Simply repeat the operation at the client, the next time the metadata has caught up, and everything should work normal again.

11.6 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 middleware, simply add the following code to some controller to trigger the sign-out:

```
public async Task Logout()
{
    await HttpContext.SignOutAsync("Cookies");
    await HttpContext.SignOutAsync("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.

11.7 Further experiments

As mentioned above, the OpenID Connect middleware asks for the *profile* scope by default. This scope also includes claims like *name* or *website*.

Let's add these claims to the user, so IdentityServer can put them into the identity token:

```csharp
public static List<TestUser> GetUsers()
{
    return new List<TestUser>
    {
        new TestUser
        {
            SubjectId = "1",
            Username = "alice",
            Password = "password",

            Claims = new []
            {
                new Claim("name", "Alice"),
                new Claim("website", "https://alice.com")
            }
        },
        new TestUser
        {
            SubjectId = "2",
            Username = "bob",
            Password = "password",

            Claims = new []
            {
                new Claim("name", "Bob"),
                new Claim("website", "https://bob.com")
            }
        }
    };
}
```

Next time you authenticate, your claims page will now show the additional claims.

Feel free to add more claims - and also more scopes. The `Scope` property on the OpenID Connect middleware is where you configure which scopes will be sent to IdentityServer during authentication.

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.

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.

12.1 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).

If you are running on port 5000 - you can simply use the client id/secret from the code snippet below, since this is pre-registered by us.

Start by adding the Google authentication handler to DI. This is done by adding this snippet to `ConfigureServices` in `Startup`:

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

    // configure identity server with in-memory stores, keys, clients and scopes
    services.AddIdentityServer()
        .AddDeveloperSigningCredential()
        .AddInMemoryIdentityResources(Config.GetIdentityResources())
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients())
        .AddTestUsers(Config.GetUsers());

    services.AddAuthentication()
        .AddGoogle("Google", options =>
        {
```

(continues on next page)

```
        options.SignInScheme = IdentityServerConstants.
→ExternalCookieAuthenticationScheme;

        options.ClientId = "434483408261-55tc8n0cs4ff1fe21ea8df2o443v2iuc.apps.
→googleusercontent.com";
        options.ClientSecret = "3gcoTrEDPPJ0ukn_aYYT6PWo";
    });
}
```

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. For a better understanding of how this is done, see the `AccountController` class under the *Quickstart* folder.

Now run the MVC client and try to authenticate - you will see a Google button on the login page:
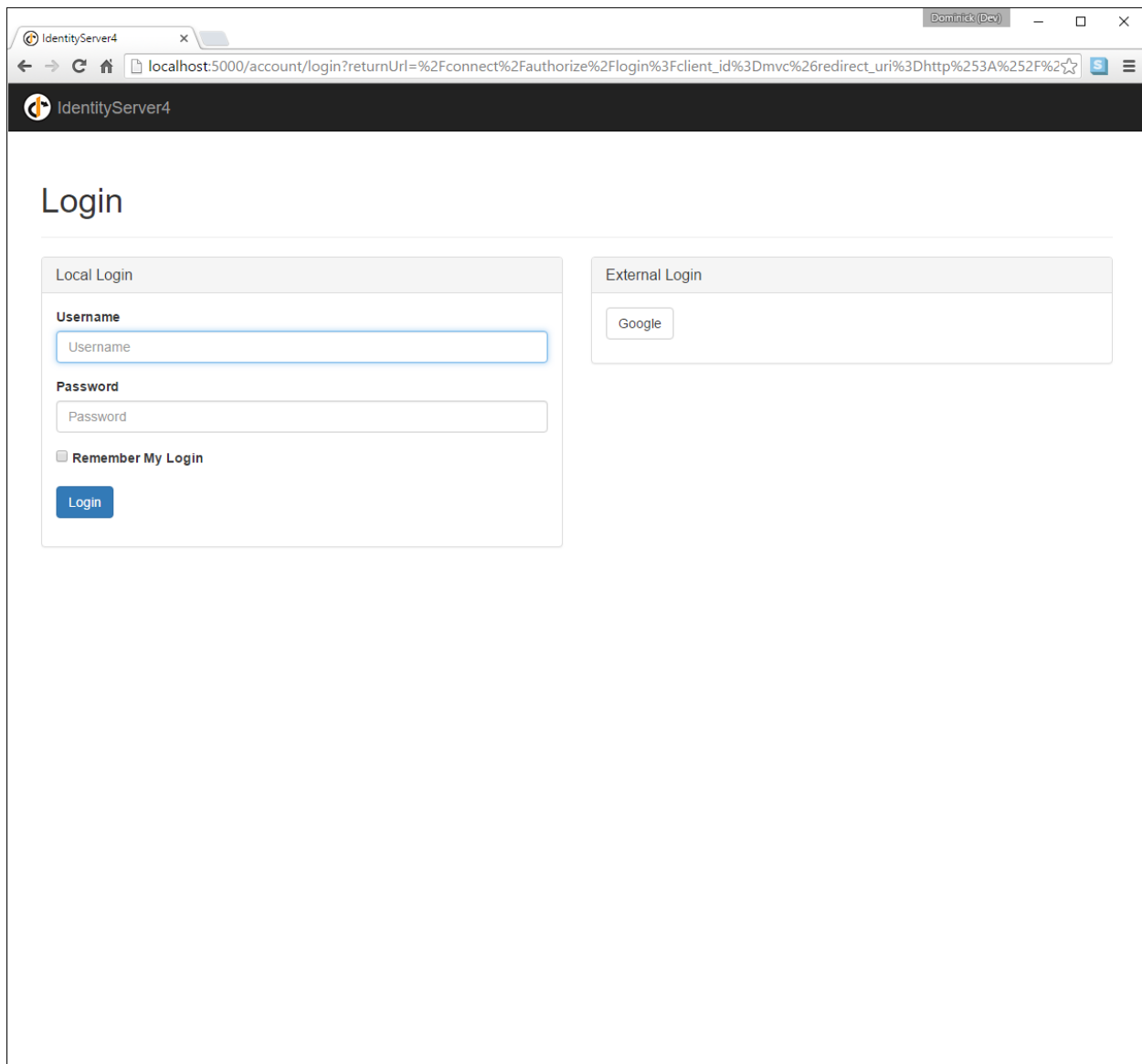
After authentication, you can see that the claims are now being sourced from Google data.
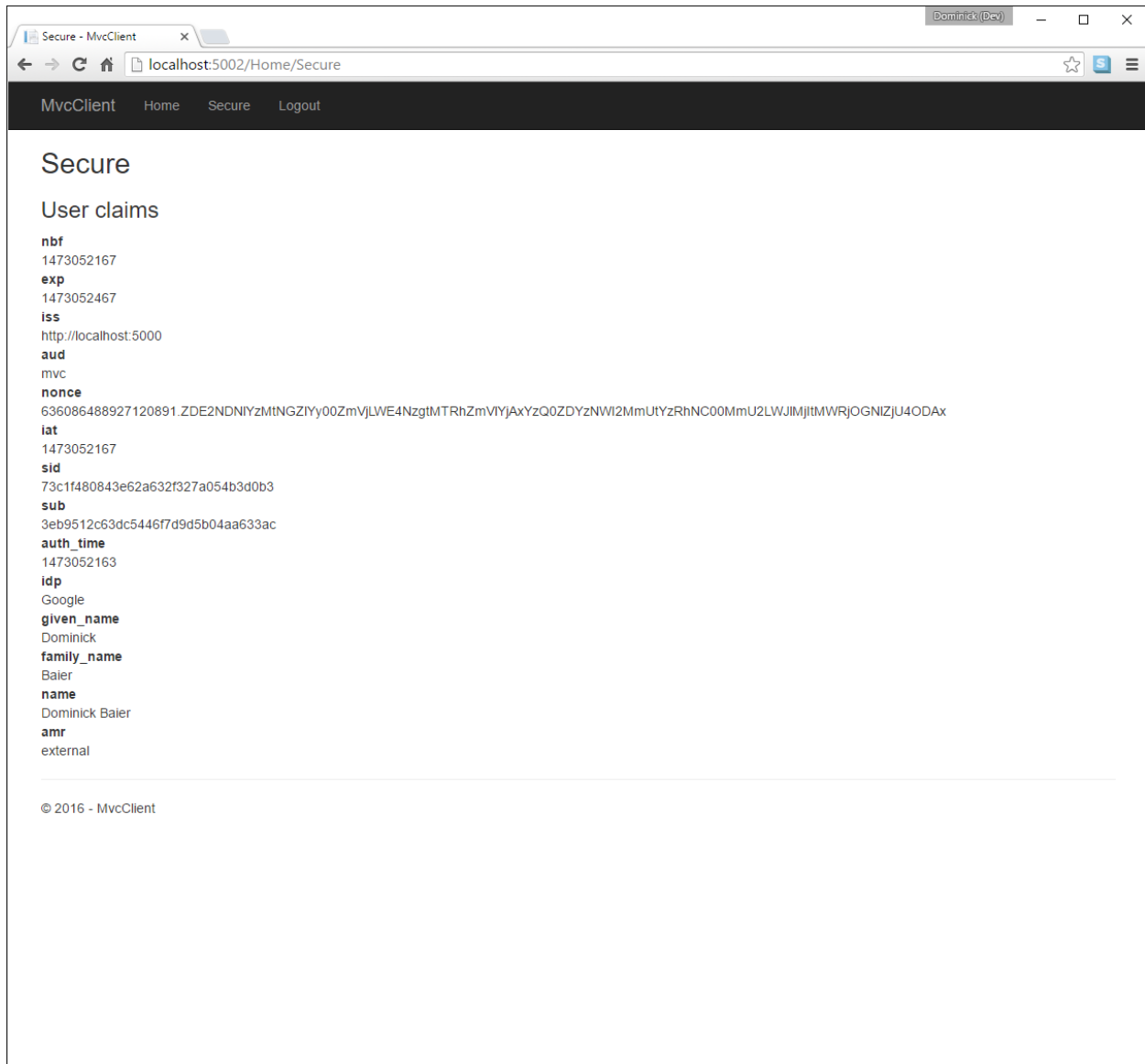
12.2 Further experiments

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

Add the OpenId Connect handler to DI:

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

```
        options.ClientId = "434483408261-55tc8n0cs4ff1fe21ea8df2o443v2iuc.apps.
→googleusercontent.com";
        options.ClientSecret = "3gcoTrEDPPJ0ukn_aYYT6PWo";
    })
    .AddOpenIdConnect("oidc", "OpenID Connect", options =>
    {
        options.SignInScheme = IdentityServerConstants.
→ExternalCookieAuthenticationScheme;
        options.SignOutScheme = IdentityServerConstants.SignoutScheme;

        options.Authority = "https://demo.identityserver.io/";
        options.ClientId = "implicit";

        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.

Switching to Hybrid Flow and adding API Access back

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.

In the previous quickstart we used the OpenID Connect implicit flow. In the implicit flow all tokens are transmitted via the browser, which is totally fine for the identity token. Now we also want to request an access token.

Access tokens are a bit more sensitive than identity tokens, and we don't want to expose them to the "outside" world if not needed. OpenID Connect includes a flow called "Hybrid Flow" which gives us the best of both worlds, the identity token is transmitted via the browser channel, so the client can validate it before doing any more work. And if validation is successful, the client opens a back-channel to the token service to retrieve the access token.

13.1 Modifying the client configuration

There are not many modifications necessary. First we want to allow the client to use the hybrid flow, in addition we also want the client to allow doing server to server API calls which are not in the context of a user (this is very similar to our client credentials quickstart). This is expressed using the `AllowedGrantTypes` property.

Next we need to add a client secret. This will be used to retrieve the access token on the back channel.

And finally, we also give the client access to the `offline_access` scope - this allows requesting refresh tokens for long lived API access:

```
new Client
{
    ClientId = "mvc",
    ClientName = "MVC Client",
    AllowedGrantTypes = GrantTypes.HybridAndClientCredentials,

    ClientSecrets =
    {
```

```
        new Secret("secret".Sha256())
    },

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

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

13.2 Modifying the MVC client

The modifications at the MVC client are also minimal - the ASP.NET Core OpenID Connect handler has built-in support for the hybrid flow, so we only need to change some configuration values.

We configure the `ClientSecret` to match the secret at IdentityServer. Add the `offline_access` and `api1` scopes, and set the `ResponseType` to `code id_token` (which basically means "use hybrid flow"). To keep the `website` claim in our mvc client identity we need to explicitly map the claim using ClaimActions.

```
.AddOpenIdConnect("oidc", options =>
{
    options.SignInScheme = "Cookies";

    options.Authority = "http://localhost:5000";
    options.RequireHttpsMetadata = false;

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

    options.SaveTokens = true;
    options.GetClaimsFromUserInfoEndpoint = true;

    options.Scope.Add("api1");
    options.Scope.Add("offline_access");
    options.ClaimActions.MapJsonKey("website", "website");
});
```

When you run the MVC client, there will be no big differences, besides that the consent screen now asks you for the additional API and offline access scope.

13.3 Using the access token

The OpenID Connect middleware saves the tokens (identity, access and refresh in our case) automatically for you. That's what the `SaveTokens` setting does.

Technically the tokens are stored inside the properties section of the cookie. The easiest way to access them is by using extension methods from the `Microsoft.AspNetCore.Authentication` namespace.

For example on your claims view:

```
<dt>access token</dt>
<dd>@await ViewContext.HttpContext.GetTokenAsync("access_token")</dd>

<dt>refresh token</dt>
<dd>@await ViewContext.HttpContext.GetTokenAsync("refresh_token")</dd>
```

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

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

    var client = new HttpClient();
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer
→", accessToken);
    var content = await client.GetStringAsync("http://localhost:5001/identity");

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

Using ASP.NET Core Identity

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

This quickstart assumes you've been through all of the prior quickstarts. The approach this quickstart takes to using ASP.NET Identity is to create a new project from the ASP.NET Identity template in Visual Studio. This new project will replace the prior IdentityServer project we built up from scratch in the previous quickstarts. All the other projects in this solution (for the clients and the API) will remain the same.

14.1 New Project for ASP.NET Identity

The first step is to add a new project for ASP.NET Identity to your solution. Given that a lot of code is required for ASP.NET Identity, it makes sense to use the template from Visual Studio. You will eventually delete the old project for IdentityServer (assuming you were following the other quickstarts), but there are several items that you will need to migrate over (or rewrite from scratch as described in the prior quickstarts).

Start by creating a new "ASP.NET Core Web Application" project.
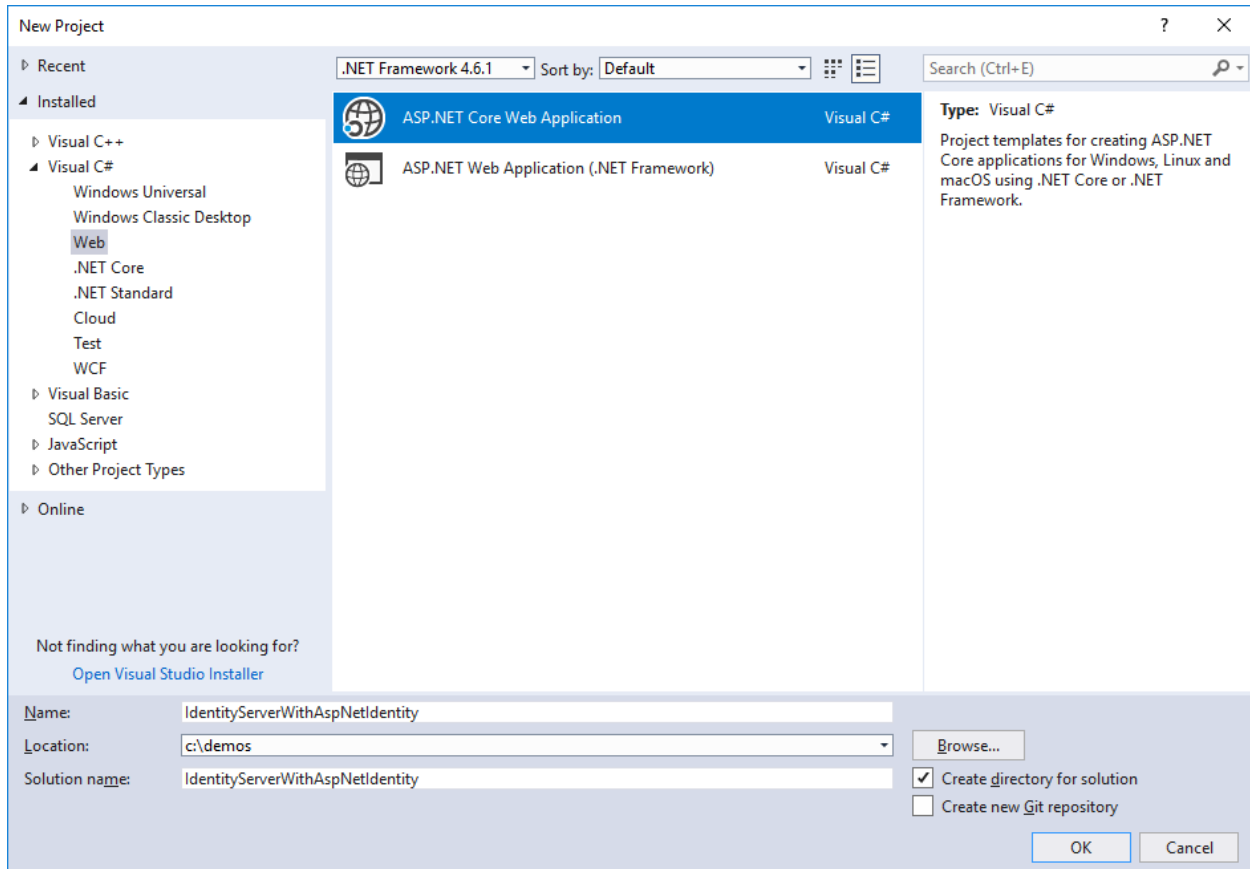
Then select the "Web Application Template (Model-View-Controller)" option.
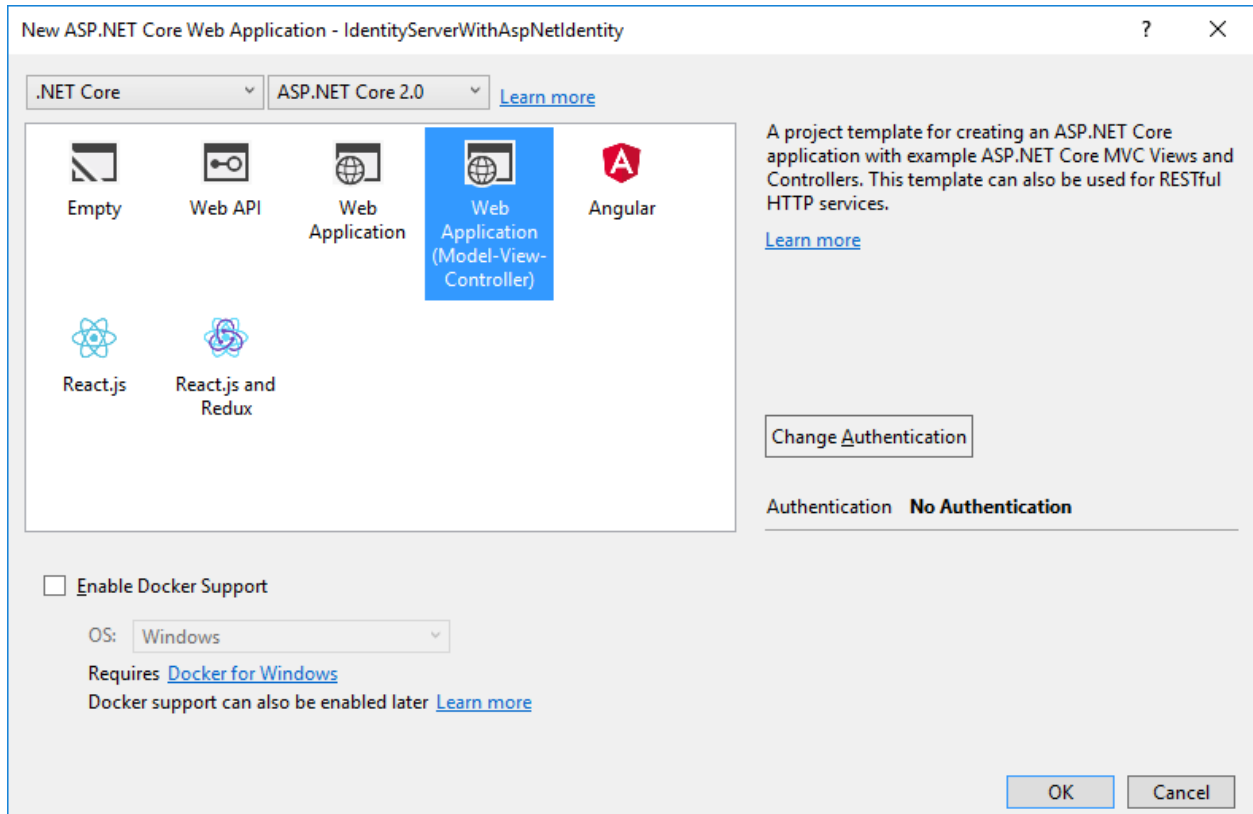
Then click the "Change Authentication" button, and choose "Individual User Accounts" (which means to use ASP.NET Identity):
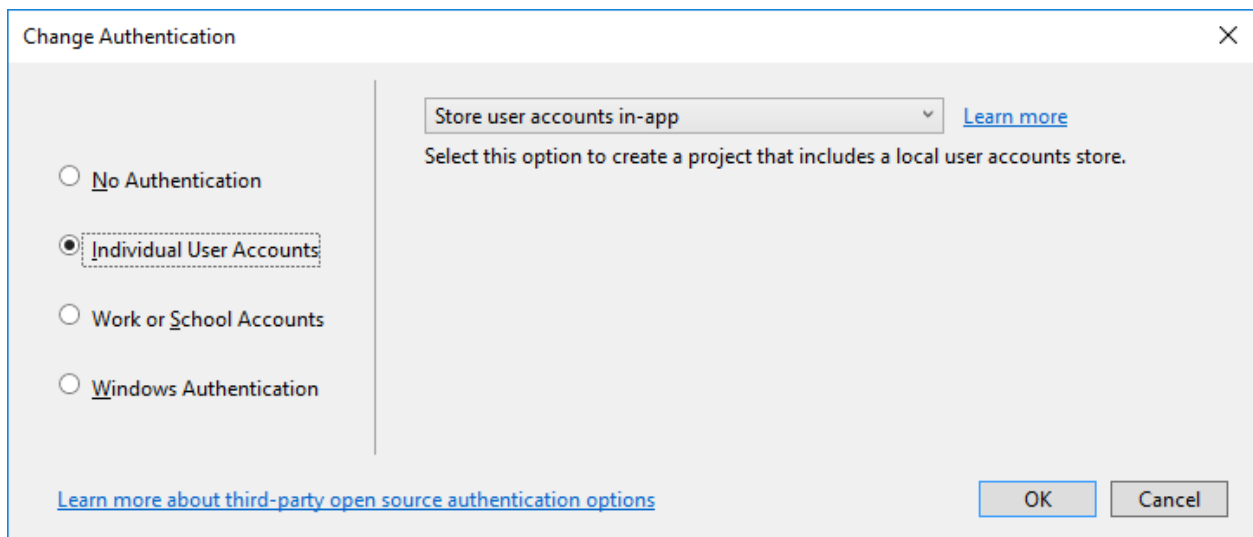
Finally, your new project dialog should look something like this. Once it does, click "OK" to create the project.
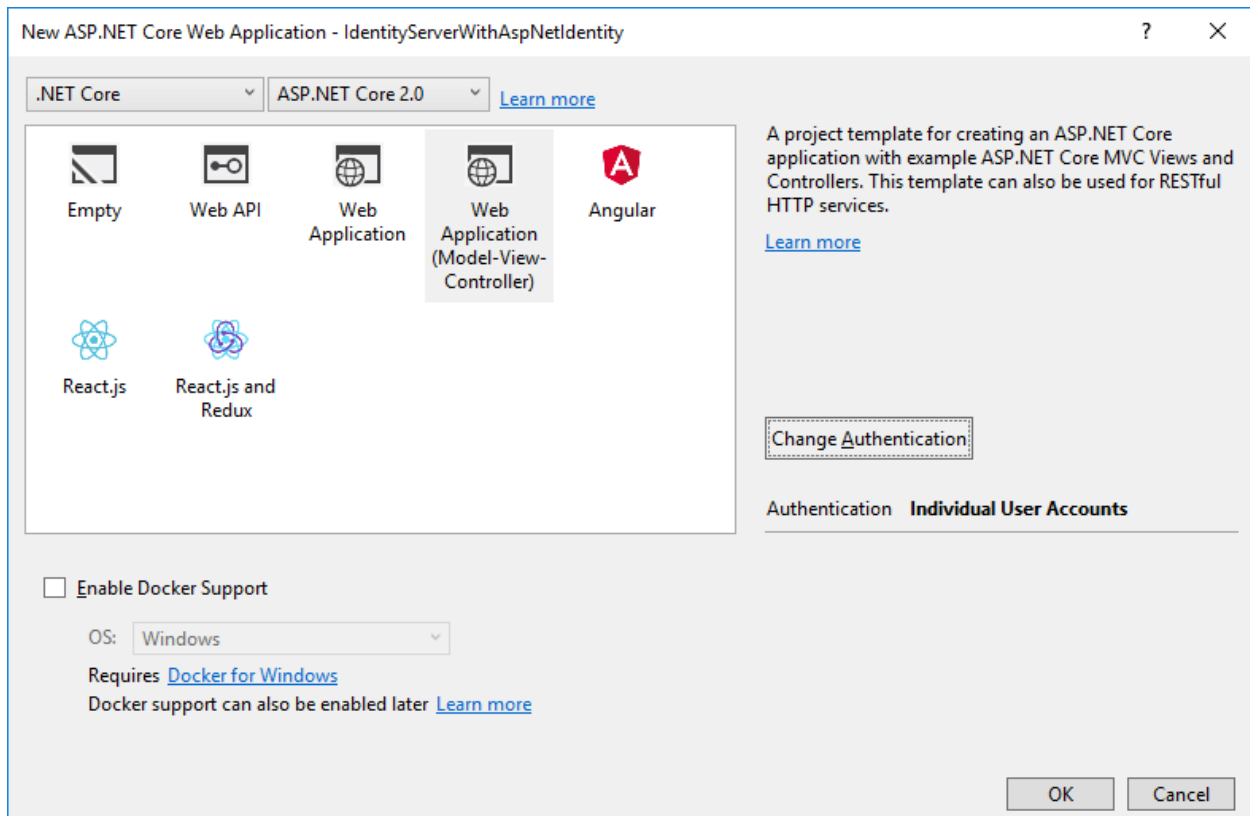
14.2 Modify hosting

Don't forget to modify the hosting (as described here) to run on port 5000. This is important so the existing clients and api projects will continue to work.

14.3 Add IdentityServer packages

Add the `IdentityServer4.AspNetIdentity` NuGet package. This depends on the `IdentityServer4` package, so that's automatically added as a transitive dependency.
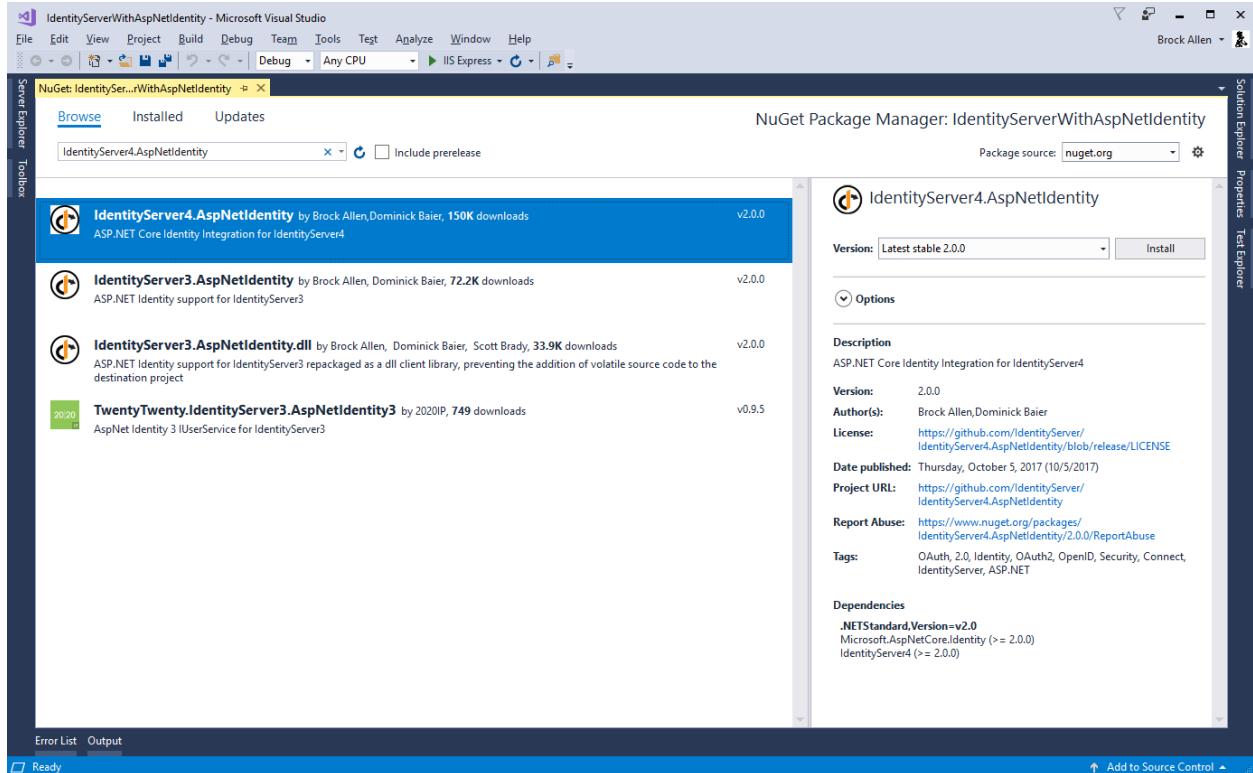
14.4 Scopes and Clients Configuration

Despite this being a new project for IdentityServer, we still need the same scope and client configuration as the prior quickstarts. Copy the configuration class (in Config.cs) you used for the previous quickstarts into this new project.

One change to the configuration that is necessary (for now) is to disable consent for the MVC client. We've not yet copied over the consent code from the prior IdentityServer project, so for now make this one modification to the MVC client and set `RequireConsent=false`:

```
new Client
{
    ClientId = "mvc",
    ClientName = "MVC Client",
    AllowedGrantTypes = GrantTypes.HybridAndClientCredentials,

    RequireConsent = false,

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

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

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
```

(continues on next page)

```
            IdentityServerConstants.StandardScopes.Profile,
            "api1"
        },
        AllowOfflineAccess = true
}
```

14.5 Configure IdentityServer

As before, IdentityServer needs to be configured in both `ConfigureServices` and in `Configure` in *Startup.cs*.

ConfigureServices

This shows both the template code generated for ASP.NET Identity, plus the additions needed for IdentityServer (at the end). In the previous quickstarts, the `AddTestUsers` extension method was used to register the users, but in this situation we replace that extension method with `AddAspNetIdentity` to use the ASP.NET Identity users instead. The `AddAspNetIdentity` extension method requires a generic parameter which is your ASP.NET Identity user type (the same one needed in the `AddIdentity` method from the template).

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    // Add application services.
    services.AddTransient<IEmailSender, EmailSender>();

    services.AddMvc();

    // configure identity server with in-memory stores, keys, clients and scopes
    services.AddIdentityServer()
        .AddDeveloperSigningCredential()
        .AddInMemoryPersistedGrants()
        .AddInMemoryIdentityResources(Config.GetIdentityResources())
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients())
        .AddAspNetIdentity<ApplicationUser>();
}
```

Note: It's important when using ASP.NET Identity that IdentityServer be registered *after* ASP.NET Identity in the DI system because IdentityServer is overwriting some configuration from ASP.NET Identity.

Configure

This shows both the template code generated for ASP.NET Identity, plus the call to `UseIdentityServer` which replaces the call to `UseAuthentication`.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
```

```
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    // app.UseAuthentication(); // not needed, since UseIdentityServer adds the␣
↪authentication middleware
    app.UseIdentityServer();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```
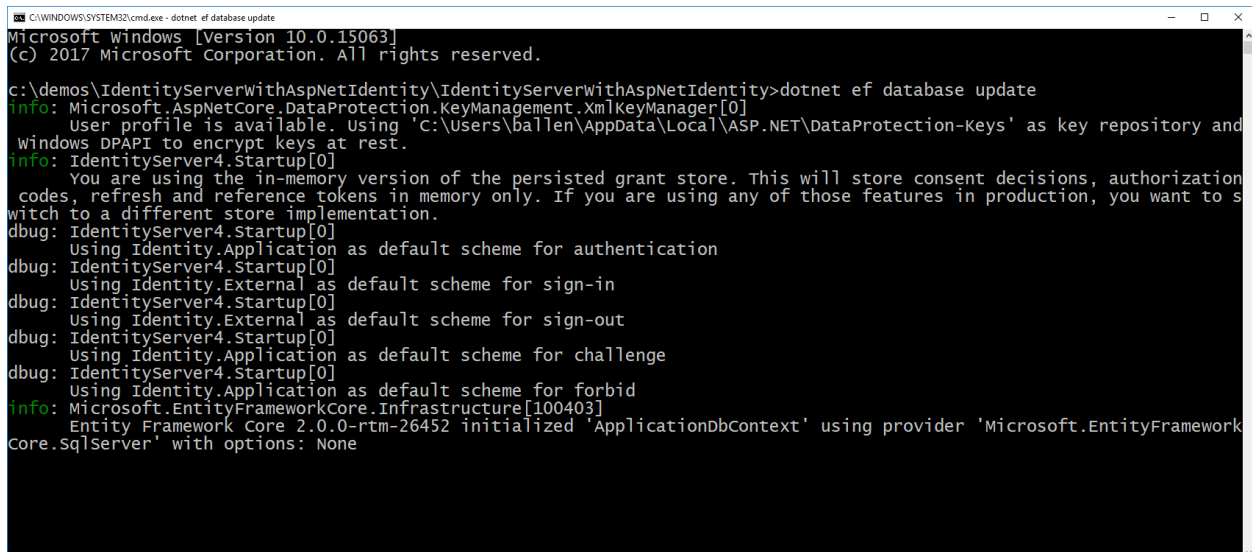
14.6 Creating the user database

Given that this is a new ASP.NET Identity project, you will need to create the database. You can do this by running a command prompt from the project directory and running `dotnet ef database update -c ApplicationDbContext`, like this:

14.7 Creating a user

At this point, you should be able to run the project and create/register a user in the database. Launch the application, and from the home page click the "Register" link:
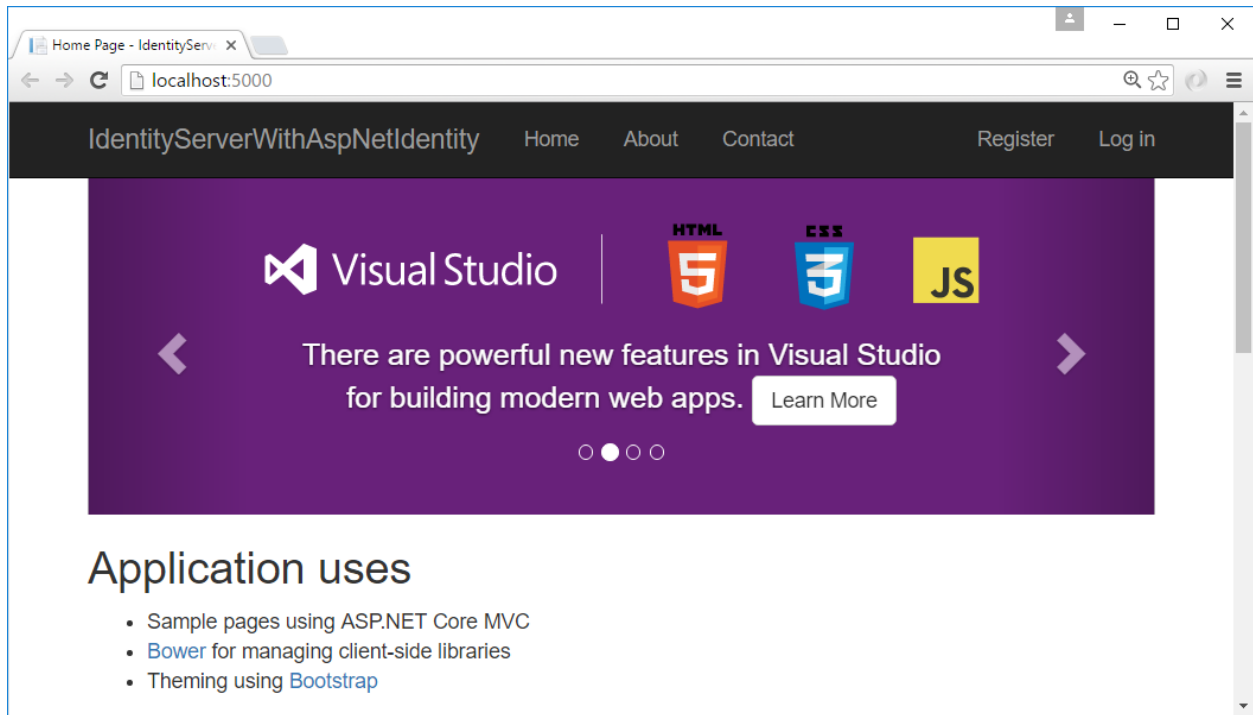
And on the register page create a new user account:

Now that you have a user account, you should be able to login, use the clients, and invoke the APIs.

14.8 Logging in with the MVC client

Launch the MVC client application, and you should be able to click the "Secure" link to get logged in.
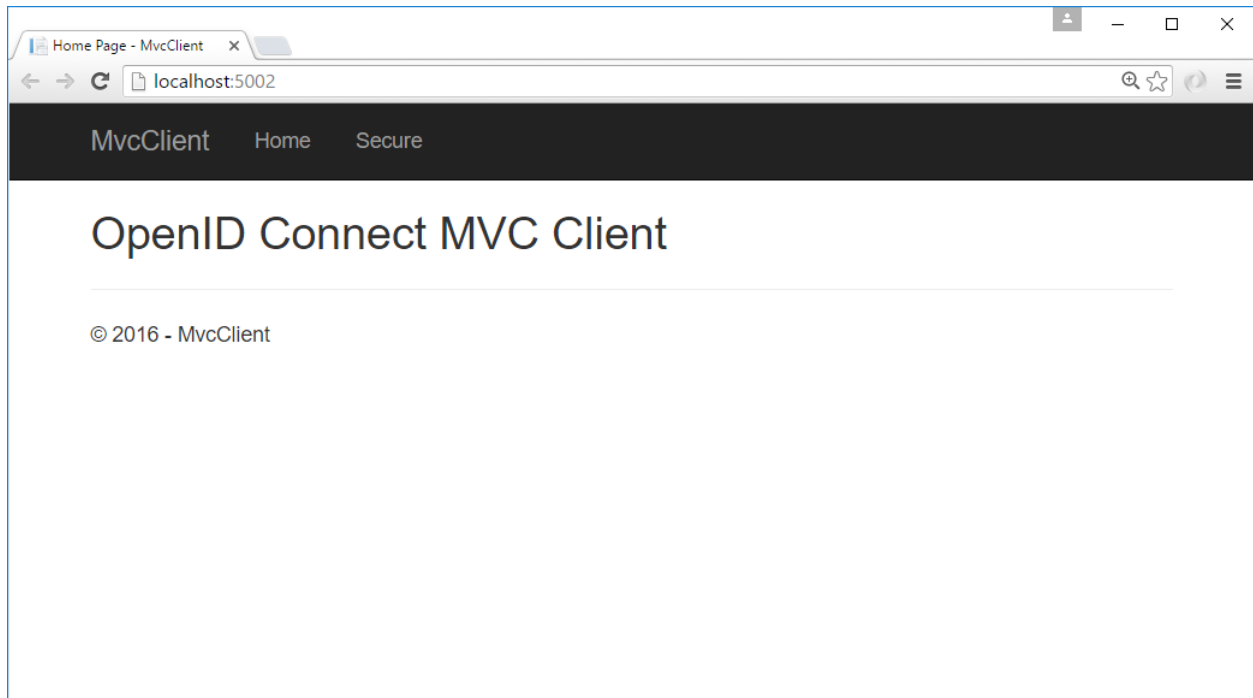
You should be redirected to the ASP.NET Identity login page. Login with your newly created user:
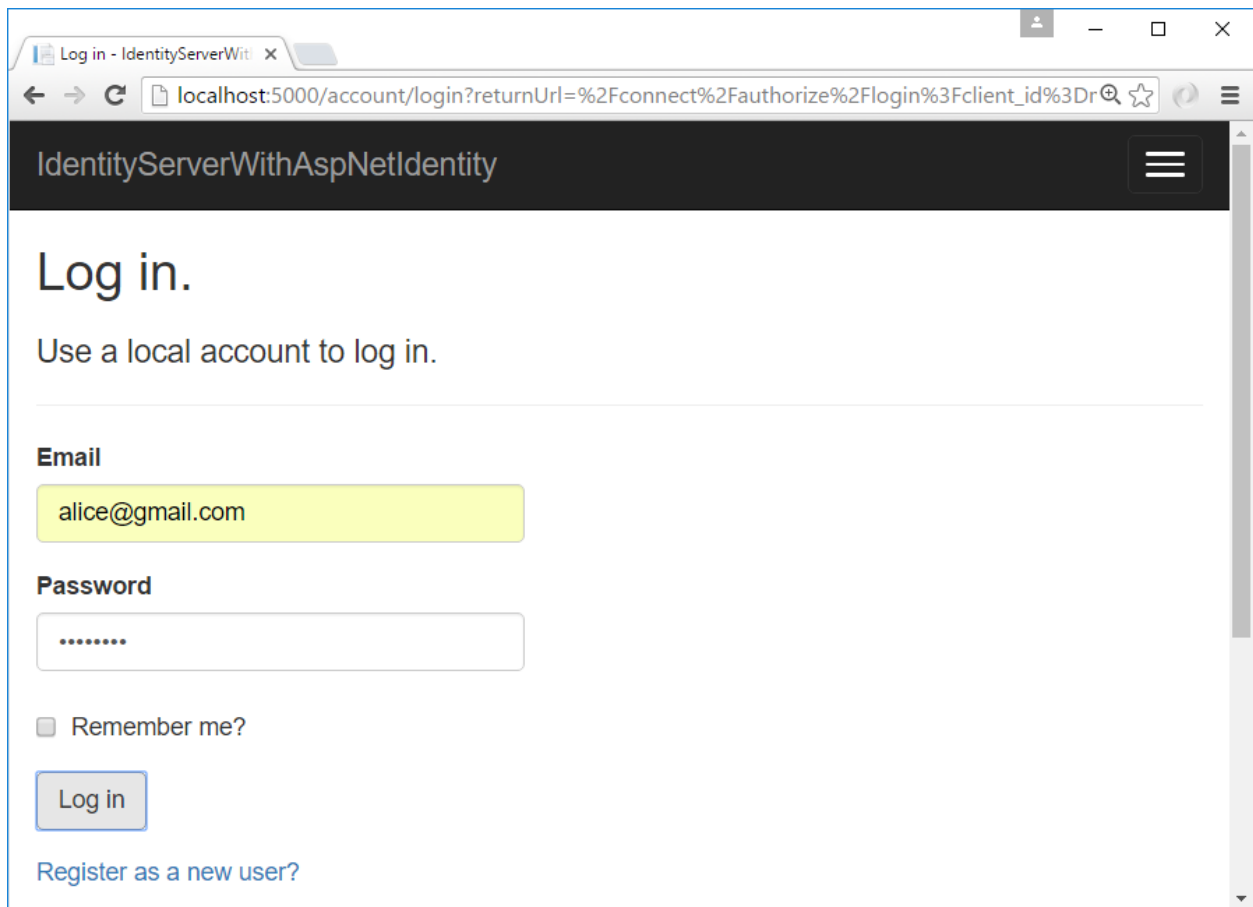
After login you should skip the consent page (given the change we made above), and be immediately redirected back

to the MVC client application where your user's claims should be listed.

You should also be able to click "Call API using application identity" to invoke the API on behalf of the user:
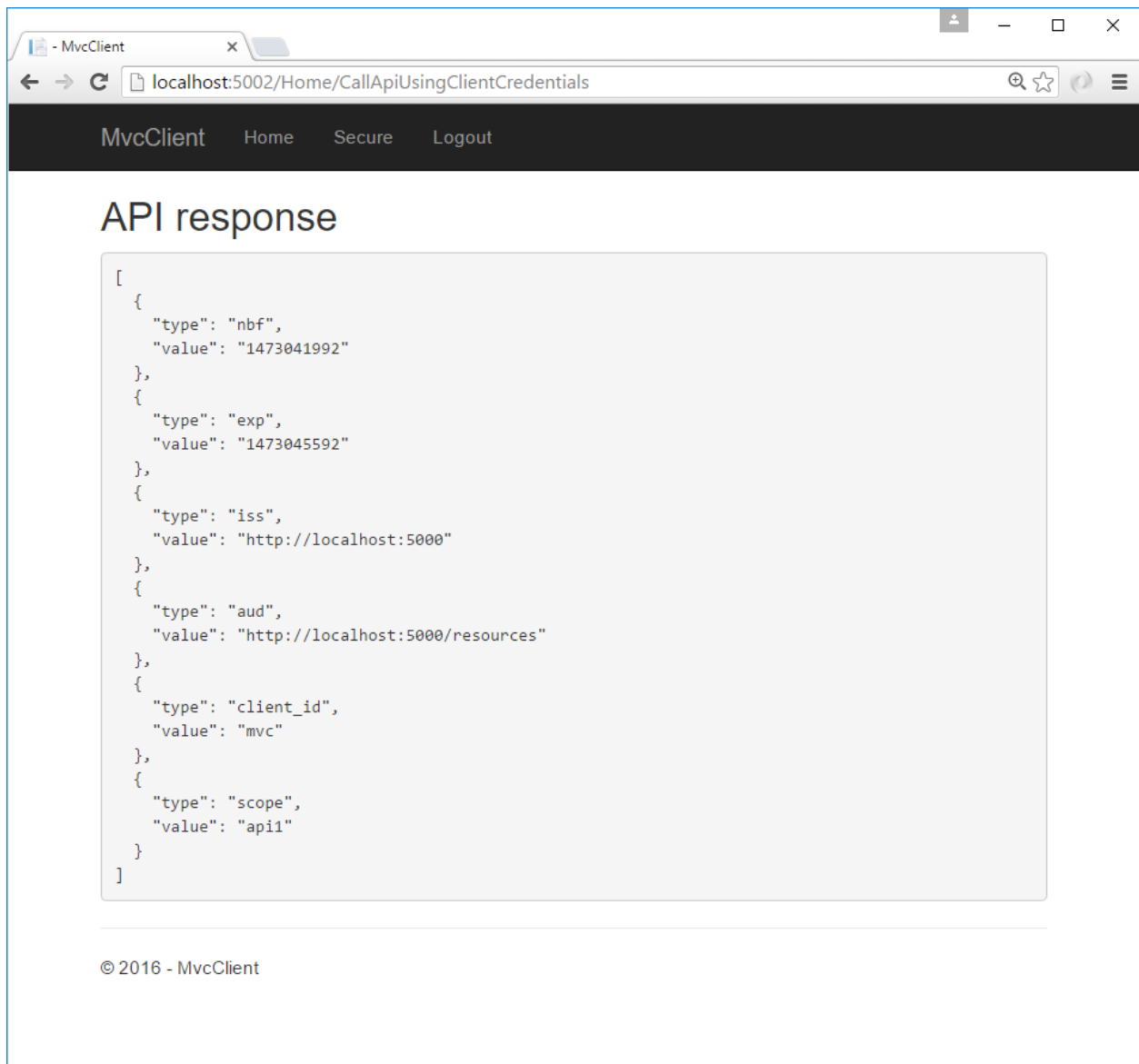
And now you've logged in with a user from ASP.NET Identity.

14.9 What's Next?

The prior quickstart project for IdentityServer provided a consent page, an error page, and a logout page. The code for these missing pieces can simply be copied over from the prior quickstart project into this one. Once you've done that, then you can finally delete/remove the old IdentityServer project. Also, once you've done this don't forget to re-enable the `RequireConsent=true` flag on the MVC client configuration.

The sample code for this quickstart has already done these steps for you, so you can get started quickly with all of these features. Enjoy!

Adding a JavaScript client

This quickstart will show how to build a JavaScript client application. The user will login to IdentityServer, invoke the web API with an access token issued by IdentityServer, and logout of IdentityServer.

15.1 New Project for the JavaScript client

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

Create a new ASP.NET Core web application:
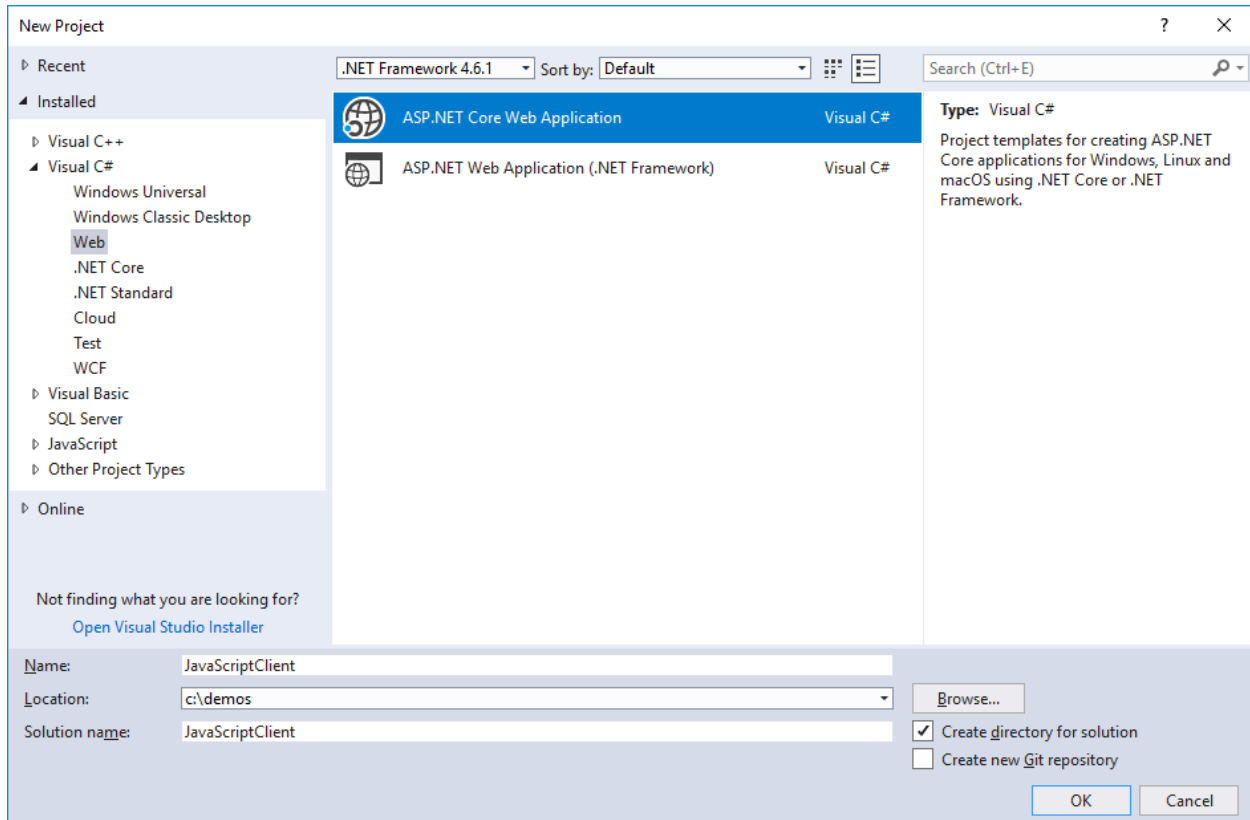
Choose the "Empty" template:
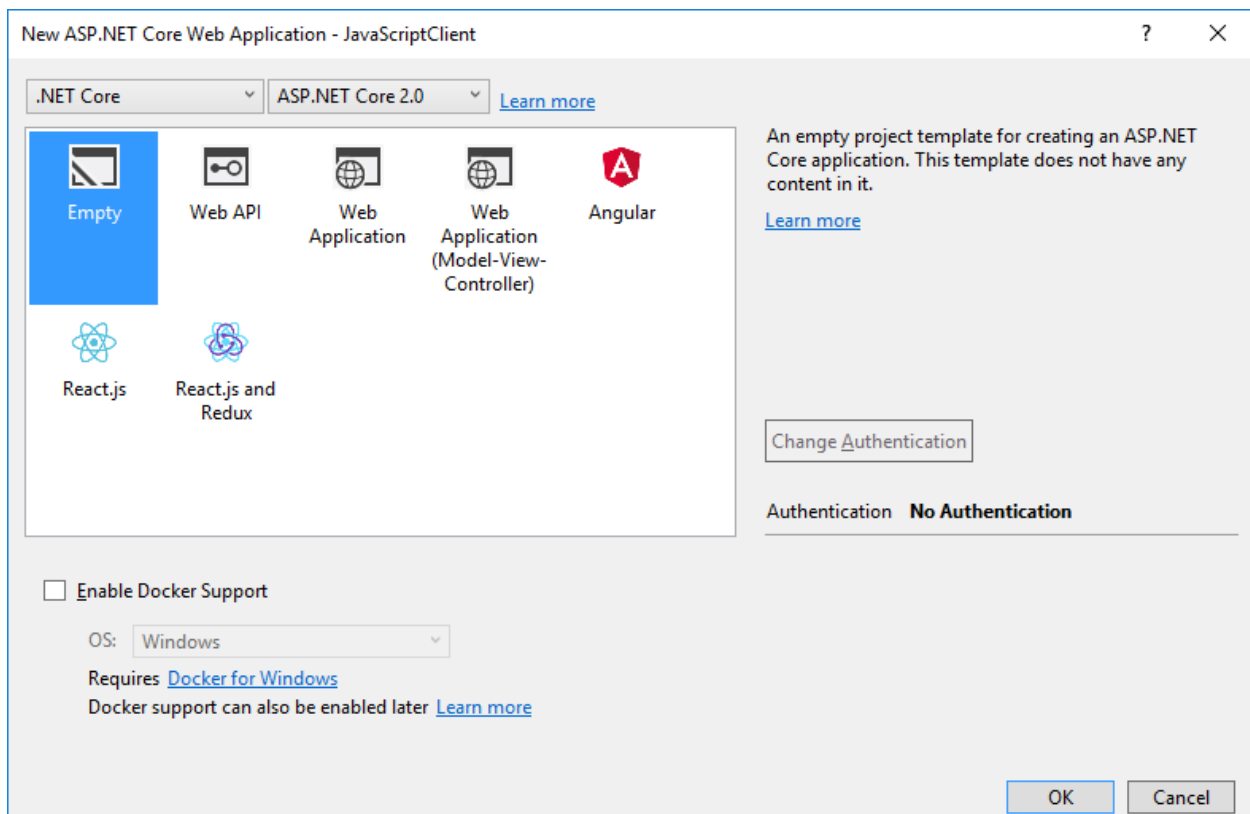
Click the "OK" button to create the project.

15.2 Modify hosting

Modify the hosting (as described here) to run on port 5003.

15.3 Add the static file middleware

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

Register the static file middleware in *Startup.cs* in the `Configure` method:

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

This middleware will now serve up static files from the application's *~/wwwroot* folder. This is where we will put our HTML and JavaScript files.

15.4 Reference oidc-client

In the MVC project, we used a library to handle the OpenID Connect protocol. In this project we need a similar library, except one that works in JavaScript and is designed to run in the browser. The oidc-client library is one such library. It is available via NPM, Bower, as well as a direct download from github.

NPM

If you want to use NPM to download *oidc-client*, then follow these steps:

Add a new NPM package file to your project and name it *package.json*:
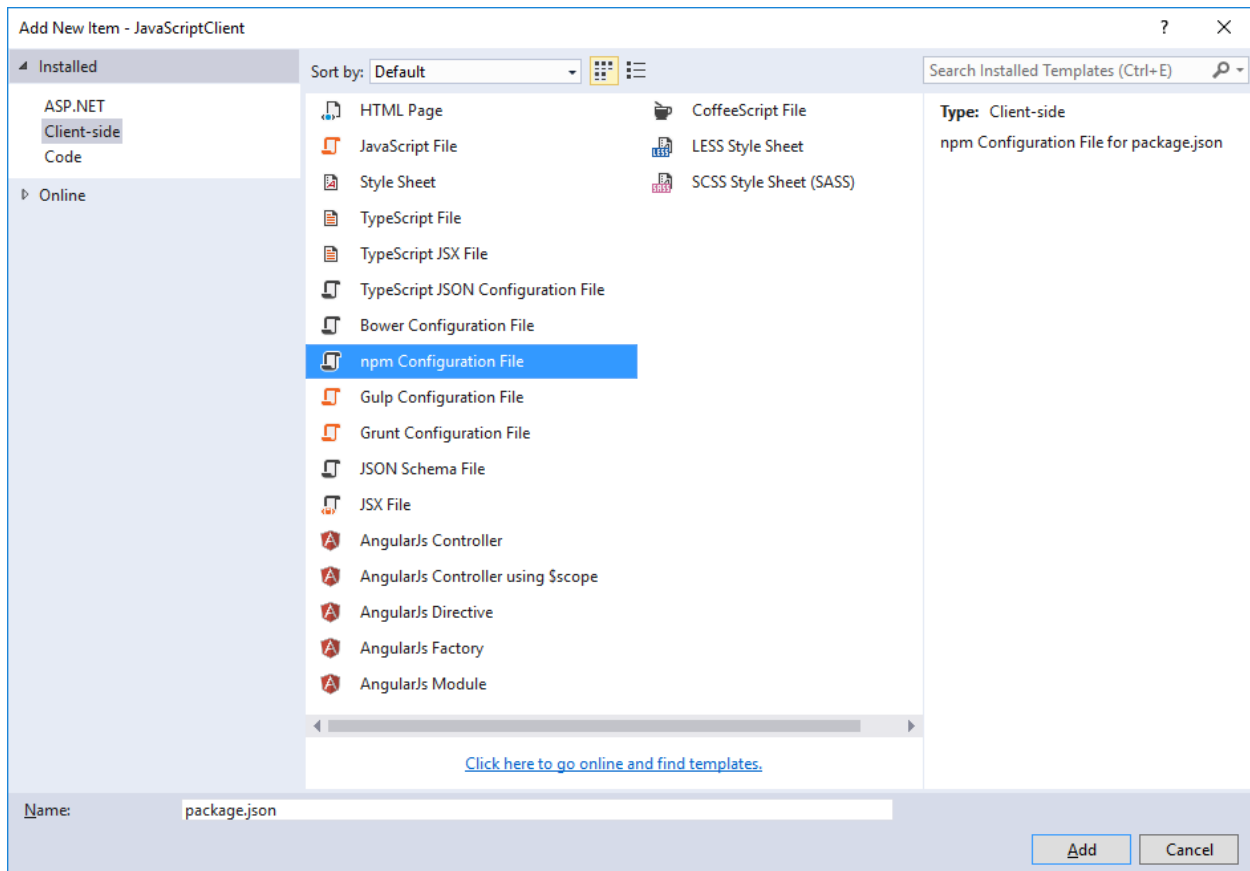
In *package.json* add a `dependency` to `oidc-client`:

```
"dependencies": {
  "oidc-client": "1.4.1"
}
```

Once you have saved this file, Visual Studio should automatically restore these packages into a folder called *node_modules*:
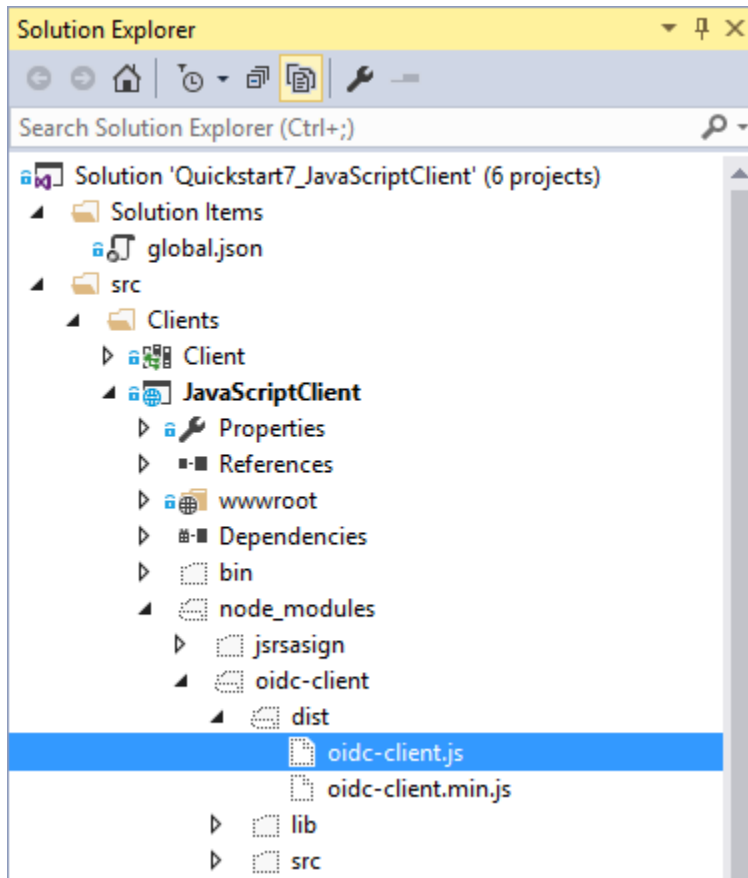
Locate the file called *oidc-client.js* in the *~/node_modules/oidc-client/dist* folder and copy it into your application's *~/wwwroot* folder. There are more sophisticated ways of copying your NPM packages into *~/wwwroot*, but those techniques are beyond the scope of this quickstart.

15.5 Add your HTML and JavaScript files

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

index.html

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

It should look like this:

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

(continues on next page)

```
    <button id="api">Call API</button>
    <button id="logout">Logout</button>

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

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

app.js

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

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

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

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

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

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

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

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

```
mgr.getUser().then(function (user) {
    if (user) {
        log("User logged in", user.profile);
```

```
    }
    else {
        log("User not logged in");
    }
});
```

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

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

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

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

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

See: *Protecting an API using Client Credentials <http://docs.identityserver.io/en/release/quickstarts/1_client_credentials.html>* for information on how to create the api used in the code above.

callback.html

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

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <script src="oidc-client.js"></script>
    <script>
        new Oidc.UserManager().signinRedirectCallback().then(function () {
            window.location = "index.html";
        }).catch(function (e) {
            console.error(e);
```

```
        });
    </script>
</body>
</html>
```

15.6 Add a client registration to IdentityServer for the JavaScript client

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

```
// JavaScript Client
new Client
{
    ClientId = "js",
    ClientName = "JavaScript Client",
    AllowedGrantTypes = GrantTypes.Implicit,
    AllowAccessTokensViaBrowser = true,

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

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

15.7 Allowing Ajax calls to the Web API with CORS

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

Configure CORS

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

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvcCore()
        .AddAuthorization()
        .AddJsonFormatters();

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

```
            options.RequireHttpsMetadata = false;

            options.ApiName = "api1";
        });

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

Add the CORS middleware to the pipeline in `Configure`:

```
public void Configure(IApplicationBuilder app)
{
    app.UseCors("default");

    app.UseAuthentication();

    app.UseMvc();
}
```

15.8 Run the JavaScript application

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

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

And click the "API" button to invoke the web API:

And finally click "Logout" to sign the user out.

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

Using EntityFramework Core for configuration and operational data

IdentityServer is designed for extensibility, and one of the extensibility points is the storage mechanism used for data that IdentityServer needs. This quickstart shows how to configure IdentityServer to use EntityFramework (EF) as the storage mechanism for this data (rather than using the in-memory implementations we had been using up until now).

Note: In addition to manually configuring EF support, there is also an IdentityServer template to create a new project with EF support. Use `dotnet new is4ef` to create it. See here for more information.

16.1 IdentityServer4.EntityFramework

There are two types of data that we are moving to the database. The first is the configuration data (resources and clients). The second is operational data that IdentityServer produces as it's being used (tokens, codes, and consents). These stores are modeled with interfaces, and we provide an EF implementation of these interfaces in the *Identity-Server4.EntityFramework* Nuget package.

Get started by adding a reference to the *IdentityServer4.EntityFramework* Nuget package the IdentityServer project.

16.2 Using SqlServer

Given EF's flexibility, you can then use any EF-supported database. For this quickstart we will use the LocalDb version of SqlServer that comes with Visual Studio.

16.3 Database Schema Changes and Using EF Migrations

The *IdentityServer4.EntityFramework* 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* and upgrade over time, you are responsible for your own database schema and changes necessary to that schema as the entity classes change. One approach for managing those changes is to use EF migrations, and this quickstart will show how that can be done. If migrations are not your preference, then you can manage the schema changes in any way you see fit.

Note: SQL scripts for SqlServer are maintained for the entities in *IdentityServer4.EntityFramework.Storage*. They are located here.

16.4 EF Tooling for Migrations

In addition to tracking schema changes with EF migrations, we will also use it to create the initial schema in the database. This requires the use of the EF Core tooling (more details here). We will add those now, and unfortunately

this must be done by hand-editing your *.csproj* file. To edit the *.csproj* by right-click the project and select "Edit projectname.csproj":

Note: Depending on how you created your initial project for the IdentityServer host, you might already have these tools configured in your *csproj* file. If they are, you can skip to the next section.

And then add the below snippet before the end *</Project>* element:

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
→Version="2.0.0" />
</ItemGroup>
```

It should look something like this:

```
QuickstartIdentityServer.csproj  ⊕ ✕
  1  ⊟<Project Sdk="Microsoft.NET.Sdk.Web">
  2
  3  ⊟  <PropertyGroup>
  4       <TargetFramework>netcoreapp2.0</TargetFramework>
  5     </PropertyGroup>
  6
  7  ⊟  <ItemGroup>
  8       <PackageReference Include="IdentityServer4" Version="2.0.0-rc1-update1" />
  9       <PackageReference Include="IdentityServer4.EntityFramework" Version="2.0.0-rc1" />
 10       <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
 11     </ItemGroup>
 12
 13  ⊟  <ItemGroup>
 14       <Folder Include="wwwroot\" />
 15     </ItemGroup>
 16
 17  ⊟  <ItemGroup>
 18       <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
 19     </ItemGroup>
 20
 21   </Project>
 22
```

Save and close the file. To test that you have the tools properly installed, you can open a command shell in the same directory as the project and run *dotnet ef*. It should look like this:

```
C:\WINDOWS\SYSTEM32\cmd.exe                                                          –  ☐  ✕

C:\ballen\github\identity\IdSvr4\Samples\Quickstarts\8_EntityFrameworkStorage\src\QuickstartIdentityServer>dotnet ef

                 ---==/    \\
              |    |  \\   \\
          |  /   |  |   |  \\   \\\\
          | |    |  |   |    \\   \\\\\\
          | |    |  |   |      \\   \\\\\\

Entity Framework Core .NET Command Line Tools 2.0.0-rtm-26452

Usage: dotnet ef [options] [command]

Options:
  --version         Show version information
  -h|--help         Show help information
  -v|--verbose      Show verbose output.
  --no-color        Don't colorize output.
  --prefix-output   Prefix output with level.

Commands:
  database    Commands to manage the database.
  dbcontext   Commands to manage DbContext types.
  migrations  Commands to manage migrations.

Use "dotnet ef [command] --help" for more information about a command.

C:\ballen\github\identity\IdSvr4\Samples\Quickstarts\8_EntityFrameworkStorage\src\QuickstartIdentityServer>
```

16.5 Configuring the stores

The next step is to replace the current calls to `AddInMemoryClients`, `AddInMemoryIdentityResources`, and `AddInMemoryApiResources` in the `ConfigureServices` method in *Startup.cs*. We will replace them with this code:

```
const string connectionString = @"Data Source=(LocalDb)\MSSQLLocalDB;
↪database=IdentityServer4.Quickstart.EntityFramework-2.0.0;trusted_connection=yes;";
var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

// configure identity server with in-memory stores, keys, clients and scopes
```

```
services.AddIdentityServer()
    .AddDeveloperSigningCredential()
    .AddTestUsers(Config.GetUsers())
    // this adds the config data from DB (clients, resources)
    .AddConfigurationStore(options =>
    {
        options.ConfigureDbContext = builder =>
            builder.UseSqlServer(connectionString,
                sql => sql.MigrationsAssembly(migrationsAssembly));
    })
    // this adds the operational data from DB (codes, tokens, consents)
    .AddOperationalStore(options =>
    {
        options.ConfigureDbContext = builder =>
            builder.UseSqlServer(connectionString,
                sql => sql.MigrationsAssembly(migrationsAssembly));

        // this enables automatic token cleanup. this is optional.
        options.EnableTokenCleanup = true;
        options.TokenCleanupInterval = 30;
    });
```

You might need these namespaces added to the file:

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

The above code is hard-coding a connection string, which you should feel free to change if you wish. Also, the calls to `AddConfigurationStore` and `AddOperationalStore` are registering the EF-backed store implementations.

The "builder" callback function passed to these APIs is the EF mechanism to allow you to configure the `DbContextOptionsBuilder` for the `DbContext` for each of these two stores. This is how our `DbContext` classes can be configured with the database provider you want to use. In this case by calling `UseSqlServer` we are using SqlServer. As you can also tell, this is where the connection string is provided.

The "options" callback function in `UseSqlServer` is what configures the assembly where the EF migrations are defined. EF requires the use of migrations to define the schema for the database.

Note: It is the responsibility of your hosting application to define these migrations, as they are specific to your database and provider.

We'll add the migrations next.

16.6 Adding migrations

To create the migrations, open a command prompt in the IdentityServer project directory. In the command prompt run these two commands:

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

It should look something like this:

You should now see a *~/Data/Migrations/IdentityServer* folder in the project. This contains the code for the newly created migrations.

Note: If your database project is a separate class library and you fixed the error 'Unable to create an object of type '<your-name>DbContext'. Add an implementation of 'IDesignTimeDbContextFactory' to the project, or see https://go.microsoft.com/fwlink/?linkid=851728 for additional patterns supported at design time.' by adding implementations of the IDesignTimeDbContextFactory, you will also need implementations of the factory for both the PersistedGrantDbContext as well as the ConfigurationDbContext.

16.7 Initialize the database

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

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

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

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

(continues on next page)

```
        }

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

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

And then we can invoke this from the `Configure` method:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
↪ILoggerFactory loggerFactory)
{
    // 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 applicaion runs. Once your database is populated, consider removing the call to the API.

16.8 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.

Note: The code as it stands in this section still relies upon Config.cs and its fictitious users Alice and Bob. If your user list is short and static, an adjusted version of Config.cs may suffice, however you may wish to manage a larger and more fluid user list dynamically within a database. ASP.NET Identity is one option to consider, and a sample implementation of this solution is listed among the quickstarts in the next section.

Community quickstarts & samples

These samples are not maintained by the IdentityServer organization. The IdentityServer organization happily links to community samples, but can't make any guarantees about the samples. Please contact the authors directly.

17.1 Various ASP.NET Core security samples

https://github.com/leastprivilege/AspNetCoreSecuritySamples

17.2 IdentityServer4 EF and ASP.NET Identity

This sample combines the EF and ASP.NET Identity quickstarts (#6 and #8).

17.3 Co-hosting IdentityServer4 and a Web API

This sample shows how to host an API in the same host as the IdentityServer that is protecting the API.

https://github.com/brockallen/IdentityServerAndApi

17.4 IdentityServer4 samples for MongoDB

- IdentityServer4-mongo: Similar to Quickstart #8 EntityFramework configuration but using MongoDB for the configuration data.
- IdentityServer4-mongo-AspIdentity: More elaborated sample based on uses ASP.NET Identity for identity management that uses using MongoDB for the configuration data

https://github.com/souzartn/IdentityServer4.Samples.Mongo

17.5 Exchanging external tokens from Facebook, Google and Twitter

- Shows how to exchange an external authentication token to an identity server acesss token using an extension grant

https://github.com/waqaskhan540/IdentityServerExternalAuth

17.6 ASP.NET Core MVC RazorPages template for IdentityServer4 Quickstart UI

Razor Pages based QuickStart sample by Martin Fletcher.

17.7 .NET Core and ASP.NET Core "Platform" scenario

- Shows an interaction of trusted "internal" applications and "external" applications with .NET Core 2.0 and ASP.NET Core 2.0 applications

https://github.com/BenjaminAbt/Samples.AspNetCore-IdentityServer4

17.8 Securing a Node API with tokens from IdentityServer4 using JWKS

- Shows how to secure a Node (Express) API using the JWKS endpoint and RS256 algorithm from IdentityServer4.

- Provides an alternative to the NodeJsApi sample from IdentityServer4.Samples using higher quality - production ready modules.

https://github.com/lyphtec/idsvr4-node-jwks

Startup

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

18.1 Configuring services

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

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

Optionally you can pass in options into this call. See *here* for details on options.

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

18.2 Key material

- **AddSigningCredential** Adds a signing key service that provides the specified key material to the various token creation/validation services. You can pass in either an X509Certificate2, a SigningCredential or a reference to a certificate from the certificate store.

- **AddDeveloperSigningCredential** Creates temporary key material at startup time. This is for dev only scenarios when you don't have a certificate to use. The generated key will be persisted to the file system so it stays stable between server restarts (can be disabled by passing false). This addresses issues when the client/api metadata caches get out of sync during development.

- **AddValidationKey** Adds a key for validating tokens. They will be used by the internal token validator and will show up in the discovery document. You can pass in either an X509Certificate2, a SigningCredential or a reference to a certificate from the certificate store. This is useful for key roll-over scenarios.

18.3 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.

18.4 Test stores

The `TestUser` class models a user, their credentials, and claims in IdentityServer. Use of `TestUser` is simiar 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`.

18.5 Additional services

- **AddExtensionGrantValidator** Adds `IExtensionGrantValidator` implementation for use with extension grants.

- **AddSecretParser** Adds `ISecretParser` implementation for parsing client or API resource credentials.

- **AddSecretValidator** Adds `ISecretValidator` implementation for validating client or API resource credentials against a credential store.

- **AddResourceOwnerValidator** Adds `IResourceOwnerPasswordValidator` implementation for validating user credentials for the resource owner password credentials grant type.

- **AddProfileService** Adds `IProfileService` implementation for connecting to your *custom user profile store*. The `DefaultProfileService` class provides the default implementation which relies upon the authentication cookie as the only source of claims for issuing in tokens.

- **AddAuthorizeInteractionResponseGenerator** Adds `IAuthorizeInteractionResponseGenerator` implementation to customize logic at authorization endpoint for when a user must be shown a UI for error, login, consent, or any other custom page. The `AuthorizeInteractionResponseGenerator` class provides a default implementation, so consider deriving from this existing class if you need to augment the existing behavior.

- **AddCustomAuthorizeRequestValidator** Adds `ICustomAuthorizeRequestValidator` implementation to customize request parameter validation at the authorization endpoint.

- **AddCustomTokenRequestValidator** Adds `ICustomTokenRequestValidator` implementation to customize request parameter validation at the token endpoint.

- **AddRedirectUriValidator** Adds `IRedirectUriValidator` implementation to customize redirect URI validation.

- **AddAppAuthRedirectUriValidator** Adds a an "AppAuth" (OAuth 2.0 for Native Apps) compliant redirect URI validator (does strict validation but also allows http://127.0.0.1 with random port).

- **AddJwtBearerClientAuthentication** Adds support for client authentication using JWT bearer assertions.

18.6 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.

18.7 Configuring the pipeline

You need to add IdentityServer to the pipeline by calling:

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

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

There is no additional configuration for the middleware.

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

Defining Resources

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.

19.1 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`:

```
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:

```
public static IEnumerable<IdentityResource> GetIdentityResources()
{
```

(continues on next page)

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

19.2 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:

```
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.Profile(),
        customProfile
    };
}
```

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

19.3 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*:

```
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",
```

```csharp
            // 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 loaded by the *IProfileService* extensibility point.

Defining Clients

Clients represent applications that can request tokens from your identityserver.

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

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

---

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

20.1 Defining a client for server to server communication

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

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

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

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

20.2 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:

```
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"
    }
};
```

20.3 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):

```
var mvcClient = new Client
{
    ClientId = "mvc",
    ClientName = "MVC Client",
    ClientUri = "http://identityserver.io",

    AllowedGrantTypes = GrantTypes.Hybrid,
```

```csharp
    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"
    },
};
```

Sign-in

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

21.1 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.

21.2 Overriding cookie handler configuration

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

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

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

(continues on next page)

```
    {
        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`.

21.3 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*.

21.4 Login Workflow

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

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

21.5 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 the *interaction service*.

21.6 Issuing a cookie and Claims

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

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

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

Sign-in with External Identity Providers

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

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

22.1 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.

22.2 The role of cookies

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

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

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

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

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

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

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

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

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

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

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

22.3 Triggering the authentication handler

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

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

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

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

return Challenge(provider, props);
```

22.4 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("~/");
```

22.5 State, URL length, and ISecureDataFormat

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

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

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

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

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

```
        });
}
```

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 =>
        {
            // ...
        });
}
```

Windows Authentication

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.

23.1 Using Kestrel

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

```
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`:

```
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.

Sign-out

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.

24.1 Removing the authentication cookie

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

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

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

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

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

24.2 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.

24.3 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.

Sign-out of External Identity Providers

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

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

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

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

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

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

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

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

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

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

Federated Sign-out

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

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

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

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

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

Federation Gateway

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

This architecture has the following advantages

- your applications only need to know about the one token service (the gateway) and are shielded from all the details about connecting to the external provider(s). This also means that you can add or change those external providers without needing to update your applications.

- you control the gateway (as opposed to some external service provider) - this means you can make any changes to it and can protect your applications from changes those external providers might do to their own services.

- most external providers only support a fixed set of claims and claim types - having a gateway in the middle allows post-processing the response from the providers to transform/add/amend domain specific identity information.

- some providers don't support access tokens (e.g. social providers) - since the gateway knows about your APIs, it can issue access tokens based on the external identities.

- some providers charge by the number of applications you connect to them. The gateway acts as a single application to the external provider. Internally you can connect as many applications as you want.

- some providers use proprietary protocols or made proprietary modifications to standard protocols - with a gateway there is only one place you need to deal with that.

- forcing every authentication (internal or external) through one single place gives you tremendous flexibility with regards to identity mapping, providing a stable identity to all your applications and dealing with new requirements

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

27.1 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.

Consent

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*.

28.1 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.

28.2 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.

28.3 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.

28.4 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.

Protecting APIs

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

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

- JWT bearer authentication handler for ASP.NET Core
- JWT bearer authentication middleware for Katana
- IdentityServer authentication middleware for Katana
- jsonwebtoken for nodejs

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

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

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

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

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

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

29.1 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.

29.2 Supporting reference tokens

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

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

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

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

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

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

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

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

29.3 Validating scopes

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

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

Creating a global policy:

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

Composing a scope policy:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("myPolicy", builder =>
    {
        // require scope1
```

(continues on next page)

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

Deployment

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.

30.1 Typical architecture

Typically you will design your IdentityServer deployment for high availability:

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

30.2 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`.

30.3 Key material

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

30.4 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.

30.5 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.

Logging

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...

31.1 Setup for Serilog

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

31.1.1 ASP.NET Core 2.0+

For the following configuration you need the `Serilog.AspNetCore` and `Serilog.Sinks.Console` packages:

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

        Log.Logger = new LoggerConfiguration()
            .MinimumLevel.Debug()
            .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
            .MinimumLevel.Override("System", LogEventLevel.Warning)
            .MinimumLevel.Override("Microsoft.AspNetCore.Authentication",
→LogEventLevel.Information)
            .Enrich.FromLogContext()
            .WriteTo.Console(outputTemplate: "[{Timestamp:HH:mm:ss} {Level}]
→{SourceContext}{NewLine}{Message:lj}{NewLine}{Exception}{NewLine}", theme:
→AnsiConsoleTheme.Literate)
            .CreateLogger();

        BuildWebHost(args).Run();
    }

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

31.1.2 .NET Core 1.0, 1.1

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

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

        Log.Logger = new LoggerConfiguration()
            .MinimumLevel.Debug()
            .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
            .MinimumLevel.Override("System", LogEventLevel.Warning)
            .MinimumLevel.Override("Microsoft.AspNetCore.Authentication",
→LogEventLevel.Information)
            .Enrich.FromLogContext()
            .WriteTo.Console(outputTemplate: "[{Timestamp:HH:mm:ss} {Level}]
→{SourceContext}{NewLine}{Message:lj}{NewLine}{Exception}{NewLine}", theme:
→AnsiConsoleTheme.Literate)
            .CreateLogger();

        BuildWebHost(args).Run();
    }
```

(continues on next page)

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

Events

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.

32.1 Emitting events

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

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

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

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

```
    }
}
```

32.2 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.

32.3 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.

32.4 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; }
}
```

Cryptography, Keys and HTTPS

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

33.1 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*.

33.2 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.

33.3 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.

33.4 HTTPS

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

Grant Types

Grant types are a way to specify how a client wants to interact with IdentityServer. The OpenID Connect and OAuth 2 specs define the following grant types:

- Implicit

- Authorization code

- Hybrid

- Client credentials

- Resource owner password

- Device flow

- Refresh tokens

- Extension grants

You can specify which grant type a client can use via the `AllowedGrantTypes` property on the `Client` configuration.

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

```
Client.AllowedGrantTypes = GrantTypes.HybridAndClientCredentials;
```

You can also specify the grant types list manually:

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

If you want to transmit access tokens via the browser channel, you also need to allow that explicitly on the client configuration:

```
Client.AllowAccessTokensViaBrowser = true;
```

Note: For security reasons, not all grant type combinations are allowed. See below for more details.

For the remainder, the grant types are briefly described, and when you would use them. It is also recommended, that in addition you read the corresponding specs to get a better understanding of the differences.

34.1 Client credentials

This is the simplest grant type and is used for server to server communication - tokens are always requested on behalf of a client, not a user.

With this grant type you send a token request to the token endpoint, and get an access token back that represents the client. 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.

34.2 Resource owner password

The resource owner password grant type allows to request tokens on behalf of a user by sending the user's name and password to the token endpoint. This is so called "non-interactive" authentication and is generally not recommended.

There might be reasons for certain legacy or first-party integration scenarios, where this grant type is useful, but the general recommendation is to use an interactive flow like implicit or hybrid for user authentication instead.

See the Resource Owner Password Quick Start for a sample how to use it. You also need to provide code for the username/password validation which can be supplied by implementing the `IResourceOwnerPasswordValidator` interface. You can find more information about this interface *here*.

34.3 Implicit

The implicit grant type is optimized for browser-based applications. Either for user authentication-only (both server-side and JavaScript applications), or authentication and access token requests (JavaScript applications).

In the implicit flow, all tokens are transmitted via the browser, and advanced features like refresh tokens are thus not allowed.

This quickstart shows authentication for service-side web apps, and *this* shows JavaScript.

34.4 Authorization code

Authorization code flow was originally specified by OAuth 2, and provides a way to retrieve tokens on a back-channel as opposed to the browser front-channel. It also support client authentication.

While this grant type is supported on its own, it is generally recommended you combine that with identity tokens which turns it into the so called hybrid flow. Hybrid flow gives you important extra features like signed protocol responses.

34.5 Hybrid

Hybrid flow is a combination of the implicit and authorization code flow - it uses combinations of multiple grant types, most typically `code id_token`.

In hybrid flow the identity token is transmitted via the browser channel and contains the signed protocol response along with signatures for other artifacts like the authorization code. This mitigates a number of attacks that apply to the browser channel. After successful validation of the response, the back-channel is used to retrieve the access and refresh token.

This is the recommended flow for native applications that want to retrieve access tokens (and possibly refresh tokens as well) and is used for server-side web applications and native desktop/mobile applications.

See *this* quickstart for more information about using hybrid flow with MVC.

34.6 Device flow

Device flow is designed for browserless and input constrained devices, where the device is unable to securely capture user credentials. This flow outsources user authentication and consent to an external device (e.g. a smart phone).

This flow is typically used by IoT devices and can request both identity and API resources.

34.7 Refresh tokens

Refresh tokens allow gaining long lived access to APIs.

You typically want to keep the lifetime of access tokens as short as possible, but at the same time don't want to bother the user over and over again with doing a front-channel roundtrips to IdentityServer for requesting new ones.

Refresh tokens allow requesting new access tokens without user interaction. Every time the client refreshes a token it needs to make an (authenticated) back-channel call to IdentityServer. This allows checking if the refresh token is still valid, or has been revoked in the meantime.

Refresh tokens are supported in hybrid, authorization code, device flow and resource owner password flows. To request a refresh token, the client needs to include the `offline_access` scope in the token request (and must be authorized to request for that scope).

34.8 Extension grants

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

34.9 Incompatible grant types

Some grant type combinations are forbidden:

- Mixing implicit and authorization code or hybrid would allow a downgrade attack from the more secure code based flow to implicit.
- Same concern exists for allowing both authorization code and hybrid

Secrets

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.

35.1 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));
```

35.2 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.

35.3 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.:

```
var client = new Client
{
    ClientId = "client.jwt",
    ClientSecrets =
    {
        new Secret
        {
            Type = IdentityServerConstants.SecretTypes.X509CertificateBase64,
            Value =
→"MIIDATCCAe2gAwIBAgIQoHUYAquk9rBJcq8W+F0FAzAJBgUrDgMCHQUAMBIxEDAOBgNVBAMTB0RldlJvb3QwHhcNMTAwMTIwMj
→IdOQxbavmuPbhY7jX0IORu/
→GQiHjmhqWt8F4G7KGLhXLC1j7rXdDmxXRyVJBZBTEaSYukuX7zGeUXscdpgODLQVay/
→0hUGz54aDZPAhtBHaYbog+yH10sCXgV1Mxtzx3dGelA6pPwiAmXwFxjJ1HGsS/hdbt+vgXhdlzud3ZSfyI/
→TJAnFeKxsmbJUyqMfoBl1zFKG4MOvgHhBjekp+r8gYNGknMYu9JDFr1ue0wylaw9UwG8ZXAkYmYbn2wN/
→CpJl3gJgX42/9g87uLvtVAmz5L+rZQTlS1ibv54ScR2lcRpGQiQav/
→LAgMBAAGjXDBaMBMGA1UdJQQMMAoGCCsGAQUFBwMCMEMGA1UdAQQ8MDqAENIWANpX5DZ3bX3WvoDfy0GhFDASMRAwDgYDVQQDEw
→64q+Dk3z3Kt7w+grHqu5nYhsn7xQFAQUf3y2KcJnRdIEk0jrLM4vgIzYdXsoC6YO+9QnlkNqcN36Y8IpSVSTda6gRKvGXiAhu42
→WNMFOL+YzMXGt/nDHL/qRKsuXBOarIb++43DV3YnxGTx22llhOnPpuZ9/gnNY7KLjODaiEciKhaKqt/
→b57mTEz4jTF4kIg6BP03MUfDXeVlM1Qf1jB43G2QQ19n5lUiqTpmQkcfLfyci2uBZ8BkOhXr3Vk9HIk/
→xBXQ="
        }
    },

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

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

Extension Grants

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

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

```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>();
```

36.1 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:

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

(continues on next page)

```csharp
    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"
    }
}
```

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:

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

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

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

Refresh Tokens

Since access tokens have finite lifetimes, refresh tokens allow requesting new access tokens without user interaction.

Refresh tokens are supported for the following flows: authorization code, hybrid and resource owner password credential flow. The clients needs to be explicitly authorized to request refresh tokens by setting `AllowOfflineAccess` to `true`.

38.1 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:

```
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 middleware or use the identity server authentication

middleware which can validate both JWTs and reference tokens.

The introspection endpoint requires authentication - since the client of an introspection endpoint is an API, you configure the secret on the `ApiResource`:

```
var api = new ApiResource("api1")
{
    ApiSecrets = { new Secret("secret".Sha256()) }
}
```

See *here* for more information on how to configure the IdentityServer authentication middleware for APIs.

CORS

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.

40.1 Client-based CORS Configuration

One approach to configuring CORS is to use the `AllowedCorsOrigins` collection on the *client configuration*. Simply add the origin of the client to the collection and the default configuration in IdentityServer will consult these values to allow cross-origin calls from the origins.

Note: Be sure to use an origin (not a URL) when configuring CORS. For example: `https://foo:123/` is a URL, whereas `https://foo:123` is an origin.

This default CORS implementation will be in use if you are using either the "in-memory" or EF-based client configuration that we provide. If you define your own `IClientStore`, then you will need to implement your own custom CORS policy service (see below).

40.2 Custom Cors Policy Service

IdentityServer allows the hosting application to implement the `ICorsPolicyService` to completely control the CORS policy.

The single method to implement is: `Task<bool> IsOriginAllowedAsync(string origin)`. Return `true` if the *origin* is allowed, `false` otherwise.

Once implemented, simply register the implementation in DI and IdentityServer will then use your custom implementation.

DefaultCorsPolicyService

If you simply wish to hard-code a set of allowed origins, then there is a pre-built `ICorsPolicyService` implementation you can use called `DefaultCorsPolicyService`. This would be configured as a singleton in DI, and hard-coded with its `AllowedOrigins` collection, or setting the flag `AllowAll` to `true` to allow all origins. For example, in `ConfigureServices`:

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

Note: Use `AllowAll` with caution.

40.3 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`).

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.:

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

41.1 Extending discovery

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

```
services.AddIdentityServer(options =>
{
    options.Discovery.CustomEntries.Add("my_setting", "foo");
    options.Discovery.CustomEntries.Add("my_complex_setting",
        new
        {
            foo = "foo",
            bar = "bar"
        });
});
```

When you add a custom value that starts with ~/ it will be expanded to an absolute path below the IdentityServer base address, e.g.:

```
options.Discovery.CustomEntries.Add("my_custom_endpoint", "~/custom");
```

If you want to take full control over the rendering of the discovery (and jwks) document, you can implement the `IDiscoveryResponseGenerator` interface (or derive from our default implementation).

Adding more API Endpoints

You can add more API endpoints to the application hosting IdentityServer4.

You typically want to protect those APIs by the very instance of IdentityServer they are hosted in. That's not a problem. Simply add the token validation handler to the host (see *here*):

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

    // details omitted
    services.AddIdentityServer();

    services.AddAuthentication()
        .AddIdentityServerAuthentication("token", isAuth =>
        {
            isAuth.Authority = "base_address_of_identityserver";
            isAuth.ApiName = "name_of_api";
        });
}
```

On your API, you need to add the `[Authorize]` attribute and explicitly reference the authentication scheme you want to use (this is `token` in this example, but you can choose whatever name you like):

```csharp
public class TestController : ControllerBase
{
    [Route("test")]
    [Authorize(AuthenticationSchemes = "token")]
    public IActionResult Get()
    {
        var claims = User.Claims.Select(c => new { c.Type, c.Value }).ToArray();
        return Ok(new { message = "Hello API", claims });
    }
}
```

If you want to call that API from browsers, you additionally need to configure CORS (see *here*).

42.1 Discovery

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

```
services.AddIdentityServer(options =>
{
    options.Discovery.CustomEntries.Add("custom_endpoint", "~/api/custom");
})
```

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.

43.1 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)

43.2 Useful IdentityServer services

To achieve the above workflow, some interaction points with IdentityServer are needed.

Access to configuration and redirecting to the login page

You can get access to the IdentityServer configuration by injecting the `IdentityServerOptions` class into your code. This, e.g. has the configured path to the login page:

```
var returnUrl = Url.Action("Index");
returnUrl = returnUrl.AddQueryString(Request.QueryString.Value);

var loginUrl = _options.UserInteraction.LoginUrl;
var url = loginUrl.AddQueryString(_options.UserInteraction.LoginReturnUrlParameter,␣
↪returnUrl);

return Redirect(url);
```

Interaction between the login page and current protocol request

The `IIdentityServerInteractionService` supports turning a protocol return URL into a parsed and validated context object:

```
var context = await _interaction.GetAuthorizationContextAsync(returnUrl);
```

By default the interaction service only understands OpenID Connect protocol messages. To extend support, you can write your own `IReturnUrlParser`:

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

..and then register the parser in DI:

```
builder.Services.AddTransient<IReturnUrlParser, WsFederationReturnUrlParser>();
```

This allows the login page to get to information like the client configuration and other protocol parameters.

Access to configuration and key material for creating the protocol response

By injecting the `IKeyMaterialService` into your code, you get access to the configured signing credential and validation keys:

```
var credential = await _keys.GetSigningCredentialsAsync();
var key = credential.Key as Microsoft.IdentityModel.Tokens.X509SecurityKey;

var descriptor = new SecurityTokenDescriptor
{
    AppliesToAddress = result.Client.ClientId,
    Lifetime = new Lifetime(DateTime.UtcNow, DateTime.UtcNow.AddSeconds(result.Client.
↪IdentityTokenLifetime)),
    ReplyToAddress = result.Client.RedirectUris.First(),
    SigningCredentials = new X509SigningCredentials(key.Certificate, result.
↪RelyingParty.SignatureAlgorithm, result.RelyingParty.DigestAlgorithm),
    Subject = outgoingSubject,
    TokenIssuerName = _contextAccessor.HttpContext.GetIdentityServerIssuerUri(),
    TokenType = result.RelyingParty.TokenType
};
```

Tools

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
}
```

Discovery Endpoint

The discovery endpoint can be used to retrieve metadata about your IdentityServer - it returns information like the issuer name, key material, supported scopes etc. See the spec for more details.

The discovery endpoint is available via */.well-known/openid-configuration* relative to the base address, e.g.:

```
https://demo.identityserver.io/.well-known/openid-configuration
```

Note: You can use the IdentityModel client library to programmatically access the discovery endpoint from .NET code. For more information check the IdentityModel docs.

Authorize Endpoint

The authorize endpoint can be used to request tokens or authorization codes via the browser. This process typically involves authentication of the end-user and optionally consent.

Note: IdentityServer supports a subset of the OpenID Connect and OAuth 2.0 authorize request parameters. For a full list, see here.

client_id identifier of the client (required).

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 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.

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)

47.1 Example

```
POST /connect/token

    client_id=client1&
    client_secret=secret&
    grant_type=authorization_code&
    code=hdh922&
    redirect_uri=https://myapp.com/callback
```

(Form-encoding removed and line breaks added for readability)

Note: You can use the IdentityModel client library to programmatically access the token endpoint from .NET code. For more information check the IdentityModel docs.

UserInfo Endpoint

The UserInfo endpoint can be used to retrieve identity information about a user (see spec).

The caller needs to send a valid access token representing the user. Depending on the granted scopes, the UserInfo endpoint will return the mapped claims (at least the *openid* scope is required).

48.1 Example

```
GET /connect/userinfo
Authorization: Bearer <access_token>
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "sub": "248289761001",
    "name": "Bob Smith",
    "given_name": "Bob",
    "family_name": "Smith",
    "role": [
        "user",
        "admin"
    ]
}
```

Note: You can use the IdentityModel client library to programmatically access the userinfo endpoint from .NET code. For more information check the IdentityModel docs.

Device Authorization Endpoint

The device authorization endpoint can be used to request device and user codes. This endpoint is used to start the device flow authorization process.

Note: The URL for the end session endpoint is available via the *discovery endpoint*.

---

client_id client identifier (required)

client_secret client secret either in the post body, or as a basic authentication header. Optional.

scope one or more registered scopes. If not specified, a token for all explicitly allowed scopes will be issued.

49.1 Example

```
POST /connect/deviceauthorization

    client_id=client1&
    client_secret=secret&
    scope=openid api1
```

(Form-encoding removed and line breaks added for readability)

Note: You can use the IdentityModel client library to programmatically access the device authorization endpoint from .NET code. For more information check the IdentityModel docs.

Introspection Endpoint

The introspection endpoint is an implementation of RFC 7662.

It can be used to validate reference tokens (or JWTs if the consumer does not have support for appropriate JWT or cryptographic libraries). The introspection endpoint requires authentication - since the client of an introspection endpoint is an API, you configure the secret on the `ApiResource`.

50.1 Example

```
POST /connect/introspect
Authorization: Basic xxxyyy

token=<token>
```

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

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

Unknown or expired tokens will be marked as inactive:

```
{
    "active": false,
}
```

An invalid request will return a 400, an unauthorized request 401.

Note: You can use the IdentityModel client library to programmatically access the introspection endpoint from .NET code. For more information check the IdentityModel docs.

Revocation Endpoint

This endpoint allows revoking access tokens (reference tokens only) and refresh token. It implements the token revocation specification (RFC 7009).

token the token to revoke (required)

token_type_hint either access_token or refresh_token (optional)

51.1 Example

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

token=45ghiukldjahdnhzdauz&token_type_hint=refresh_token
```

Note: You can use the IdentityModel client library to programmatically access the revocation endpoint from .NET code. For more information check the IdentityModel docs.

End Session Endpoint

The end session endpoint can be used to trigger single sign-out (see spec).

To use the end session endpoint a client application will redirect the user's browser to the end session URL. All applications that the user has logged into via the browser during the user's session can participate in the sign-out.

Note: The URL for the end session endpoint is available via the *discovery endpoint*.

52.1 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.

52.2 Example

```
GET /connect/endsession?id_token_
→hint=eyJhbGciOiJSUzI1NiIsImtpZCI6IjdlOGFkZmMzMjU1OTEyNzI0ZDY4NWZmYmIwOThjNDEyIiwidHlwIjoiSldUIn0.
→eyJuYmYiOjE0OTE3NjUzMjEsImV4cCI6MTQ5MTc2NTYyMSwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo1MDAwIiwiYXVkIjoianN
→STzOWoeVYMtZdRAeRT95cMYEmClixWkmGwVH2Yyiks9BETotbSZiSfgE5kRh72kghN78N3-
→RgCTUmM2edB3bZx4H5ut3wWsBnZtQ2JLfhTwJAjaLE9Ykt68ovNJySbm8hjZhHzPWKh55jzshivQvTX0GdtlbcDoEA1oNONxHkp
→rAALhFPkyKnVc-uB8IHtGNSyRWLFhwVqAdS3fRNO7iIs5hYRxeFSU7a5ZuUqZ6RRi-bcDhI-
→djKO5uAwiyhfpbpYcaY_TxXWoCmq8N8uAw9zqFsQUwcXymfOAi2UF3eFZt02hBu-shKA&post_logout_
→redirect_uri=http%3A%2F%2Flocalhost%3A7017%2Findex.html
```

Note: You can use the IdentityModel client library to programmatically create end_session requests .NET code. For more information check the IdentityModel docs.

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.

54.1 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.

54.2 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:

```
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:

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

Using the convenience constructor is equivalent to this:

```
new ApiResource
{
    Name = "api1",
    DisplayName = "Some API 1",

    Scopes =
    {
        new Scope()
        {
            Name = "api1",
            DisplayName = "Some API 1"
        }
    }
}
```

Client

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.

55.1 Basics

Enabled Specifies if client is enabled. Defaults to *true*.

ClientId Unique ID of the client

ClientSecrets List of client secrets - credentials to access the token endpoint.

RequireClientSecret Specifies whether this client needs a secret to request tokens from the token endpoint (defaults to `true`)

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.

55.2 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`.

55.3 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 requring 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.

55.4 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)

55.5 Device flow

UserCodeType Specifies the type of user code to use for the client. Otherwise falls back to default.

DeviceCodeLifetime Lifetime to device code in seconds (defaults to 300 seconds / 5 minutes)

GrantValidationResult

The `GrantValidationResult` class models the outcome of grant validation for extensions grants and resource owner password grants.

The most common usage is to either new it up using an identity (success case):

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

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

```
context.Result = new GrantValidationResult(
    TokenRequestErrors.InvalidGrant,
    "invalid custom credential");
```

In both case you can pass additional custom values that will be included in the token response.

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.

57.1 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`.

57.2 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 `Claim` s 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`.

57.3 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`.

57.4 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.

IdentityServer Interaction Service

The `IIdentityServerInteractionService` interface is intended to provide services to be used by the user interface to communicate with IdentityServer, mainly pertaining to user interaction. It is available from the dependency injection system and would normally be injected as a constructor parameter into your MVC controllers for the user interface of IdentityServer.

58.1 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.

58.2 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.

58.3 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.

58.4 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.

58.5 ConsentResponse

ScopesConsented The collection of scopes the user consented to.

RememberConsent Flag indicating if the user's consent is to be persisted.

58.6 Consent

SubjectId The subject id that granted the consent.

ClientId The client identifier for the consent.

Scopes The collection of scopes consented to.

CreationTime The date and time when the consent was granted.

Expiration The date and time when the consent will expire.

Device Flow Interaction Service

The `IDeviceFlowInteractionService` interface is intended to provide services to be used by the user interface to communicate with IdentityServer during device flow authorization. It is available from the dependency injection system and would normally be injected as a constructor parameter into your MVC controllers for the user interface of IdentityServer.

59.1 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`.

59.2 DeviceFlowAuthorizationRequest

ClientId The client identifier that initiated the request.

ScopesRequested The scopes requested from the authorization request.

59.3 DeviceFlowInteractionResult

IsError Specifies if the authorization request errored.

ErrorDescription 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.

60.1 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.

60.2 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.

60.3 Authentication

- **CookieLifetime** The authentication cookie lifetime (only effective if the IdentityServer-provided cookie handler is used).

- **CookieSlidingExpiration** Specified if the cookie should be sliding or not (only effective if the IdentityServer-provided cookie handler is used).

- **RequireAuthenticatedUserForSignOutMessage** Indicates if user must be authenticated to accept parameters to end session endpoint. Defaults to false.

- **CheckSessionCookieName** The name of the cookie used for the check session endpoint.

- **RequireCspFrameSrcForSignout** If set, will require frame-src CSP headers being emitting on the end session callback endpoint which renders iframes to clients for front-channel signout notification. Defaults to true.

60.4 Events

Allows configuring if and which events should be submitted to a registered event sink. See *here* for more information on events.

60.5 InputLengthRestrictions

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

60.6 UserInteraction

- **LoginUrl, LogoutUrl, ConsentUrl, ErrorUrl, DeviceVerificationUrl** Sets the 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.

60.7 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.

60.8 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.

60.9 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.

60.10 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*.

Entity Framework Support

An EntityFramework-based implementation is provided for the configuration and operational data extensibility points in IdentityServer. The use of EntityFramework allows any EF-supported database to be used with this library.

The 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.

61.1 Configuration Store support for Clients, Resources, and CORS settings

If client, identity resource, API resource, or CORS data is desired to be loaded from a EF-supported database (rather than use in-memory configuration), then the configuration store can be used. This support provides implementations of the `IClientStore`, `IResourceStore`, and the `ICorsPolicyService` extensibility points. These implementations use a `DbContext`-derived class called `ConfigurationDbContext` to model the tables in the database.

To use the configuration store support, use the `AddConfigurationStore` extension method after the call to `AddIdentityServer`:

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    const string connectionString = @"Data Source=(LocalDb)\MSSQLLocalDB;
↪database=IdentityServer4.EntityFramework-2.0.0;trusted_connection=yes;";
    var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

    services.AddIdentityServer()
        // this adds the config data from DB (clients, resources, CORS)
        .AddConfigurationStore(options =>
        {
            options.ConfigureDbContext = builder =>
```

(continues on next page)

```
            builder.UseSqlServer(connectionString,
                sql => sql.MigrationsAssembly(migrationsAssembly));
        });
}
```

To configure the configuration store, use the `ConfigurationStoreOptions` options object passed to the configuration callback.

61.2 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`.

61.3 Operational Store support for authorization grants, consents, and tokens (refresh and reference)

If authorization grants, consents, and tokens (refresh and reference) are desired to be loaded from a EF-supported database (rather than the default in-memory database), then the operational store can be used. This support provides implementations of the `IPersistedGrantStore` extensibility point. The implementation uses a `DbContext`-derived class called `PersistedGrantDbContext` to model the table in the database.

To use the operational store support, use the `AddOperationalStore` extension method after the call to `AddIdentityServer`:

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    const string connectionString = @"Data Source=(LocalDb)\MSSQLLocalDB;
→database=IdentityServer4.EntityFramework-2.0.0;trusted_connection=yes;";
    var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

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

61.4 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).

61.5 Database creation and schema changes across different versions of IdentityServer

It is very likely that across different versions of IdentityServer (and the EF support) that the database schema will change to accommodate new and changing features.

We do not provide any support for creating your database or migrating your data from one version to another. You are expected to manage the database creation, schema changes, and data migration in any way your organization sees fit.

Using EF migrations is one possible approach to this. If you do wish to use migrations, then see the *EF quickstart* for samples on how to get started, or consult the Microsoft documentation on EF migrations.

We also publish sample SQL scripts for the current version of the database schema.

ASP.NET Identity Support

An ASP.NET Identity-based implementation is provided for managing the identity database for users of IdentityServer. This implementation implements the extensibility points in IdentityServer needed to load identity data for your users to emit claims into tokens.

The repo for this support is located here and the NuGet package is here.

To use this library, configure ASP.NET Identity normally. Then use the `AddAspNetIdentity` extension method after the call to `AddIdentityServer`:

```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).

Training

Here are some online, remote and classroom training options to learn more about ASP.NET Core identity & Identity-Server4.

63.1 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.

63.2 PluralSight courses

There are some good courses on PluralSight around identity, ASP.NET Core and IdentityServer.

new

- Securing Angular Apps with OpenID and OAuth2
- ASP.NET Core Identity Management Playbook
- Getting Started with ASP.NET Core and OAuth
- Securing ASP.NET Core with OAuth2 and OpenID Connect
- Understanding ASP.NET Core Security (Centralized Authentication with a Token Service)

older

- Introduction to OAuth2, OpenID Connect and JSON Web Tokens (JWT)
- Web API v2 Security
- Using OAuth to Secure Your ASP.NET API

- OAuth2 and OpenID Connect Strategies for Angular and ASP.NET

Blog posts

64.1 Team posts

- Platforms where you can run IdentityServer4
- Optimizing Tokens for size
- Identity vs Permissions
- Bootstraping OpenID Connect: Discovery
- Extending IdentityServer4 with WS-Federation Support
- Announcing IdentityServer4 RC1
- Getting Started with IdentityServer 4
- IdentityServer 4 SharePoint Integration using WS-Federation
- IdentityServer and Swagger

64.2 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

64.3 Community posts

- 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

Videos

65.1 2018

- 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

65.2 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

65.3 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

65.4 2015

- Securing Web APIs – Patterns & Anti-Patterns - Dominick Baier

- Authentication and authorization in modern JavaScript web applications – how hard can it be? - Brock Allen

65.5 2014

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