Api security with IdentityServer4 & AspNetCore.Identity

[**IdentityServer4**](#_o4s9i7n12vqe) **1**

[Endpoints](#_ab42wm159429) 1

[Configuration](#_8kyt0w7zyuxp) 2

[The Quickstart template](#_v2lnwr8q450a) 2

[Using a persistent database](#_ta7j4g5gi7ss) 3

[Persistent grant store](#_mlhj9wb5g5wv) 3

[Client and Scope stores](#_jp1cm4tcmc1k) 4

[EF Migrations](#_wvv8sw9egzzu) 4

[AspNetCore.Identity](#_7r54f6vspif) **5**

[Working with users](#_iqmrhtberi59) 7

[Working with roles](#_cujn6q2mhu79) 7

[Modifying the Quickstart template](#_3uceut601u9k) 8

[**Configuring clients, resources & scopes**](#_lojym79k5c68) **11**

[Identity Resources vs Api Resources & Api Scopes](#_db6zbio3mfyv) 12

[Identity Resources](#_mlrdog7i2us3) 12

[Adding role permissions to user claims](#_oqsag8v3239a) 13

[Api resources & scopes](#_mgxp97cabq2) 13

[Scopes](#_f3vtd5x1ygk7) 14

[Resources](#_um3af49w0x0z) 16

[Clients](#_5xmfy1wslrl5) 17

[Server to server clients](#_bcjgzdyks1we) 17

[Interactive application clients](#_qwcaxdmsvj27) 17

[**Api Security**](#_ioatfv2mpe86) **18**

[Authentication](#_el6y8nhbvomi) 18

[Authorization](#_acc7ild3f0nm) 19

[Policy based authorization](#_rldmaj8bvzip) 20

[**Demo database seeding**](#_6ktdcfiwkacx) **21**

# IdentityServer4

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

When implemented it enables the following within a solution:

* **Authentication as a Service** (Officially certified implementation of OpenID)
* **Single Sign-on / Sign-out**
* **Access Control for APIs**
* **Federation Gateway**
* **Focus on Customization** (Adaptable to system needs)
* **Mature Open Source**
* **Free and Commercial Support**

Extensive documentation can be found here:

<http://docs.identityserver.io/en/latest/index.html>

## Endpoints

IdentityServer4 serves the following endpoints out of the box:

* **Discovery** (Accessed at */.well-known/openid-configuration* [GET request])
  + Returns all OpenID provider metadata configured on the project (See more about metadata: <https://openid.net/specs/openid-connect-discovery-1_0.html#ProviderMetadata>)
* **Authorize** (Accessed at */connect/authorize* [GET request])
  + Used for requesting tokens or authorization codes. Involves resource owner authentication.
* **Token** (Accessed at */connect/token* [POST request])
  + For programmatically requesting tokens. Supports *password, authorization\_code, client\_credentials, refresh\_token* and *urn:ietf:params:oauth:grant-type:device\_code* grant types out of the box
* **UserInfo** (Accessed at */connect/userinfo* [GET request])
  + Given a valid access token returns information on resource owner(contextually a user) claims depending on scope of request. Requires minimum scope of openid and will not serve any other valid scopes without it.
* **Device Authorization** (Accessed at */connect/deviceauthorization* [POST request])
  + Used to start device flow authentication. Returns device or user code
* **Introspection** (Accessed at */connect/introspect* [POST request])
  + Used to validate reference tokens and JWT
* **Revoke** (Accessed at */connect/revocation* [POST request])
  + Used to revoke access and refresh tokens
* **End Session** (Accessed at */connect/endsession* [GET request])
  + Triggers single sign-out. Will access for confirmation prompt but can be bypassed by passing the original id token from sign in.

More information all the endpoints can be found here:

<https://github.com/IdentityServer/IdentityServer4/tree/main/docs/endpoints>

## Configuration

Before you can start using IdentityServer4 in ASP.NET Core you must first include its package into the project:

dotnet add package IdentityServer4

After including the package the bare minimum steps of implementation require you to configure the service by going to the ConfigureServices method in Startup.cs as following:

services.AddIdentityServer()

.AddInMemoryClients(new List<Client>())

.AddInMemoryIdentityResources(new List<IdentityResource>())

.AddInMemoryApiResources(new List<ApiResource>())

.AddInMemoryApiScopes(new List<ApiScope>())

.AddTestUsers(new List<TestUser>())  
 .AddDeveloperSigningCredential();

This configures IdentityServer service and tells it to use in memory entities that will not persist on project reloads and includes a signing credential for tokens that should only be used in a Development environment. Once the service is configured to enable IdentityServer to start handling OAuth and OpenID connect requests in the Configure method in Startup.cs call the following:

app.UseIdentityServer();

With this the project is ready to serve the mentioned endpoints. It is not necessary to call the UseAuthentication() method because the UseIdentityServer() already includes all of its functions and capabilities.

### The Quickstart template

So far the IdentityServer solution does not have a UI which makes actions like login a bit more inconvenient. To remedy the problem IdentityServer has a quickstart UI stored on a readily accessible repository.

To add the UI to the project the following powershell command can be executed:

iex ((New-Object System.Net.WebClient).DownloadString('https://raw.githubusercontent.com/IdentityServer/IdentityServer4.Quickstart.UI/main/getmain.ps1'))

To actually enable it some modifications to the Startup.cs are required. In the Configure method add the following calls:

app.UseStaticFiles();

app.UseRouting();

app.UseIdentityServer();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapDefaultControllerRoute();

});

### Using a persistent database

Implementing an in memory version of IdentityServer isn’t very practical outside of simple testing and demo. Luckily IdentityServer4 includes integration for persistent databases and has an entity framework package.

In my solution I use MS SQL as my persistent database so the following packages are added:

dotnet add package IdentityServer4.EntityFramework

dotnet add package Microsoft.EntityFramework.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Design

Furthermore for working with entity framework a EF Core tool is required to be installed which can be found at:

<https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/>

To support the necessary entity framework calls that will be necessary for setting up IdentityServer to work with a persistent database the following is included in Startup.cs.

string connectionString;

string migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

public Startup(IConfiguration configuration)

{

Configuration = configuration;

connectionString = configuration.GetConnectionString("sqlDB");

}

#### Persistent grant store

The grant store maintains all the temporary data IdentityServer uses for operation such as consent, all the various tokens and authorization/device codes. Without a persistent grant store all this temporary information would be lost on restart invalidating tokens before they should be invalidated.

To include support for the persistent grant store the AddIdentityServer call in ConfigureServices method is updated as following:

services.AddIdentityServer()

.AddInMemoryClients(new List<Client>())

.AddInMemoryIdentityResources(new List<IdentityResource>())

.AddInMemoryApiResources(new List<ApiResource>())

.AddInMemoryApiScopes(new List<ApiScope>())

.AddTestUsers(new List<TestUser>())

.AddOperationalStore(options =>

{

options.ConfigureDbContext = builder =>

{

builder.UseSqlServer(connectionString, sqlOptions =>

{

sqlOptions.MigrationsAssembly(migrationsAssembly);

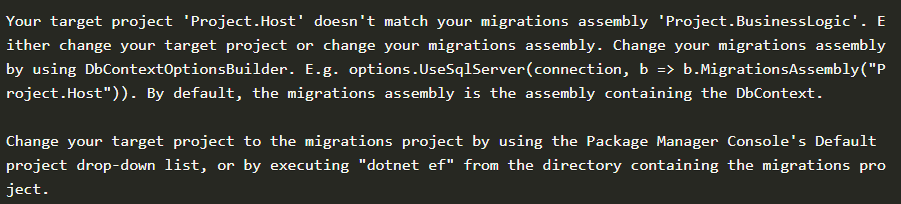
});

};

})

.AddDeveloperSigningCredential();

Migration assembly is necessary to be defined as the project hosting the IdentityServer otherwise the DbContext will not be able to be targeted since it is within a nuget package. With the migration assembly correctly configured we can run EF migrations effortlessly. Otherwise we would be met with an exception similar to as following:



#### Client and Scope stores

To store all the resources, scopes and clients used for authentication with IdentityServer a similar call as the grant store is used and this replaces all the InMemory calls from before leading to the AddIdentityServer call to look as following:

services.AddIdentityServer()

.AddConfigurationStore(options =>

{

options.ConfigureDbContext = builder =>

{

builder.UseSqlServer(connectionString, sqlOptions =>

{

sqlOptions.MigrationsAssembly(migrationsAssembly);

});

};

})

.AddTestUsers(new List<TestUser>())

.AddOperationalStore(options =>

{

options.ConfigureDbContext = builder =>

{

builder.UseSqlServer(connectionString, sqlOptions =>

{

sqlOptions.MigrationsAssembly(migrationsAssembly);

});

};

})

#### EF Migrations

Finally to actually include all the stores configured in configure service for IdentityServer EF migrations are required to be run to be able to update the databases. This looks as following:

dotnet ef migrations add InitialIdentityServerMigration -c PersistedGrantDbContext

dotnet ef migrations add InitialIdentityServerMigration -c ConfigurationDbContext

Then to update and include all the tables into the database the following commands are executed:

dotnet ef database update -c PersistedGrantDbContext

dotnet ef database update -c ConfigurationDbContext

# AspNetCore.Identity

For persistent storage of users IdentityServer4 does not have any personal implementation but it does however support AspNetCore’s Identity as a user store. This is extremely convenient as AspNetCore’s Identity includes all the basic features required to set up a production-ready authentication system. There is also the added benefit of Identity not being just a user store but a role store as well which will come into play later for role-based authorization.

To start implementing AspNetCore.Identity first the following nuget packages must be installed:

dotnet add package IdentityServer4.AspNetIdentity

dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore

Second step is to create a DbContext that inherits IdentityDbContext which overrides the constructor to use a non generic DbContextOptions. Otherwise IdentityDbContext will return an InvalidOperationException when using multiple DbContext like this project is.

Reference:<https://github.com/aspnet/Identity/issues/962>

The class should look something like this at a bare minimum:

public class ApplicationDbContext : IdentityDbContext

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)

: base(options)

{

}

}

Now that the custom DbContext is created, it and the IdentityDbContext have to be registered in the ConfigureServices method in Startup.cs and this has to be done **BEFORE** the AddIdentityServer call.

services.AddDbContext<ApplicationDbContext>(builder =>

{

builder.UseSqlServer(connectionString, sqlOptions =>

{

sqlOptions.MigrationsAssembly(migrationsAssembly);

});

});

services.AddIdentity<IdentityUser, IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>();

services.AddIdentityServer()

...

...

Note that using services.AddIdentity() changes the application's default cookie scheme to IdentityConstants.ApplicationScheme.

Now finally the last artifact of the AddIdentityServer() for demo purposes .AddTestUsers() can be replaced with the following call:

services.AddIdentityServer()

.AddConfigurationStore(options =>

{

options.ConfigureDbContext = builder =>

{

builder.UseSqlServer(connectionString, sqlOptions =>

{

sqlOptions.MigrationsAssembly(migrationsAssembly);

});

};

})

.AddAspNetIdentity<IdentityUser>()

.AddOperationalStore(options =>

{

options.ConfigureDbContext = builder =>

{

builder.UseSqlServer(connectionString, sqlOptions =>

{

sqlOptions.MigrationsAssembly(migrationsAssembly);

});

};

})

Note that the call takes a type argument that is the same user class we used in our definition of the IdentityDbContext. This call is incredibly important for using AspNetCore Identity as a user store because it implements definitions for IdentityServer’s IProfileService, IUserClaimsPrincipalFactory, IResourceOwnerPasswordValidator which are essential for using Identity’s users as the resource owners for IdentityServer and will come into play later.

At a high level this call does:

* Adds an ASP.NET Identity compatible profile service (how IdentityServer generated user claims)
* Adds an extended implementation of ASP.NET Identity’s IUserClaimsPrincipalFactory (how ASP.NET Identity transforms a user object into claims)
* Configures IdentityServer to use ASP.NET Identity’s cookie scheme and tweaks those cookies to be suitable for OpenID Connect

See more on the method call at:

<https://identityserver4.readthedocs.io/en/latest/reference/aspnet_identity.html>

Last step in adding the Identity user and role stores is running the DbContext migrations as following:

dotnet ef migrations add InitialIdentityServerMigration -c ApplicationDbContext

dotnet ef database update -c ApplicationDbContext

## Working with users

To work with users the AddAspNetIdentity call includes a dependency injection towards a UserManger<User> class which as the name says manages all the CRUD for users in the user store. It serves as an interface towards the user store regardless of implementation, in memory or database. An example of working with the usermanager looks as following:

if (!userManager.Users.Any())

{

foreach (var testUser in TestUsers.Users)

{

var identityUser = new IdentityUser(testUser.Username)

{

Id = testUser.SubjectId

};

userManager.CreateAsync(identityUser, testUser.Password).Wait();

userManager.AddClaimsAsync(identityUser, testUser.Claims.ToList()).Wait();

foreach (var claim in testUser.Claims.ToList())

{

if (claim.Type == "role")

{

if (claim.Value == "admin")

{

userManager.AddToRoleAsync(identityUser,"admin").Wait();

}

}

}

}

}

In this example it checks if the usermanager contains any users. If not based on the users in the TestUser class creates new user objects and populates the usermanager with the users information and claims. Additionally if it contains a role claim it will check if the value is equal to admin and add it to the Identityrole named admin creating an association in the table between the role and user.

For more information on the user manager:

<https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.usermanager-1?view=aspnetcore-3.1>

## Working with roles

Analogically to UserManager the RoleManager does the same except instead of handling the user store it works with the roles store. An example using the role manager looks as following:

var roleManager = serviceScope.ServiceProvider.GetRequiredService<RoleManager<IdentityRole>>();

if (!roleManager.Roles.Any())

{

var adminRole = new IdentityRole("admin");

roleManager.CreateAsync(adminRole).Wait();

roleManager.AddClaimAsync(adminRole, new Claim("permission", "projects.view")).Wait();

roleManager.AddClaimAsync(adminRole, new Claim("permission", "projects.create")).Wait();

roleManager.AddClaimAsync(adminRole, new Claim("permission", "projects.update")).Wait();

var superUserRole = new IdentityRole("superuser");

roleManager.CreateAsync(superUserRole).Wait();

roleManager.AddClaimAsync(superUserRole, new Claim("permission", "projects.view")).Wait();

roleManager.AddClaimAsync(superUserRole, new Claim("permission", "projects.create")).Wait();

var userRole = new IdentityRole("user");

roleManager.CreateAsync(userRole).Wait();

roleManager.AddClaimAsync(userRole, new Claim("permission", "projects.view")).Wait();

}

In this example the rolemanager creates 3 new roles if it is empty named admin, superuser and user for which each are defined specific claims of the type “permission”.

For more information on the rolemanager:

<https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.rolemanager-1?view=aspnetcore-3.1>

## Modifying the Quickstart template

With the addition of using AspNetCore.Identity the quickstart UI needs some modifications to properly implement the user and role stores instead of the testuser classes it uses by default.

First is the HomeController.cs that needs to be modified. In the constructor we replace all references to TestUserStore with SignInManager.

private readonly IIdentityServerInteractionService \_interaction;

private readonly IClientStore \_clientStore;

private readonly IAuthenticationSchemeProvider \_schemeProvider;

private readonly IEventService \_events;

private readonly SignInManager<IdentityUser> \_signInManager;

public AccountController(

IIdentityServerInteractionService interaction,

IClientStore clientStore,

IAuthenticationSchemeProvider schemeProvider,

IEventService events,

SignInManager<IdentityUser> signInManager)

{

\_interaction = interaction;

\_clientStore = clientStore;

\_schemeProvider = schemeProvider;

\_events = events;

\_signInManager = signInManager;

}

This in turn breaks the Login method. This is a simple fix of simply using the \_signInManager to get the user instead of the testUserStore. Which looks something as following:

**BEFORE**

if (\_users.ValidateCredentials(model.Username, model.Password))

{

var user = \_users.FindByUsername(model.Username);

await \_events.RaiseAsync(new UserLoginSuccessEvent(user.Username, user.SubjectId, user.Username, clientId: context?.Client.ClientId));

AuthenticationProperties props = null;

if (AccountOptions.AllowRememberLogin && model.RememberLogin)

{

props = new AuthenticationProperties

{

IsPersistent = true,

ExpiresUtc = DateTimeOffset.UtcNow.Add(AccountOptions.RememberMeLoginDuration)

};

};

var isuser = new IdentityServerUser(user.SubjectId)

{

DisplayName = user.Username

};

await HttpContext.SignInAsync(isuser, props);

}

**AFTER**

var user = await \_signInManager.UserManager.FindByNameAsync(model.Username);

if (user != null && (await \_signInManager.CheckPasswordSignInAsync(user, model.Password, true)) == Microsoft.AspNetCore.Identity.SignInResult.Success)

{

await \_events.RaiseAsync(new UserLoginSuccessEvent(user.UserName, user.Id, user.UserName, clientId: context?.Client.ClientId));

AuthenticationProperties props = null;

if (AccountOptions.AllowRememberLogin && model.RememberLogin)

{

props = new AuthenticationProperties

{

IsPersistent = true,

ExpiresUtc = DateTimeOffset.UtcNow.Add(AccountOptions.RememberMeLoginDuration)

};

};

var isuser = new IdentityServerUser(user.Id)

{

DisplayName = user.UserName

};

Similar goes for the ExternalController which serves external identity providers.

First we update the constructor but this time we use the UserManager instead of the SignInManager:

private readonly UserManager<IdentityUser> \_userManager;

private readonly IIdentityServerInteractionService \_interaction;

private readonly IClientStore \_clientStore;

private readonly ILogger<ExternalController> \_logger;

private readonly IEventService \_events;

public ExternalController(

IIdentityServerInteractionService interaction,

IClientStore clientStore,

IEventService events,

ILogger<ExternalController> logger,

UserManager<IdentityUser> userManager)

{

\_userManager = userManager;

\_interaction = interaction;

\_clientStore = clientStore;

\_logger = logger;

\_events = events;

}

And then we fix the following methods: FindUserFromExternalProvider & AutoProvisionUser

**BEFORE**

private (TestUser user, string provider, string providerUserId, IEnumerable<Claim> claims) FindUserFromExternalProvider(AuthenticateResult result)

{

var externalUser = result.Principal;

var userIdClaim = externalUser.FindFirst(JwtClaimTypes.Subject) ??

externalUser.FindFirst(ClaimTypes.NameIdentifier) ??

throw new Exception("Unknown userid");

var claims = externalUser.Claims.ToList();

claims.Remove(userIdClaim);

var provider = result.Properties.Items["scheme"];

var providerUserId = userIdClaim.Value;

var user = \_users.FindByExternalProvider(provider, providerUserId);

return (user, provider, providerUserId, claims);

}

private TestUser AutoProvisionUser(string provider, string providerUserId, IEnumerable<Claim> claims)

{

var user = \_users.AutoProvisionUser(provider, providerUserId, claims.ToList());

return user;

}

**AFTER**

private async Task<(IdentityUser user, string provider, string providerUserId, IEnumerable<Claim> claims)> FindUserFromExternalProvider(AuthenticateResult result)

{

var externalUser = result.Principal;

var userIdClaim = externalUser.FindFirst(JwtClaimTypes.Subject) ??

externalUser.FindFirst(ClaimTypes.NameIdentifier) ??

throw new Exception("Unknown userid");

var claims = externalUser.Claims.ToList();

claims.Remove(userIdClaim);

var provider = result.Properties.Items["scheme"];

var providerUserId = userIdClaim.Value;

var user = await \_userManager.FindByLoginAsync(provider, providerUserId);

return (user, provider, providerUserId, claims);

}

private async Task<IdentityUser> AutoProvisionUser(string provider, string providerUserId, IEnumerable<Claim> claims)

{

var user = new IdentityUser(Guid.NewGuid().ToString());

await \_userManager.CreateAsync(user);

await \_userManager.AddLoginAsync(user, new UserLoginInfo(provider, providerUserId, provider));

return user;

}

# Configuring clients, resources & scopes

OAuth and OpenID Connect will not work if the IdentityServer does not have defined clients which are allowed to attempt to communicate with the IdentityServer. These clients are not only restricted by the ID of the client but in what they can access by defining it’s allowed scopes which are used to access specific parts of resources. Configuration is done programmatically for purposes of setting an initial seed for the IdentityServer.

## Identity Resources vs Api Resources & Api Scopes

IdentityServer defines 2 types of resources which are crucial for understanding how it works and how tokens are created. The first kind is an IdentityResource which is a group of claims about a user, the resource owner. While the second is an ApiResource which represents functionalities a client wants to access (typically endpoints). It is used as a namespacing to group and organise a set of scopes. You will also want to define an ApiResource if you want for example an audience claim in your access token.

### Identity Resources

Typically a list of specific claims used for requesting information about the resource owner (the user). When an identity resource is requested by a client IdentityServer will return the information of these claims in an id token. This is the only way to get an id token from the Identity server, if the client does not have a scope covered by the identity resource the response will only contain an access token given it is a valid request.

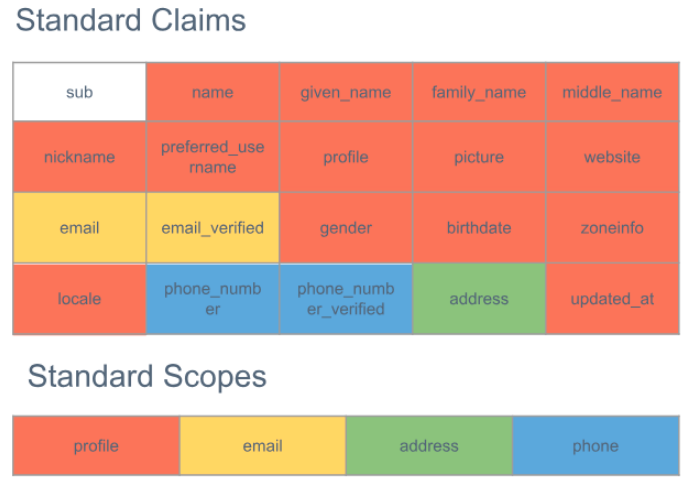
Each identity resource is treated as a scope and has a subset of claims associated with it according to the possible claims a user can have.

The minimum requirement for identity resources is to have a defined scope for openid given that this is mandatory for IdentityServer to give an id token response.

Other standard OpenId Connect scopes should be included as well. These are Profile, Email and Telephone as per the OpenId Connect specification found at:

<https://openid.net/specs/openid-connect-core-1_0.html#ScopeClaims>

A visualisation of the standard OpenId Connect scope claims can be found on the following image:



The scope openid is not defined under standard scope as it is mandatory and covers the “white” sub claim.

To create custom claims that an id token will return you must define a new Identity Resource with the bare minimum of giving it a name and at least a single claim.

My solution has the following identity resources defined:

return new IdentityResource[]

{

new IdentityResources.OpenId(),

new IdentityResources.Profile(),

new IdentityResources.Email(),

new IdentityResource

{

Name = "role",

UserClaims = new List<string> {"role"}

},

new IdentityResource

{

Name = "permission",

UserClaims = new List<string> {"permission"}

}

};

This implementation covers the OpenId Connect standard scopes of openid, profile, email and includes 2 custom scopes called “role” which returns user claims of the type “role” and a second scope called “permission” that returns all user claims of type “permission”. Usually for adding custom claims one would need to implement it’s own IUserClaimsPrincipalFactory and IProfileService but thanks to the call “AddAspNetIdentity” on the AddIdentityServer service the default AspNetIdentity implementation covers creating the custom user claims and as per request and the profile service that is incharge for loading claims for a user when the id token is request will add the custom user claims to the token itself.

For more about IdentityResources:

<https://identityserver4.readthedocs.io/en/latest/reference/identity_resource.html>

#### Adding role permissions to user claims

Since AspNetIdentity defines a role store alongside the user store it would make sense to also access the permissions of a role if the user is associated to one. In the context of the role store, every role has claims that it can declare much like a user declares its own claims. These claims could also include and be attributed as permissions for the role. Luckily if using the AspNetIdentity implementations of IUserClaimsPrincipalFactory the Identity object already includes all of the IdentityRole claims alongside the IdentityUser claims as one list of claims and exposes them all as user claims to be accessed by an IdentityResource requesting the specific claim type.

### Api resources & scopes

OAuth 2.0 specification contains the concept of scopes in which it defines them as “the scope of access” - these are essentially just a list of values. It is up to the programmer to implement the logic and structure of what to do with these values.

For more complex systems with a robust number of api’s or scopes, simply using a list of scopes becomes hard to maintain and it is desirable to create some kind of groupation. This often manifests as a resource. It can be a representation of a physical API or simply a logical API for which api specific scopes are defined. It is also possible to attribute the same scope to multiple resources as a shared scope.

#### Scopes

Scopes are the main method of defining access, formally speaking IdentityResources are also a type of specialised scopes that are designed to manage resource owner information. They are the minimal requirement of defining what a client is trying to do. When a client asks for a scope and it is allowed in configuration its value will be included in the access token as a claim of type scope. Generally this is added as a way to verify on the token consumers end if the client is actually allowed to invoke the function it is trying to do. This is called scope based authorization.

Some simple custom api scopes would be something of the sorts as following:

return new[]

{

new ApiScope("api1.read", "Read Access to Api #1"),

new ApiScope("api1.write", "Write Access to Api #1")

};

We define the name of the scope in the first parameter, this is used for when a user tries to invoke a scope for the identityserver to actually be able to find the scope and can validate it. Second is just a simple description. ApiScopes can also have a user claim property which if called will add an identity resource claim of the same name to the request. If the claim has a value it will be exposed to the access token as following:

new ApiScope("test","Test scope",new List<string>{"role"});

In this example we create a scope that exposes just the user claim of type “role”, this is defined by the third parameter in the declaration. Now when we request a token with the scope “test”, if this is a valid scope defined by the client and the resource owner that is requesting has a defined claim of role with a value we get an access token as following:



If we are to edit the scope and define it without this claim we can see in the following token the claim “role” is missing from the access token.



Essentially we are passing the requested claim type to the profile service and are telling the profile service when you receive a request for this scope when creating the tokens, add the user claim of this type if they have a value. Usually this would require an implementation of profile service but since in my solution the AspNetIdentity extension is included the default implementation handles this just fine and will add any kind of defined claim for the user.

For more on ApiScopes:

<https://identityserver4.readthedocs.io/en/latest/reference/api_scope.html>

#### Resources

Resources are designed to be a model of an api that identityserver is protecting. They are primarily used to group and organise a list of ApiScopes that are designated for accessing the api. Multiple resources can share the same scope but they can also have their own scopes specifically for that resource.

Using resources also gives the following features:

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

Example of defined api resources:

return new[]

{

new ApiResource

{

Name = "api1",

DisplayName = "Api #1",

Description = "Allow the application to access API #1 on your behalf",

Scopes = new List<string>{"api1.read","test"},

ApiSecrets = new List<Secret>{new Secret("Secret".Sha256())}

},

new ApiResource

{

Name = "weatherApi",

DisplayName = "Weather Api",

Description = "Allow the application to access Weather Api on your behalf",

Scopes = new List<string>{"api1.read"},

ApiSecrets = new List<Secret>{new Secret("Secret".Sha256())},

UserClaims = new List<string>{"role", "permission"}

}

};

}

The Scopes property assigns what scopes should be grouped under which resource, when this scope is called the audience claim will assign the name of the api resource as its value. Api secret is for the Api itself to authenticate the request. As with ApiScopes, ApiResources also have a UserClaims property in which functions identically to the ApiScope except it will append it to all access tokens with the audience of that specific resource rather than just being dependent on a specific scope. This helps with not needing to repeat the same claims across multiple scopes.

For information about ApiResource properties:

<https://identityserver4.readthedocs.io/en/latest/reference/api_resource.html>

## Clients

These represent applications that can request tokens from the identityserver service configured in the project. Typical settings for a client are as following:

* 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

Typically there are two types of clients for defining. First type is a client for server to server communication, these are for scenarios where there is no interactive user present I.E. a service accessing an API. The other type of clients are for interactive applications that are used for authentication (logins) and delegating API access. These types of clients use the authorization code flow and have an interactive user involved in the situation. In this flow the access tokens are transmitted via back-channels only.

IdentityServer4 is designed to protect user claims and by default it will not send user claims even inside of id tokens as a response to the clients. The client needs to set the AlwaysIncludeUserClaimsInIdToken property to true, otherwise identityserver4 will require the client to access the userinfo endpoint with the access token to see information on user claims. If the property is not changed and the client calls a resource or scope that has user claims defined to be exposed to the access token the access token will not contain the claim as the id token will not have the claim nor any value for the profile service to include.

For information on all properties defined for clients:

<https://identityserver4.readthedocs.io/en/latest/reference/client.html>

### Server to server clients

Server to server clients are a simple OAuth client and therefore do not require a lot of properties to be set as they’re designed for an application or service accessing an API, sometimes without a user involved whatsoever and only defining scopes of its request.

A sample of a server to server client defined in my project is as following:

new Client

{

ClientId = "oauthClient",

ClientName = "Example client application using client credentials",

AllowedGrantTypes = GrantTypes.ClientCredentials,

ClientSecrets = new List<Secret> {new Secret("Secret".Sha256())},

AllowedScopes = new List<string> {"api1.write", "api1.read"},

AlwaysIncludeUserClaimsInIdToken = true

}

### Interactive application clients

Interactive applications use the OpenID Connect technology for authorization and therefore usually require interactive login screens. For a client using an interactive flow it is imperative to define the redirect uri that is permitted for the client to secure the information and make sure they do not get redirected elsewhere.

An example of an interactive client in my project looks like this:

new Client

{

ClientId = "oidcClient",

ClientName = "Example Client Application",

ClientSecrets = new List<Secret> {new Secret("Secret".Sha256())}

AllowedGrantTypes = GrantTypes.Code,

RedirectUris = new List<string> {"https://localhost:44377/signin-oidc"},

AllowedScopes = new List<string>

{

IdentityServerConstants.StandardScopes.OpenId,

IdentityServerConstants.StandardScopes.Profile,

IdentityServerConstants.StandardScopes.Email,

"role",

"api1.read",

"permission"

},

RequirePkce = true,

AllowPlainTextPkce = false

}

Note that the Grant type is different to support the required flow and the addition of a redirecturi and proof key certificate exchange properties. The Pkce properties aren’t necessary but since authorization code supports it is a good idea to define properties regarding it.

# Api Security

It is not enough to just define and configure the IdentityServer service and include it into the solution. While some forms of authentication and authorization exist to fully implement API Security there are steps that are required to be included on the API’s we are protecting.

## Authentication

First step in securing the API is adding a new service in the Startup.cs in the ConfigureServices method and to call app.UseAuthentication() in the Configure method. An example of a service from my projects API looks as following:

services.AddAuthentication("Bearer")

.AddJwtBearer("Bearer", options =>

{

options.Authority = Configuration.GetConnectionString("identity");

options.RequireHttpsMetadata = false;

options.Audience = "doctorsPracticeApi";

});

In this for authentication of requests since IdentityServer gives us access tokens and identity tokens in the form of jwt we define authentication of type “bearer” and tell it to use JWT bearer tokens only. The bare minimum for options is to add an authority to tell the API only accept tokens from a certified token provider. It also includes an audience meaning it will only respond to a token with the audience claim containing the correct value.

If the flow is interactive using openid connect and you want the api to support openid authentication then the AddOpenIdConnect call is used. To be able to call the method one first must add the following nuget package to the project:

dotnet add package Microsoft.AspNetCore.Authentication.OpenIdConnect

An example of an API secured by openid is as following:

services.AddAuthentication(options =>

{

options.DefaultScheme = "cookie";

options.DefaultChallengeScheme = "oidc";

})

.AddCookie("cookie")

.AddOpenIdConnect("oidc", options =>

{

options.Authority = "https://localhost:5001";

options.ClientId = "oidcClient";

options.ClientSecret = "Secret";

options.ResponseType = "code";

options.UsePkce = true;

options.ResponseMode = "query";

options.Scope.Add("api1.read");

options.SaveTokens = true;

});

## Authorization

Authorization is the process of restricting functionality of a resource from the context. In this scenario of Oauth the context is the client making a request to the resource-an API. The API decides that certain methods should be restricted and can set specific criteria a request must contain to be allowed to call those methods.

First step in adding authorization to an API is going to the configure method in Startup.cs and adding the following call:

app.UseAuthorization();

Furthermore to actually restrict access on functionality of the API one must define the [Authorize] attribute. This can be upon the whole class of the controller or just above specific methods for example.

### Policy based authorization

AspNetCore defines a number of types of authorization but all of them rely on the concept of policy based authorization which uses a requirement, a requirement handler and a pre-configured policy. These policies can be expanded and customised when using policy based authorization.

Including policy based authorization within an API requires the service to be configured in the ConfigureServices method of the Startup.cs within the API. An example of defining 2 policies named “Reader” and “Writer” looks as following:

services.AddAuthorization(options =>

{

options.AddPolicy("Reader", policy =>

{

policy.Requirements.Add(new ScopeRequirement("api1.read"));

});

options.AddPolicy("Writer", policy =>

{

policy.Requirements.Add(new ScopeRequirement("api1.write"));

});

});

To include the authorization policy first upon the services is called AddAuthorization and a new options object is initialised with 2 new policies. For each policy there is a requirement specified which the requesting context must meet. When the context requests authorization a handler is invoked for the specific requirement. The requirement class and its handler look as following:

**Requirement Class**

public class ScopeRequirement : IAuthorizationRequirement

{

public string ScopeValue { get; }

public ScopeRequirement(string scopeValue)

{

ScopeValue = scopeValue;

}

}

**Requirement Handler Class**

public class ScopeHandler : AuthorizationHandler<ScopeRequirement>

{

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, ScopeRequirement requirement)

{

if (!context.User.HasClaim(c => c.Type == "scope"))

{

return Task.CompletedTask;

}

List<Claim> userScopeClaims = context.User.FindAll("scope").ToList();

foreach (Claim claim in userScopeClaims)

{

if (claim.Value.Contains(requirement.ScopeValue))

{

context.Succeed(requirement);

}

}

return Task.CompletedTask;

}

}

The requirement implements the IAuthorizationRequirement interface while the handler extends the AuthorizationHandler<> class. To configure the API to use the requirement handler it must be inserted into the dependency injection pipeline via the ConfigureServices method in the startup.cs.

services.AddSingleton<IAuthorizationHandler, ScopeHandler>();

Finally to actually use the defined policies, where the attribute [Authorize] is called modify it to look as following [Authorize(Policy=”name”)] in this case name would be replaced with Reader or Writer as those are the defined policy names.

For more information on policy based authorization:

<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies?view=aspnetcore-3.1>

# Demo database seeding

Include a call to private method in the Configure method of Startup.cs as following

InitializeDbTestData(app);

Then add the following method into the same class.

private static void InitializeDbTestData(IApplicationBuilder app)

{

using (var serviceScope = app.ApplicationServices.GetService<IServiceScopeFactory>().CreateScope())

{

serviceScope.ServiceProvider.GetRequiredService<PersistedGrantDbContext>().Database.Migrate();

serviceScope.ServiceProvider.GetRequiredService<ConfigurationDbContext>().Database.Migrate();

serviceScope.ServiceProvider.GetRequiredService<ApplicationDbContext>().Database.Migrate();

var context = serviceScope.ServiceProvider.GetRequiredService<ConfigurationDbContext>();

if (!context.Clients.Any())

{

foreach (var client in Clients.Get())

{

context.Clients.Add(client.ToEntity());

}

context.SaveChanges();

}

if (!context.IdentityResources.Any())

{

foreach (var resource in Resources.GetIdentityResources())

{

context.IdentityResources.Add(resource.ToEntity());

}

context.SaveChanges();

}

if (!context.ApiScopes.Any())

{

foreach (var scope in Resources.GetApiScopes())

{

context.ApiScopes.Add(scope.ToEntity());

}

context.SaveChanges();

}

if (!context.ApiResources.Any())

{

foreach (var resource in Resources.GetApiResources())

{

context.ApiResources.Add(resource.ToEntity());

}

context.SaveChanges();

}

var userManager = serviceScope.ServiceProvider.GetRequiredService<UserManager<IdentityUser>>();

var roleManager = serviceScope.ServiceProvider.GetRequiredService<RoleManager<IdentityRole>>();

if (!roleManager.Roles.Any())

{

var adminRole = new IdentityRole("admin");

roleManager.CreateAsync(adminRole).Wait();

roleManager.AddClaimAsync(adminRole, new Claim("permission", "projects.view")).Wait();

roleManager.AddClaimAsync(adminRole, new Claim("permission", "projects.create")).Wait();

roleManager.AddClaimAsync(adminRole, new Claim("permission", "projects.update")).Wait();

var superUserRole = new IdentityRole("superuser");

roleManager.CreateAsync(superUserRole).Wait();

roleManager.AddClaimAsync(superUserRole, new Claim("permission", "projects.view")).Wait();

roleManager.AddClaimAsync(superUserRole, new Claim("permission", "projects.create")).Wait();

var userRole = new IdentityRole("user");

roleManager.CreateAsync(userRole).Wait();

roleManager.AddClaimAsync(userRole, new Claim("permission", "projects.view")).Wait();

}

var listOfRoles = roleManager.Roles.ToList();

if (!userManager.Users.Any())

{

foreach (var testUser in TestUsers.Users)

{

var identityUser = new IdentityUser(testUser.Username)

{

Id = testUser.SubjectId

};

userManager.CreateAsync(identityUser, testUser.Password).Wait();

userManager.AddClaimsAsync(identityUser, testUser.Claims.ToList()).Wait();

foreach (var claim in testUser.Claims.ToList())

{

if (claim.Type == "role")

{

foreach(var role in listOfRoles)

{

if (claim.Value.Contains(role.Name))

{

userManager.AddToRoleAsync(identityUser, role.Name).Wait();

}

}

}

}

}

}

}

}