Identity Server 4 Configuration

1. Basic configuration from the server side

In order for the identity server to be able to read its configuration from the database we added the identity server 4 storage project to the main repository solution. This project contains the entity framework core entities and stores in order to get the necessary data from the database.

The project provides two entity framework core contexts. The operational context and the configuration context. By default, the project does not provide the context for reading user and role related data from a database. For this reason, we added the necessary user related entities to a separate db context that and we named it “IdentityContext”. The first two database context can be instantiated through service extensions that can be found in the storage project.

The storage project can be found in a separate solution under the src folder of the main Identity server repository.

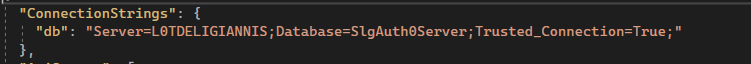
<https://github.com/IdentityServer/IdentityServer4/tree/main/src/EntityFramework.Storage>

The database was initialized with migrations but some necessary new tables were added so running a default migration from the repository will not result the database we currently use. The database that we use for our configuration is in the DB folder, under the host in the src folder of our repository.

[host - Repos (visualstudio.com)](https://singularlogic-ts.visualstudio.com/_git/OAuth2Server?path=/src/IdentityServer4/host)

In order to make the instance of the IdentityServer to read from the database we need to add the necessary code at the startup of the host.

We firstly added the connection string of the database to the appsettings.json file .



We need to change this to our local database. After that we read our connection string and we add the db contexts at the startup.

var cn = \_config.GetConnectionString("db");

We add the operational context:

services.AddOperationalDbContext(options =>

{

options.ConfigureDbContext = b =>

b.UseSqlServer(cn, dbOpts => dbOpts.MigrationsAssembly(typeof(Startup).Assembly.FullName)); });

The configuration context:

services.AddConfigurationDbContext(options =>

{

options.ConfigureDbContext = b =>

b.UseSqlServer(cn, dbOpts => dbOpts.MigrationsAssembly(typeof(Startup).Assembly.FullName));

});

Adding the necessary stores in the startup:

On the builder action that adds the identity server service we add the stores

var builder = services.AddIdentityServer(options =>

{

options.Events.RaiseSuccessEvents = true;

options.Events.RaiseFailureEvents = true;

options.Events.RaiseErrorEvents = true;

options.Events.RaiseInformationEvents = true;

options.EmitScopesAsSpaceDelimitedStringInJwt = true;

options.MutualTls.Enabled = true;

options.MutualTls.DomainName = "mtls";

})

.AddMutualTlsSecretValidators().AddClientStore<ClientStore>()

.AddResourceStore<ResourceStore>()

.AddPersistedGrantStore<PersistedGrantStore>()

.AddDeviceFlowStore<DeviceFlowStore>()

.AddUsers()

The service extensions AddClientStore, AddResourceStore , AddPersistedGrandStore and AddDeviceFlowStore are used as found. We Changed the AddTestUserStore provided in the repository by default with the AddUsers Extension. There were also a number of changes applied in order to make the user functionality work with our database.

The AddUsers Service extension:

To explain the AddUsers service extension we need to provide information about the IdentityContext that we added to the storage project. This context contains the entities related to user data. It contains the classes to handle:

-AspNetRole

-AspNetRoleClaim

-AspNetUser

-AspNetUserClaim

-AspNetUserLogin

-AspNetUserRole

-AspNetUserToken

So After adding this under DBContexts folder at the IdentityServer4.EntityFramework.Storage project we can instantiate this context at the startup of our project.

We add the DBContext here:

services.AddDbContext<IdentityContext>(opts =>

{

opts.UseSqlServer(cn);

opts.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);

});

After that we need to present the changes made in the AddUser functionality the AddUser Service Extension looks like this:

public static IIdentityServerBuilder AddUsers(this IIdentityServerBuilder builder)

{

builder.Services.AddTransient<UserStore>();

builder.AddProfileService<UserProfileService>();

builder.AddResourceOwnerValidator<UserResourceOwnerPasswordValidator>();

return builder;

}

These changes will make our instance use the users roles and claims as configured from our admin configuration tool. Our tool can be found here:

[OAuth2ConfigurationTool - Repos (visualstudio.com)](https://singularlogic-ts.visualstudio.com/OAuth2Server/_git/OAuth2ConfigurationTool)

We also passed we injected the IdentityContext to the necessary controllers that make use of it in order to use the user data to login and assign claims to our server users. We included the functionality of assigning claims to users. The claims are provided straight to the users or via assigned roles.

1. The configuration tool

We developed a tool in order to configure the database we previously mentioned. This tool is a web application written in .net 5 and angular using the company’s preexisting code structure from other projects. We will go through the entities we have to configure presenting photos and providing instructions for sample configurations. At this point we will go through the entities we need to configure and after that we will go through the basic menus of the tool we created for this configuration.

Entities

There are a few basic entities and many detail entities that are used in our configuration. The basic entities that we have are:

1. Clients
2. Api Scopes
3. Api Resources
4. Identity Resources
5. Users
6. Devices
7. Persisted Grants
   1. Clients Entity

The Client entity has a basic table named “Clients”. This table holds the basic information for the client. There is a number of detail tables that also hold information for this entity.

The detail tables for the “client” entity are:

ClientClaims

ClientCorsOrigins

ClientGrandTypes

ClientIDPRestrictions

ClientPostLogoutRedirectUris

ClientProperties

ClientRedirectUris

ClientScopes

ClientSecrets

All these tables are related with the master table “Clients” with a foreign key that is the “ClientId” column of the Clients Table.

*Notice: Client scopes is defining the resources that the client is allowed to access. These resources can be either api scopes or identity resources or api resources. We will analyze where we can define these resources.*

* 1. Api resources

There is a primary table that holds the basic information about the api resource the table is named “ApiResources” there are also some detail tables that hold information about this entity. The detail tables for this entity are as follows:

ApiResourceClaims

ApiResourceProperties

ApiResourceScopes

ApiResourceSecrets

The configuration of this entity will tie up with the allowed scopes property of the client entity that that we mentioned before to give access to api resources to the client.

* 1. Api Scopes

There is a primary table that holds the basic information about the api scopes that is named “ApiScopes”. There are also detail tables that hold information about this entity. The detail tables for this entity are as follows:

ApiScopeClaims

ApiScopeProperties

This table will also tie up with the allowed scopes property of the client to give access of api scopes to the client.

* 1. Identity Resources

The primary table that holds information about the Identity Resources is named “IdenityResources”. This table holds information about resources that are related to user data. There are also some detail tables that are associated with this table. These tables are:

IdentityResourceClaims

IdentityResourceProperties

This entity will also be connected with the client entity through the allowed scopes property to give access to user data to the client when the flow allows this.

* 1. Users

Some flows require user authentication. For this reason, user information and user logins have to be able be stored in the configuration database. The user table is .net identity compliant. The basic entity information is kept in the “AspNetUsers” table. There are also some related tables that hold claims, logins and roles about the user. These tables are:

AspNetRoles

AspNetUserRoles

AspNetRoleClaims

AspNetUserLogins

AspNetUserTokens

* 1. Devices

For the device flow it is necessary to keep information about the devices we want to authorize. The only table that holds information about devices is the “DeviceCode” table.

* 1. Persisted Grants

In some flows the user is asked to give access to the resources of an api. The user can give access and is also able to select the “remember” option. In this case the choice will be saved in the database and the user won’t be asked again as soon as the current session is still active. The information about this choice is hold in the “PersistedGrants” table.

*Notice: This is not a table that is related to the configuration but it is part of the schema and we have to mention it.*

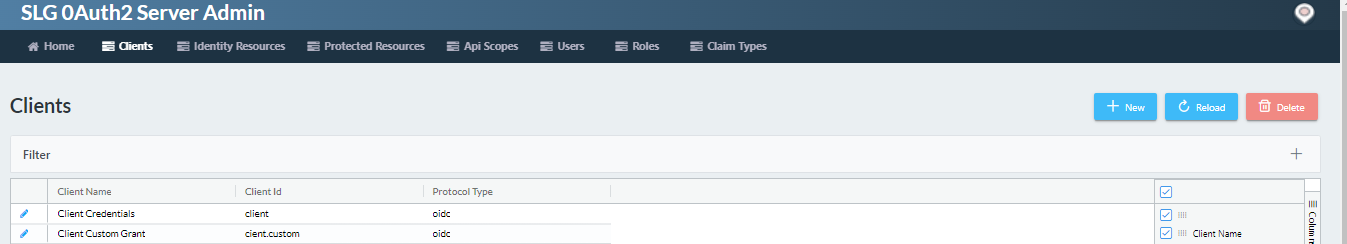
3 Sample Configuration flow

It is very helpful to go through a sample configuration menus of the identity server using the existing sample UI, to understand the flow that the server administrator has to follow in order to activate a client flow and the users that will be eligible to take part in this authorization procedure. In the previous section we analyzed the entities that take part in this procedure and now we will present how these entities get initialized and how this results to records in the database.

As a starting point for this step by step guide we will start from the welcome screen the “Admin UI” Application.

Notice: *The resulting records that appear on this document are from the sqlite database that the Admin UI uses but can be easily mapped to the sql server database that we will use from now on. We also assume that all the actions described in the following configuration flows are done by the identity server administrator.*

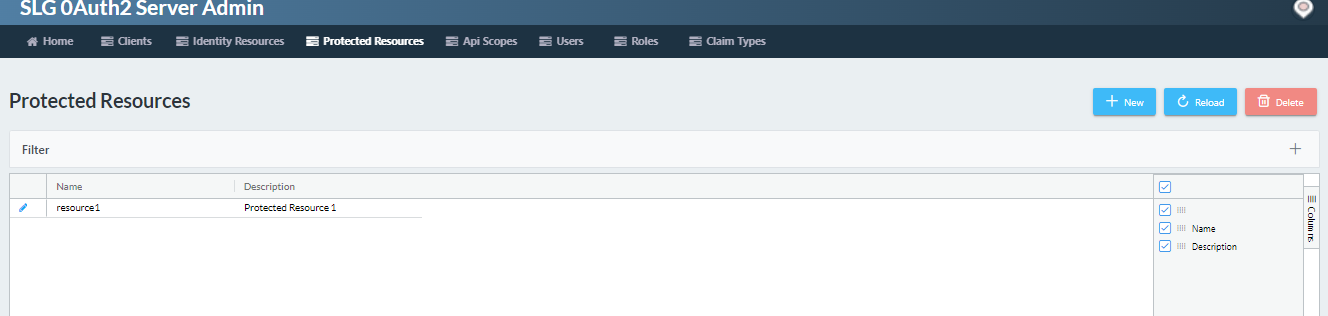
The Menu of the application looks like this:



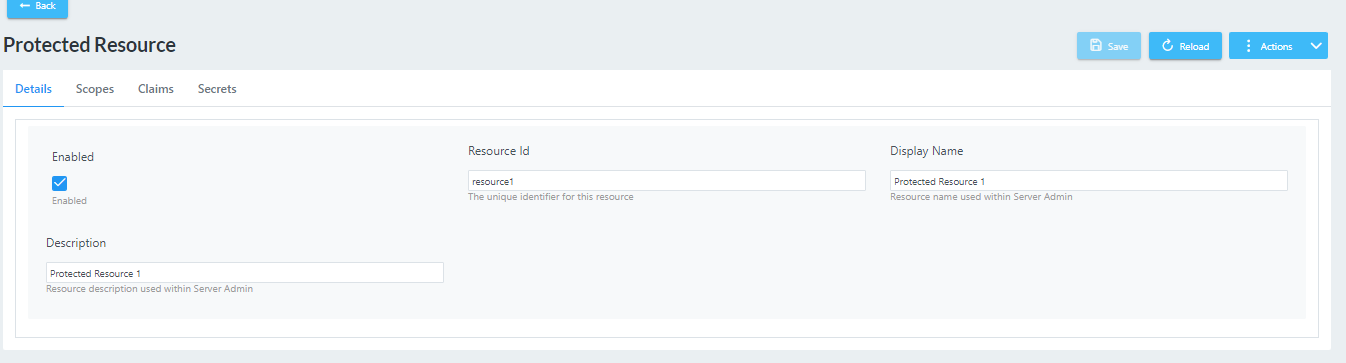
2.1 Configuring the Protected Resources

As we mentioned in the entities section in order to start configuring a client we have to configure the resources first. We start from the “ApiResources” entity this entity is configured through the path Resources => Protected Resources in the menu.

By selecting this option, the index page of the protected or api resources will appear on the right upper side we can find the option to add a new resource.



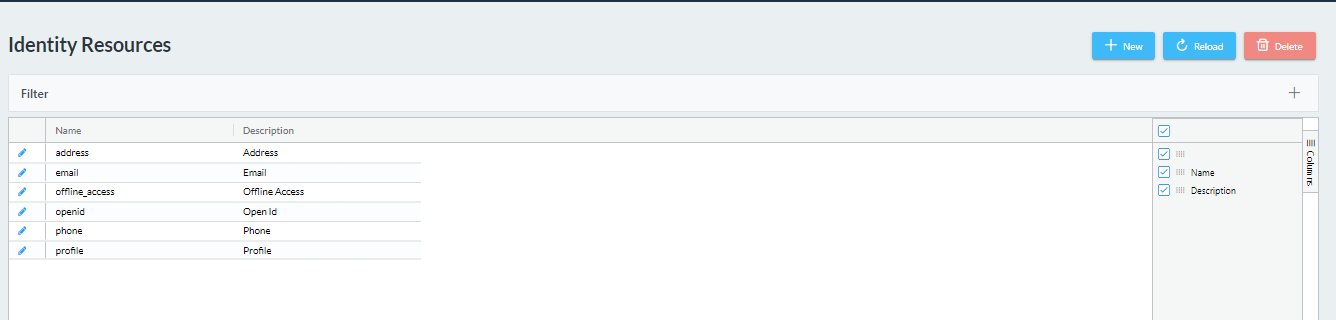
By clicking the new button the next page will appear:

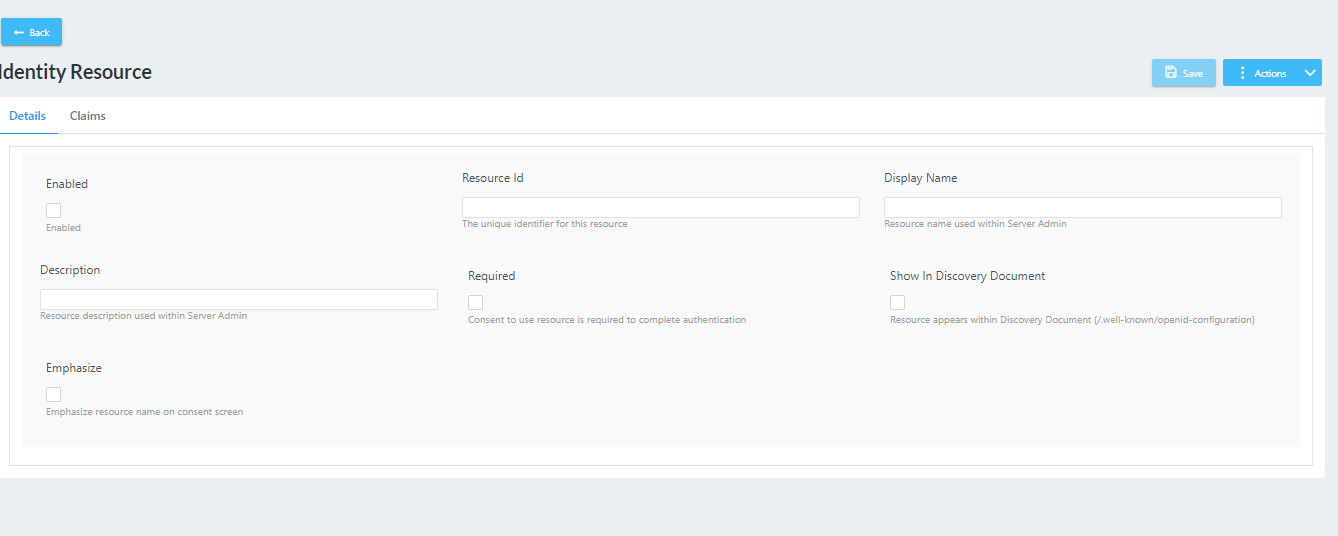


We can add an api resource through this menu. After that we can edit this record and add scopes, claims and secrets for this resource. This procedure will create a record in ApiResources table. The additional information will be added in ApiScopes , ApiClaims , ApiProperties , ApiSecrests respectively.

2.2 Configuring Identity Resources

By selecting the identity resources choice in the menu we get a similar window to add an identity resource.





After adding the identity resource, we can edit it and add claims.

The previous actions will create records in IdentityResources, IdentityClaims, IdentityProperties tables respectively.

The claims that can be added to Identity and Protected resources are maintained through the Claim Types menu.

2.3 Clients

When we have completed setting up the resources both protected and identity ones we can continue by configuring the records that hold the information about the clients that can access these resources. By selecting the choice to add a new client, a wizard is started that guides us through the procedure. For the needs of this small tutorial we will configure an mvc code client. There is an in memory configuration for this client that we will replicate through the configuration tool. The object of the client that we will create will look like this:

new Client

{

ClientId = "mvc.code",

ClientName = "MVC Code Flow",

ClientUri = "http://identityserver.io",

ClientSecrets =

{

new Secret("secret".Sha256())

},

RequireConsent = true,

AllowedGrantTypes = GrantTypes.Code,

RedirectUris = { "https://localhost:44302/signin-oidc" },

FrontChannelLogoutUri = "https://localhost:44302/signout-oidc",

PostLogoutRedirectUris = { "https://localhost:44302/signout-callback-oidc" },

AllowOfflineAccess = true,

AllowedScopes = allowedScopes

}

And the allowed scopes data structure looks like this:

static string[] allowedScopes =

{

IdentityServerConstants.StandardScopes.OpenId,

IdentityServerConstants.StandardScopes.Profile,

IdentityServerConstants.StandardScopes.Email,

"resource1.scope1",

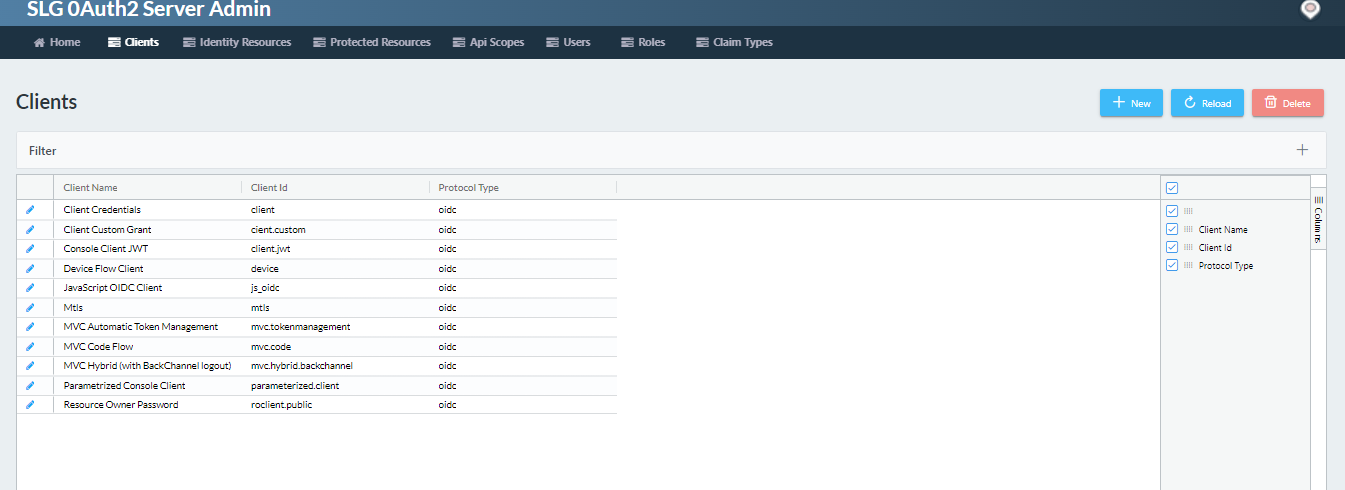
"resource2.scope1",

"transaction"

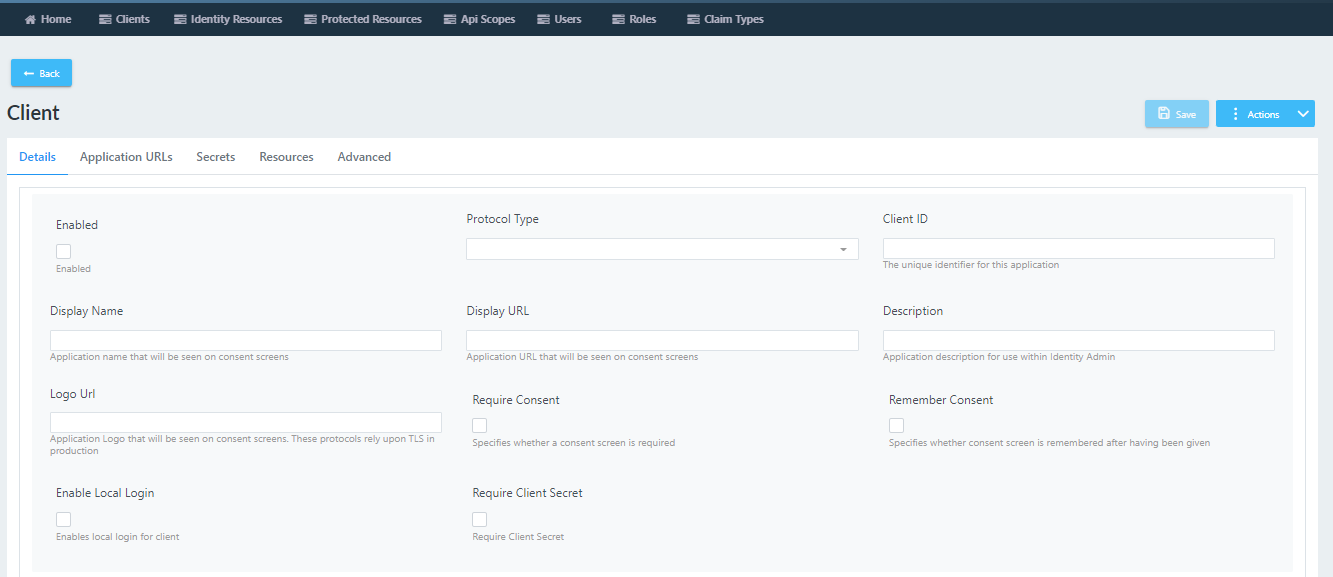
};

As we see this client will have access to both identity and protected resources. These resources must be added before starting the client configuration.

Firstly, we have to select the client element on the menu and click the new button.



We add name and other details for the client we want to add.

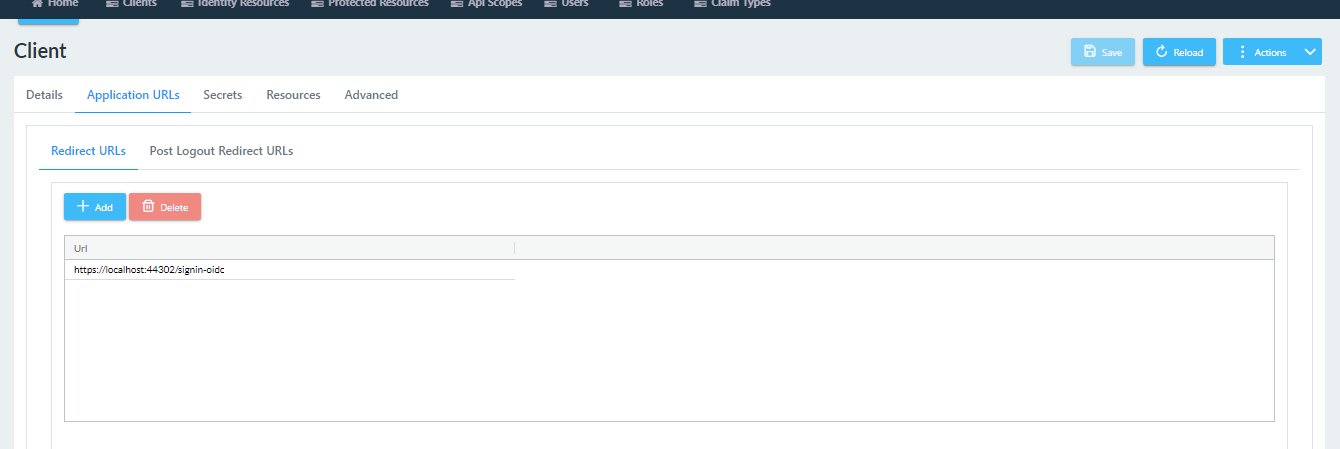


We select if the client is enabled or not. If the client is not enabled the server will refuse to serve requests for this client.

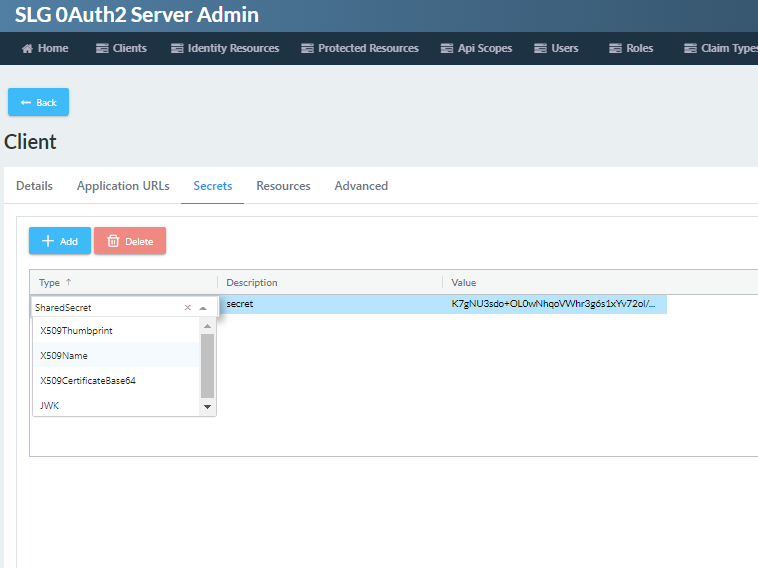
At the protocol type option, the user is opted to select between open id protocol, saml 2 and ws fed choices. The client id string value determines the unique name that the client application will use to access our configuration. The display name is the name that will be used in prompt pages by the server during the user interaction. We can also find the logo url configuration, we can determine if the server will require consent by the user and if will remember it. The ”enable local login” checkbox will enable the login page for the user. We can also specify if the client secret is required for the current client.

At the application urls submenu we can find a list of “Redirect Urls” and a list of “Post Logout Urls”.

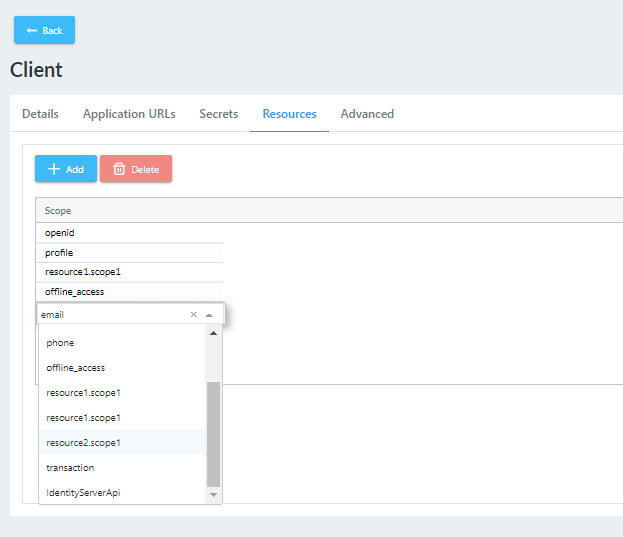
The fist list contains the urls that the client application will be redirected after the successful negotiation and the second one refers to the urls that the client application will be redirected after the ending of its session with the identity server.



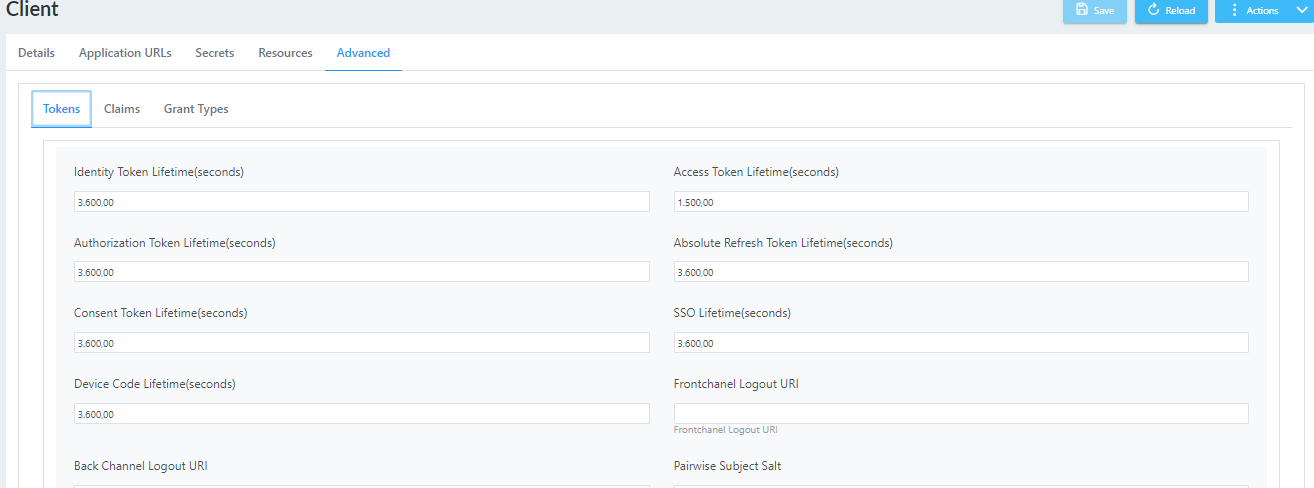
At the next tab of our menu we can add the secrets that our client will use to make the negotiation with our server. We have several choices of secret types. The “shared secret” choice will calculate and store a hash of the secret at the value field. All the other types of secret will be stored as is.



At the resources submenu we can add the choices of the resources that the client will have access to. At the dropdown menu we can find the sum of the identity resources and the protected resource scopes we have added.

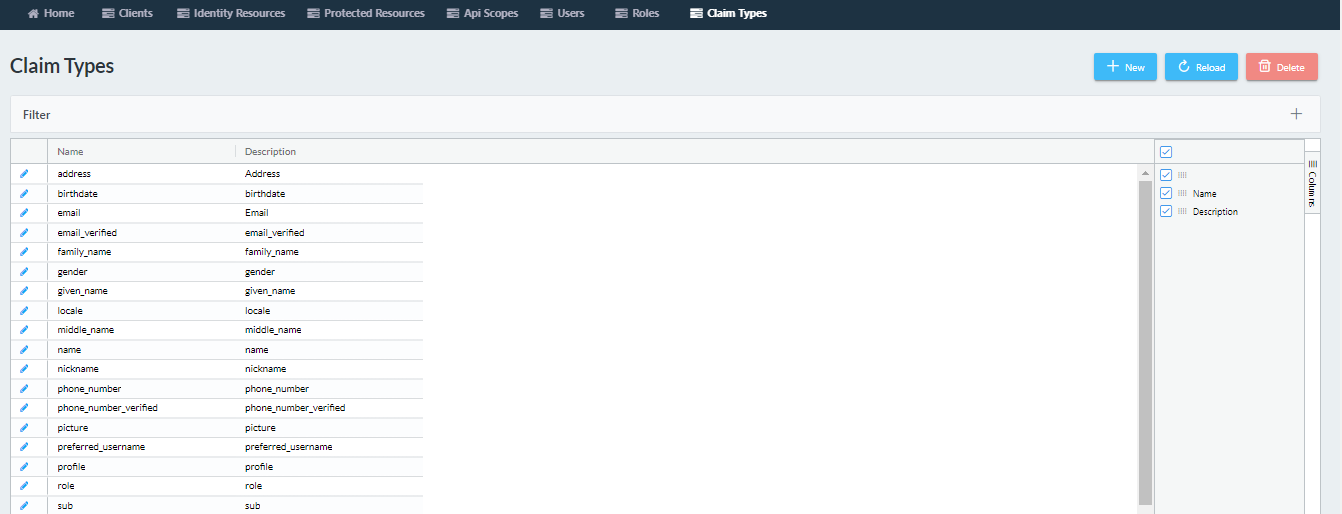


At the advanced menu of the client we can find the token specific configuration we can also find submenus to add claims and grand types.



As we see at the screenshot above we can configure the token related times and the token types. The grand type specifies the type of negotiation that will happen between the server and the client.

Many instances use claims so we added a separate menu where we can add the claim types that these entities can use. By clicking at the claims menu we can add delete and modify the claim types.



After adding the client, we can edit the record to modify its behavior. The previous procedure will create a record in “Clients” table. Records will also be created in:

ClientClaims

ClientCorsOrigins

ClientGrandTypes

ClientIDPRestrictions

ClientPostLogoutRedirectUris

ClientProperties

ClientRedirectUris

ClientScopes

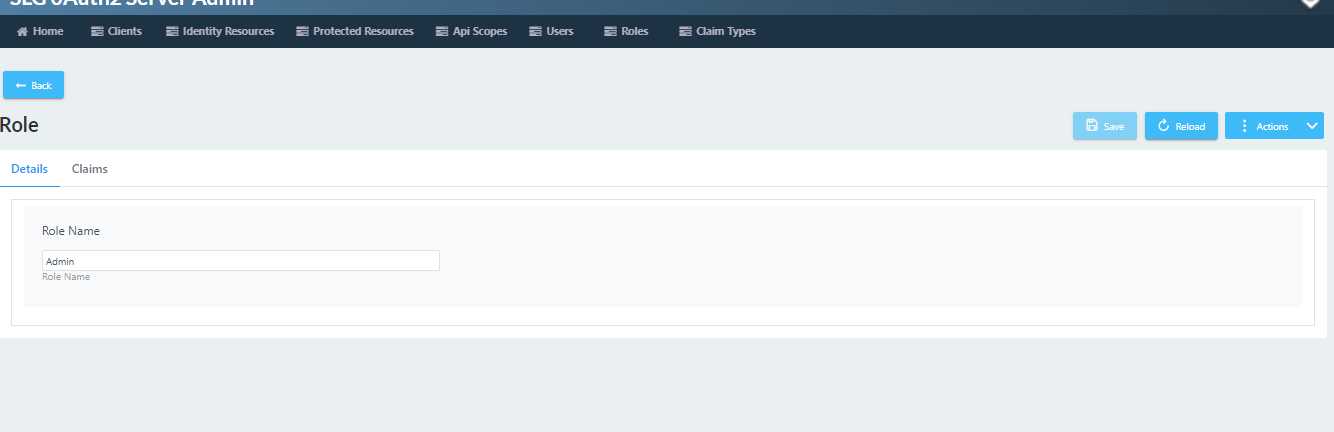
ClientSecrets

In a similar way we can add other resources and clients to support different authorization flows in our identity server 4 installation.

2.4 User Related data

Some of the authorization flows require actions from an eligible user in order for the client application to get authorized. For this reason, the configuration database need to hold information about users, claims and roles in order to support this functionality. The claim types are implemented as mentioned above and is the same for all the entities of the database.

There are two more items implemented for this functionality. The “User” menu item and the “Roles” menu item. At the Role menu we can edit add and delete a role by editing the name and the claims.



At the user menu we can add delete and edit users by editing the username, password, claims and roles.

