LIMITED BUNDLE EDITION

Marinko Spasojevic
Vladimir Pecanac

MASTERING ASP.NET CORE SECURITY

With IdentityServer 4/Duende, OpenID Connect, OAuth2 and ASP.NET Core Identity for User Management

Made with ❤ by: CodeMaze

1 Introduction

Modern web applications have changed their landscape a lot during the recent period. We build more applications where the API part and the client part aren't on the same domain. Moreover, we have applications where, for example, one API communicates with a different API, or the client application communicates with more than one API. Also, we have a great variety of client applications built with Angular or React or even MVC web app that communicate with some API, etc.

When we face a landscape such as the one previously described, it is not enough to have security implemented inside our main application or our API. As we can see, there can be multiple clients, and having the security functionality in our API can't satisfy the security requirements for all the different client applications.

Additionally, we need to keep in mind that using forms on client applications that accept a username and a password and send them with the HTTP request to the server is pretty insecure. That's something we should never do if we have a public API that requires clients to be authenticated to access its resources. If an attacker steals the password, there's more than just the API access at stake because – let's be honest – users tend to reuse their passwords on different systems/applications.

Of course, if we have a private API (API that provides resources to a single trusted client) then using this approach with the username and password is still acceptable.

Now, we may wonder what the security solution for this landscape is – when we have a public API and multiple third-party clients? This leads us to token-based security. A token is related to consent – the user grants consent to a client application to access the API on behalf of that user. Then, the token is sent with the HTTP requests and not the username and password.

So, we want to prevent two things here.

The first one is to no longer use client credentials at the application level because this should be handled on a centralized server. The second thing is to ensure that tokens are safe enough for the authentication and authorization actions for different types of applications.

That said, all of these questions lead us to the centralized Identity Provider that should be responsible for the token handling and proving the user's identity.

Using a centralized provider helps us centralize authentication and authorization operations, provide tokens, refresh tokens, etc. Additionally, the actions related to user management should be centralized as well because we want to use them for multiple clients and if something changes in the implementation, we have to make changes only in one centralized place.

We should point out that using Identity Server4, OAuth2 and OIDC will help us with the authentication and authorization actions, but for the user management actions, we have to integrate ASP.NET Core Identity at the centralized level as well.

So, a lot of ground is ahead of us, but if you follow along with this book and its examples, we are sure you will master security actions in ASP.NET Core applications.

We'll help you do that by using both **IdentityServer4** and **Duende**. We will show code examples **for IS4 first and then Duende right after** so you could see the main differences. Since IS4 uses MVC concept, whenever we mention the creation or modification of Actions or Controllers, you should know that we are working with the IS4 project.

On the other hand since Duende uses Razor Pages for UI, we will create and modify Pages and Page classes.

Since the examples and code snippets will be shown one after another, you will be able to see differences in the implementation between two concepts (MVC and Razor pages) but also the similarity in the business logic itself.

This just proves we can use different project types for authentication and authorization, but the business logic is always the same, and you can implement it in any project type you want.

2 About Tokens, OAuth 2, and OpenID Connect

Before we start learning about OAuth and OpenID Connect, we have to understand what a token is. If we want to access a protected resource, the first thing we have to do is to retrieve a token. When we talk about token-based security, most of the time we refer to the JSON web token (JWT). We've learned about JWT in our Ultimate ASP.NET Core Web API book, but let's just recall a couple of things.

For secure data transmission, we use JSON objects. A JWT consist of three basic parts: the header, the payload, and the signature. The header contains information like the type of token and the name of the algorithm. The payload contains some attributes about the logged-in user. For example, it can contain the user id, subject and information whether a user is an administrator. Finally, we have the signature part. Usually, the server uses this signature part to verify whether the token contains valid information – the information that the server is issuing.

Now, let's see how we exchange our credentials for a token:

- The user provides credentials to the authorization server and the server responds with a token.
- After that, the user can use that token to talk to the API and retrieve the required data. Of course, behind the scenes, the API will validate that token and decide whether the user has access to the requested endpoint.
- Finally, the user can use that token with a third-party application that communicates with the API and retrieves data from it. Of course, the third-party application has to provide the token to the API via headers.

After the token validation, the API provides data to the client application and that client application returns data to the user.

As we can see, we are not using our credentials with the third-party application. Instead, we use a token, which is certainly a more secure way.

2.1 Understanding OAuth2 and OpenID Connect

OAuth2 and OpenID Connect are protocols that allow us to build more secure applications. OAuth stands for Open standard for Authorization. It is the industry-standard protocol for authorization. It delegates user authentication to the service that hosts the user’s account and authorizes third-party applications to access that account. It provides different flows for our applications, whether they are web, desktop, or mobile applications. Additionally, it defines how a client application can securely get a token from the token provider and use it for authorization actions.

With authorization, we prove that we have access to a certain endpoint. But, if we want to add authentication in the process, we have to refer to OpenID Connect. So, OpenID Connect complements OAuth2 with the authentication part. It is a simple identity layer on top of the OAuth2 protocol that allows clients to verify their identity after they perform authentication on the authorization server. With this protocol, a client can request an identity token, next to the access token and use that id token to sign in to the application while the application uses the access token to access the API. Additionally, we can extract more information about the end-user by using OpenID Connect.

2.2 How OpenID Connect Generates Id Token

Generating an id token is a process that involves the client application and the Identity Provider (IDP). This process contains the following actions:

- The client application creates a request which redirects the user to the IDP, where the user proves their identity by providing their username and password. As we can see here, the user's credentials are not sent via request, but rather the user provides the username and password at the IDP level.
- After the verification process, IDP creates the id token, signs it, and sends it back to the client. This token contains user verification data.
- Finally, the client application gets the id token and validates it.

Now, we have to say that this is a simplified explanation of how OpenID Connect works because there is a lot that's happening behind the scenes. We will come to that but, for now, it is enough to understand the basics.

There is one more thing to mention here. The client application uses the id token to create claims identity and store these claims in a cookie.

2.3 Types of Clients

As we can see, the user can authenticate using the user credentials. But, the client can do the same with the client credentials (client_id and client_secret). If a client is capable of maintaining the confidentiality of its credentials and can safely authenticate – that client is considered a confidential client. It is a confidential client because it stores its credentials on the server inaccessible to the user. For example, an ASP.NET Core MVC application is a confidential client.

If a client can't maintain the confidentiality of its credentials, it is called the public client. These client applications are being executed in the browser and can't safely authenticate. JavaScript applications and mobile applications are considered public clients.

2.4 OAuth2 and OpenID Connect Endpoints and Flows

It is quite important to keep in mind that there is great documentation regarding OAuth 2.0 – RFC 6749 that makes it a lot easier to understand OAuth-related topics. One of those topics is related to OAuth endpoints. So, let’s inspect these endpoints:

- **/authorize** – a client uses this endpoint (Authorization endpoint) to obtain authorization from the resource owner. We can use different flows to obtain authorization and gain access to the API.
- **/token** – a client uses this endpoint to exchange an authorization grant for an access token. This endpoint is used for the token refresh actions as well.
- **/revocation** – this endpoint enables the token revocation action.

OpenID Connect also allows us to perform some additional actions with different endpoints, such as:

- **/userinfo** – retrieves profile information about the end-user
- **/checksession** – checks the session of the current user
- **/endsession** – ends the session for the current user

There are different flows we can use to complete authorization actions:

- Implicit Flow
- Authorization Code
- Resource Owner Password Credentials
- Client Credentials
- Hybrid (mix of Authorization Code and Implicit Flow)

The flow determines how the token is returned to the client and each flow has its specifics.

You can read more about these flows in the documentation mentioned above.

In this book, you will learn how to use the Authorization Code Flow with Proof Key for Code Exchange (PKCE, pronounced pixie) to provide a high level of security for our web application and our API. It is also a recommended flow for web applications.

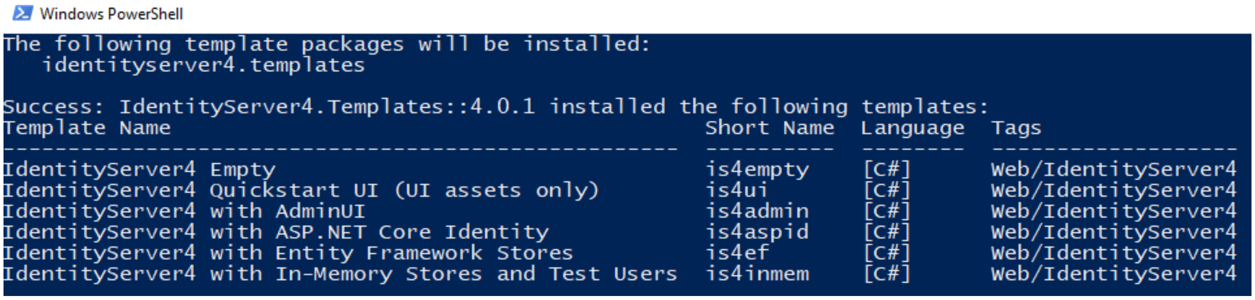
3 Setting up IdentityServer4 and UI

Before we start, we have to install the IdentityServer4 templates that help us speed up the creation process. We are going to use only the basic templates for the empty IDP application and basic UI files, nothing more than that because we want to explain every single step of the security implementation. After you learn all the steps in detail, you can use other templates to create projects with additional features out of the box.

So, let’s start with the template installation:

```
dotnet new install identityserver4.templates
```

This is the result:



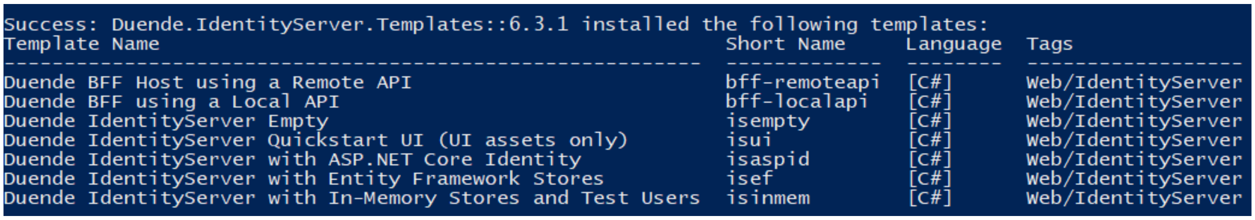
Windows PowerShell

```
The following template packages will be installed:
   identityserver4.templates

Success: IdentityServer4.Templates::4.0.1 installed the following templates:
Template Name                                        Short Name  Language  Tags
---------------------------------------------------  ----------  --------  -------------------
IdentityServer4 Empty                                is4empty    [C#]      Web/IdentityServer4
IdentityServer4 Quickstart UI (UI assets only)       is4ui       [C#]      Web/IdentityServer4
IdentityServer4 with AdminUI                         is4admin    [C#]      Web/IdentityServer4
IdentityServer4 with ASP.NET Core Identity           is4aspid    [C#]      Web/IdentityServer4
IdentityServer4 with Entity Framework Stores         is4ef       [C#]      Web/IdentityServer4
IdentityServer4 with In-Memory Stores and Test Users is4inmem    [C#]      Web/IdentityServer4
```

For Duende the command is slightly different:

```
dotnet new install Duende.IdentityServer.Templates
```

The result:



```
Success: Duende.IdentityServer.Templates::6.3.1 installed the following templates:
Template Name                                             Short Name     Language  Tags
--------------------------------------------------------  -------------  --------  ------------------
Duende BFF Host using a Remote API                        bff-remoteapi  [C#]      Web/IdentityServer
Duende BFF using a Local API                              bff-localapi   [C#]      Web/IdentityServer
Duende IdentityServer Empty                               isempty        [C#]      Web/IdentityServer
Duende IdentityServer Quickstart UI (UI assets only)      isui           [C#]      Web/IdentityServer
Duende IdentityServer with ASP.NET Core Identity          isaspid        [C#]      Web/IdentityServer
Duende IdentityServer with Entity Framework Stores        isef           [C#]      Web/IdentityServer
Duende IdentityServer with In-Memory Stores and Test Users isinmem       [C#]      Web/IdentityServer
```

We can see multiple templates installed with the Short Name values that we can use to create our projects.

Now, let's create an empty IDP project:

```
dotnet new is4empty -n CompanyEmployees.IDP
```

Of course, it is a bit different command for Duende

```
dotnet new isempty -n CompanyEmployees.IDP
```

After creation completes, let's open the project and modify the **launchsettings.json** file:

```
{
  "profiles": {
    "SelfHost": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "https://localhost:5005"
    }
  }
}
```

Since the created project from a template targets .NET Core 3.1, we are going to update that and the preinstalled libraries.

That said, let's modify the `.csproj` file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="IdentityServer4" Version="4.1.2" />

    <PackageReference Include="Serilog.AspNetCore" Version="6.0.1" />
  </ItemGroup>
</Project>
```

Here, we are targeting .NET 6.0 and using the latest versions of preinstalled libraries.

A very important note. IdentityServer 4 is not maintained after .NET Core version 3.1. So, we have to upgrade the version manualy. But we are using .NET 6 and not .NET 7 because if you upgrade the IdentityServer4 project to .NET 7, you will face issues in the migration part once we start working with EF Core. These issues are result of using .NET 7 and an older AutoMapper version.

Due to the lack of maintenance for Identity Server4 after the .NET Core 3.1, the `IdentityServer4.EntityFramework.Storage` library uses the `AutoMapper` version 10, which produces these issues when we upgrade the project to .NET 7.

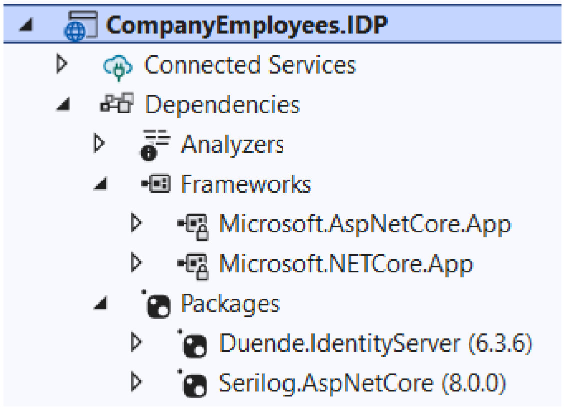
Automapper issues are fixed with later releases, but this doeasn't affect IS4, and we are stuck with the previous AutoMapper version.

On the other side, we can update the Duende project to the latest version:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Duende.IdentityServer" Version="6.3.6" />

    <PackageReference Include="Serilog.AspNetCore" Version="8.0.0" />
  </ItemGroup>
</Project>
```

Right after the modification, we can inspect all the files this template provides us with.

Let’s inspect the Dependencies:



CompanyEmployees.IDP
Connected Services
Dependencies
Analyzers
Frameworks
Microsoft.AspNetCore.App
Microsoft.NETCore.App
Packages
Duende.IdentityServer (6.3.6)
Serilog.AspNetCore (8.0.0)

We can see this project references ASP.NET Core in the Frameworks section and it has installed the **DuendeIdentityServer** library and **Serilog.ASPNetCore** library for logging. It is a similar situation with the IS4 project.

Additionally, we are going to open the **Config** class and modify it a bit:

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

    public static IEnumerable<ApiScope> ApiScopes =>
        new ApiScope[]
        { };

    public static IEnumerable<ApiResource> Apis =>
     new ApiResource[]
            { };

    public static IEnumerable<Client> Clients =>
        new Client[]
        { };
 }
```

As we can see, this is a static class with a couple of properties. For the Ids, we have added one more resource – IdentityResources.Profile. Identity resources map to scopes that enable access to identity-related information. With the OpenId method, support is provided for a subject id or sub value.

With the Profile method, support for information like given_name or family_name is provided.

Additionally, we are going to use the **ApiScopes** and **ApiResource** arrays to configure APIs and the Client array to configure our clients. But for now, we are going to leave them as is.

Now, let's inspect the **Startup.cs** class.

Duende has a bit different structure. Since it doesn't have the **Startup** class but only the **Program** class, both the **ConfigureServices** and the **Configure** methods (**ConfigurePipeline** is the current name) are now extensions methods:

```
var app = builder
    .ConfigureServices()
    .ConfigurePipeline();
```

First, let's take a look at the **ConfigureServices** method:

```
public void ConfigureServices(IServiceCollection services)
{
    // uncomment if you want to add an MVC-based UI
    //services.AddControllersWithViews();

    var builder = services.AddIdentityServer (options =>
    {
        options.EmitStaticAudienceClaim = true;
    })
      .AddInMemoryIdentityResources(Config.Ids)
      .AddInMemoryApiScopes(Config.ApiScopes)
      .AddInMemoryApiResources(Config.Apis)
      .AddInMemoryClients(Config.Clients);

    // not recommended for production - you need to store your key material somewhere
secure
        builder.AddDeveloperSigningCredential();

}
```

For now, we are going to skip the code that is commented out. We'll get back to that later. Even though the code for Duende is slightly different, the crucial part is the **AddIdentityServer** method that we use to register

IdentityServer in our application. With additional methods, we register identity resources, APIs, and clients inside the in-memory configuration from the Config class.

Lastly, we can see the **AddDeveloperSigningCredential** method which sets temporary signing credentials. You won't find this line in the Duende template.

In the `Configure` method (in Duende it is `ConfigurePipeline`), the only thing relevant to us right now is the **app.UseIdentityServer();** expression. This will add the IdentityServer to the application's request pipeline.

Now, let's navigate inside the CompanyEmployees.IDP folder and run: `dotnet new is4ui` command to generate the UI folders.

For Duende we need: `dotnet new isui`

If we want, we can create in-memory user storage to use until we integrate the ASP.NET Core Identity library for user management actions. To create such storage, we have to modify the `TestUsers` class in the `QuickStart` folder or `Pages` for Duende, by removing current users and adding new ones:

```
public static List<TestUser> Users =>
    new List<TestUser>
    {
        new TestUser
        {
            SubjectId = "a9ea0f25-b964-409f-bcce-c923266249b4",
            Username = "John",
            Password = "JohnPassword",
            Claims = new List<Claim>
            {
                new Claim("given_name", "John"),
                new Claim("family_name", "Doe")
            }
        },
        new TestUser
        {
            SubjectId = "c95ddb8c-79ec-488a-a485-fe57a1462340",
            Username = "Jane",
            Password = "JanePassword",
            Claims = new List<Claim>
            {
                new Claim("given_name", "Jane"),
                new Claim("family_name", "Doe")
            }
        }
    };
```

To support the TestUser and Claim classes, additional namespaces are included:

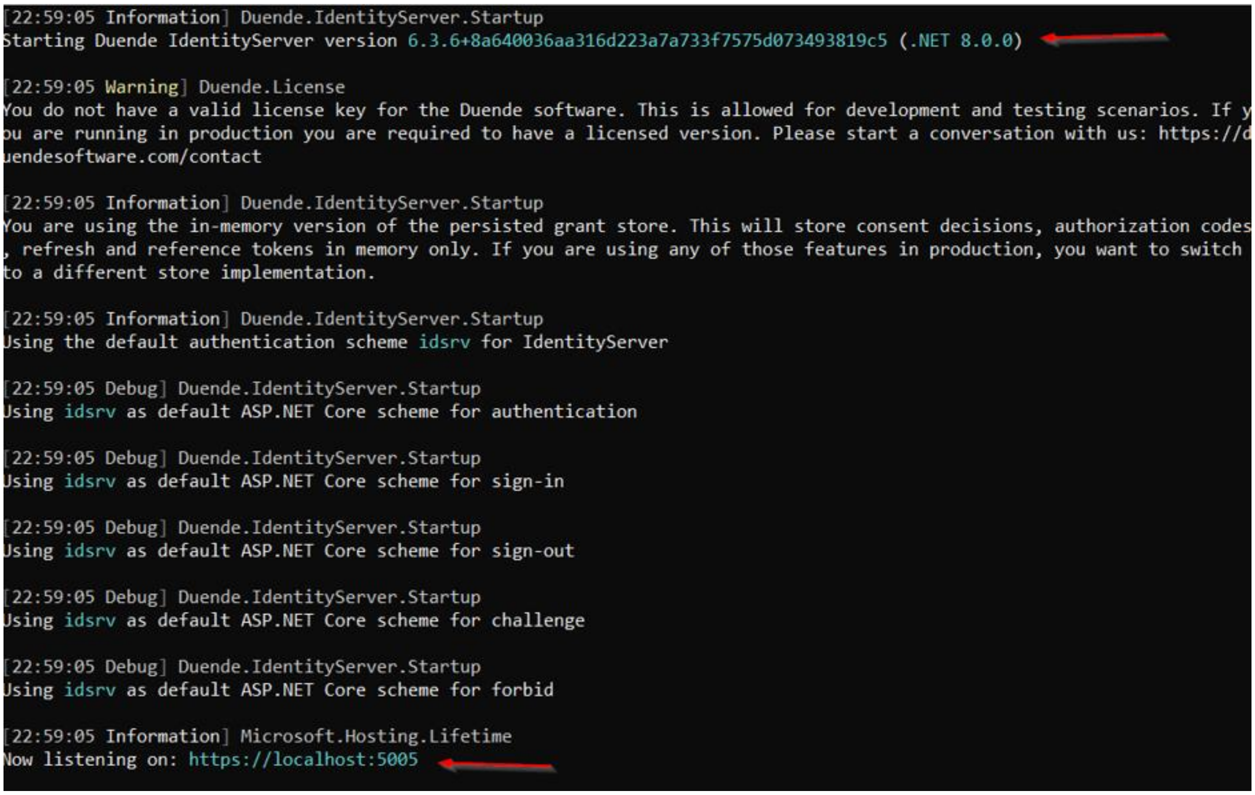
```
using System.Security.Claims;
using IdentityServer4.Test;
```

As we can see, these users have **SubjectId** supported by the **OpenId** IdentityResource and the **given_name** and **family_name** claims supported by the **Profile** IdentityResource.

Now, we have to add these test users to the configuration as well:

```
var builder = services.AddIdentityServer(options =>
{
    options.EmitStaticAudienceClaim = true;
})
.AddInMemoryIdentityResources(Config.Ids)
.AddInMemoryApiScopes(Config.ApiScopes)
.AddInMemoryApiResources(Config.Apis)
.AddInMemoryClients(Config.Clients)
.AddTestUsers(TestUsers.Users);
```

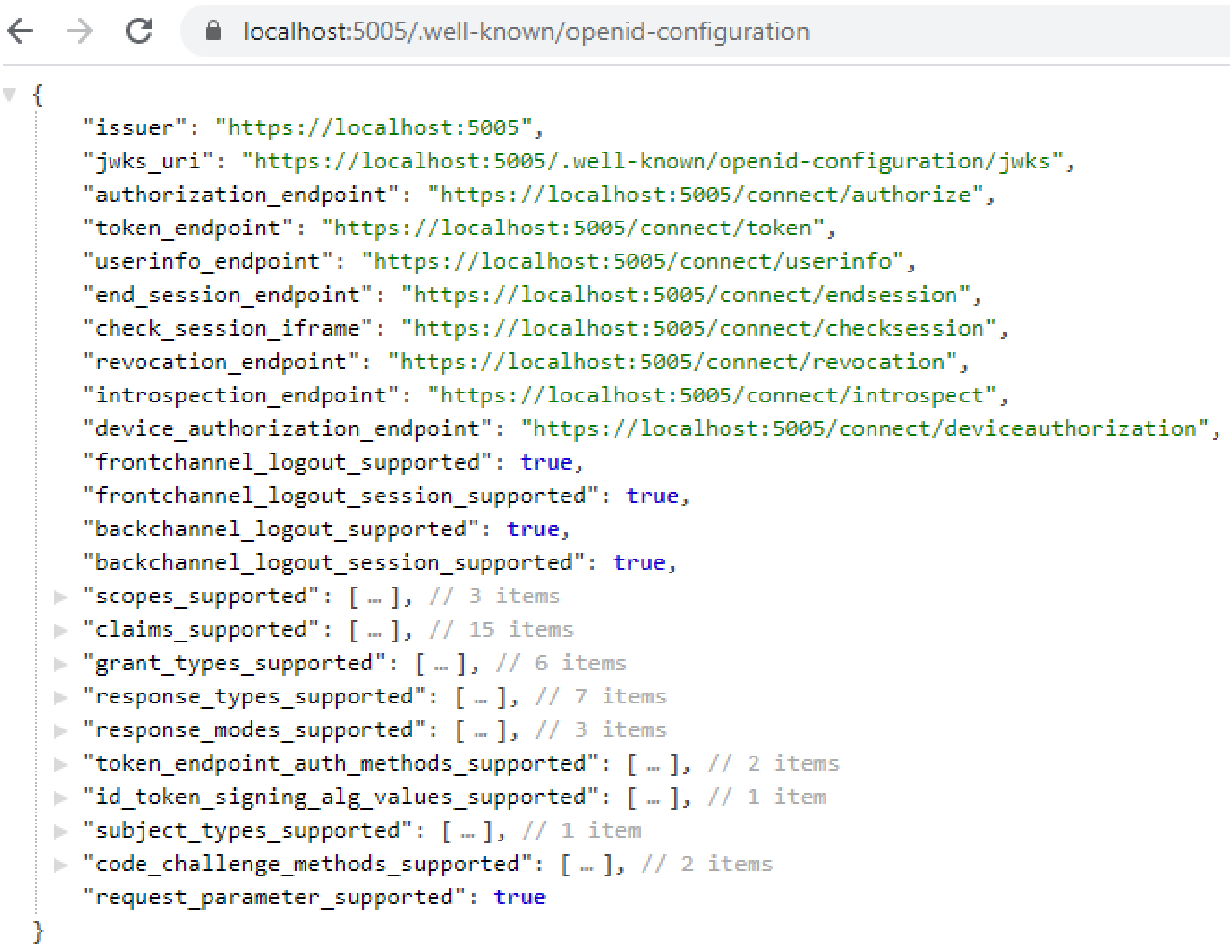
Great. Let’s start the application and inspect the console logs:



```
[22:59:05 Information] Duende.IdentityServer.Startup
Starting Duende IdentityServer version 6.3.6+8a640036aa316d223a7a733f7575d073493819c5 (.NET 8.0.0)

[22:59:05 Warning] Duende.License
You do not have a valid license key for the Duende software. This is allowed for development and testing scenarios. If y
ou are running in production you are required to have a licensed version. Please start a conversation with us: https://d
uendesoftware.com/contact

[22:59:05 Information] Duende.IdentityServer.Startup
You are using the in-memory version of the persisted grant store. This will store consent decisions, authorization codes
, refresh and reference tokens in memory only. If you are using any of those features in production, you want to switch
to a different store implementation.

[22:59:05 Information] Duende.IdentityServer.Startup
Using the default authentication scheme idsrv for IdentityServer

[22:59:05 Debug] Duende.IdentityServer.Startup
Using idsrv as default ASP.NET Core scheme for authentication

[22:59:05 Debug] Duende.IdentityServer.Startup
Using idsrv as default ASP.NET Core scheme for sign-in

[22:59:05 Debug] Duende.IdentityServer.Startup
Using idsrv as default ASP.NET Core scheme for sign-out

[22:59:05 Debug] Duende.IdentityServer.Startup
Using idsrv as default ASP.NET Core scheme for challenge

[22:59:05 Debug] Duende.IdentityServer.Startup
Using idsrv as default ASP.NET Core scheme for forbid

[22:59:05 Information] Microsoft.Hosting.Lifetime
Now listening on: https://localhost:5005
```

Our IDP application is up and running, everything is looking good. But, as soon as we inspect the browser, we are going to see a "page not found" message. So, we need a UI for our IDP server.

But, there is one thing we can do. We can navigate to **https://localhost:5005/.well-known/openid-configuration**:



localhost:5005/.well-known/openid-configuration

```
{
    "issuer": "https://localhost:5005",
    "jwks_uri": "https://localhost:5005/.well-known/openid-configuration/jwks",
    "authorization_endpoint": "https://localhost:5005/connect/authorize",
    "token_endpoint": "https://localhost:5005/connect/token",
    "userinfo_endpoint": "https://localhost:5005/connect/userinfo",
    "end_session_endpoint": "https://localhost:5005/connect/endsession",
    "check_session_iframe": "https://localhost:5005/connect/checksession",
    "revocation_endpoint": "https://localhost:5005/connect/revocation",
    "introspection_endpoint": "https://localhost:5005/connect/introspect",
    "device_authorization_endpoint": "https://localhost:5005/connect/deviceauthorization",
    "frontchannel_logout_supported": true,
    "frontchannel_logout_session_supported": true,
    "backchannel_logout_supported": true,
    "backchannel_logout_session_supported": true,
  ▶ "scopes_supported": [ … ], // 3 items
  ▶ "claims_supported": [ … ], // 15 items
  ▶ "grant_types_supported": [ … ], // 6 items
  ▶ "response_types_supported": [ … ], // 7 items
  ▶ "response_modes_supported": [ … ], // 3 items
  ▶ "token_endpoint_auth_methods_supported": [ … ], // 2 items
  ▶ "id_token_signing_alg_values_supported": [ … ], // 1 item
  ▶ "subject_types_supported": [ … ], // 1 item
  ▶ "code_challenge_methods_supported": [ … ], // 2 items
    "request_parameter_supported": true
}
```

As the URI states, this is the OpenID Configuration. Here, we can see who the issuer is, different endpoints, supported claims and scopes, etc.

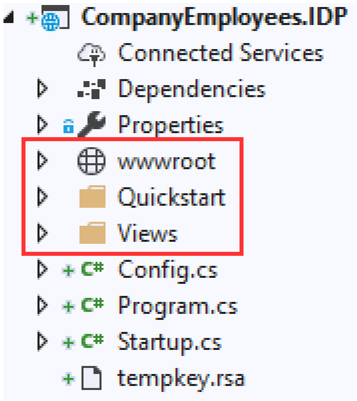
Now, let's inspect the UI for our IDP application.

3.1 Installing UI for the IDP Application

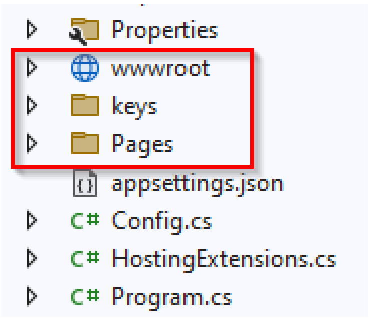
We have already installed the UI in our application by using the following commands:

```
dotnet new is4ui    OR    dotNet new isui
```

So, what this does is create additional folders with controllers, views, and static files:



For the Duende project, we have the `Pages`, and `keys` folders. This project uses the razor pages structure, thus the folders are organized a bit differently:



Now, we have to uncomment the code inside the `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    // uncomment, if you want to add an MVC-based UI
    services.AddControllersWithViews();

    var builder = services.AddIdentityServer(options =>
    {
        options.EmitStaticAudienceClaim = true;
    })
    .AddInMemoryIdentityResources(Config.Ids)
    .AddInMemoryApiResources(Config.Apis)
    .AddInMemoryClients(Config.Clients)
    .AddTestUsers(Config.Users);

    // not recommended for production - you need to store your key material somewhere
secure
    builder.AddDeveloperSigningCredential();
}
```

For Duende, as we said because it is using the Razor pages, we have a slightly different code:

```
public static WebApplication ConfigureServices(this WebApplicationBuilder builder)
{
    // uncomment if you want to add a UI
    builder.Services.AddRazorPages();
```

And in the **Configure** (**ConfigurePipeline**) method:

```
public void Configure(IApplicationBuilder app)
{
    if (Environment.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();
    app.UseRouting();

    app.UseIdentityServer();

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

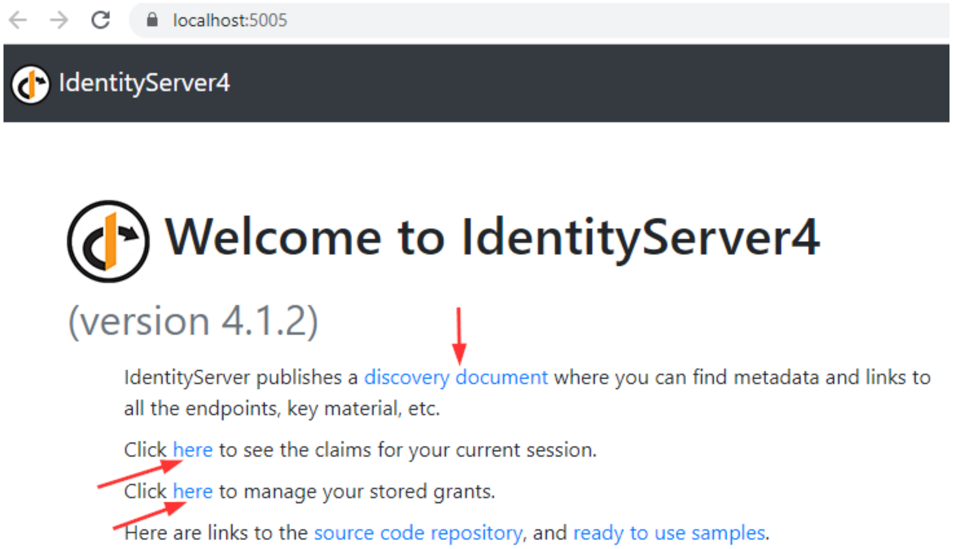
With the **UseStaticFiles** method, we enable serving static files from the **wwwroot** folder. Additionally, we are adding routing and authorization to the pipeline and configuring endpoints to use a default **/Home/Index** endpoint. If we open the **Quickstart/Home** folder, we are going to find a **HomeController** with the **Index** action inside. So, this is our entry point.

For Duende the last part is different due to the Razor pages usage:

```
app.UseAuthorization();
app.MapRazorPages().RequireAuthorization();
```

Also, the starting point is **Pages/Index.cshtml** page.

We can start our application:



Welcome to IdentityServer4

(version 4.1.2)

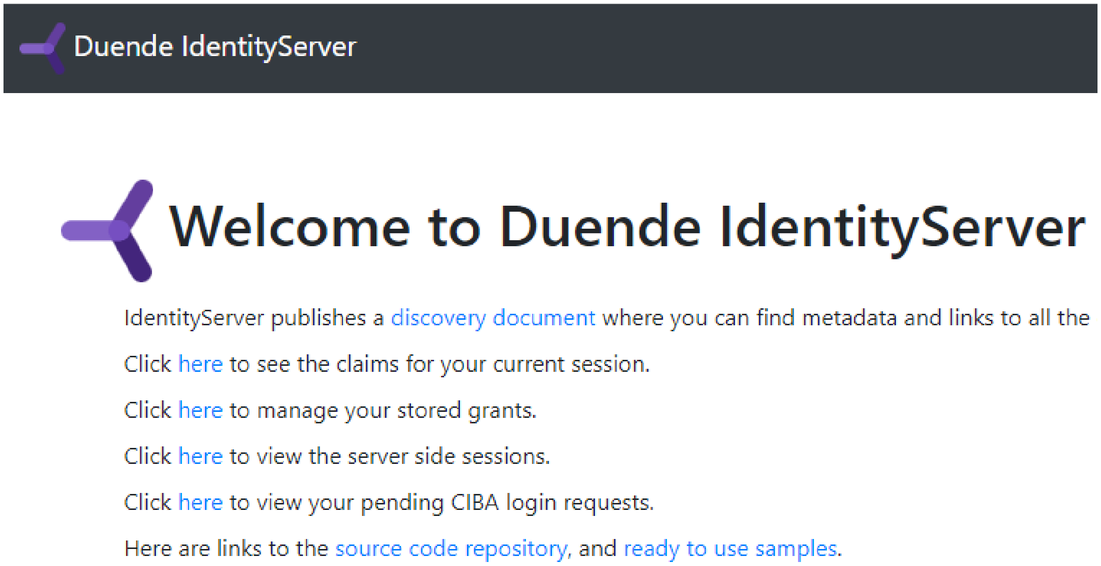
IdentityServer publishes a discovery document where you can find metadata and links to all the endpoints, key material, etc.

Click here to see the claims for your current session.

Click here to manage your stored grants.

Here are links to the source code repository, and ready to use samples.

For Duende, we are going to see a bit different screen with the same links:



Welcome to Duende IdentityServer

IdentityServer publishes a discovery document where you can find metadata and links to all the

Click here to see the claims for your current session.

Click here to manage your stored grants.

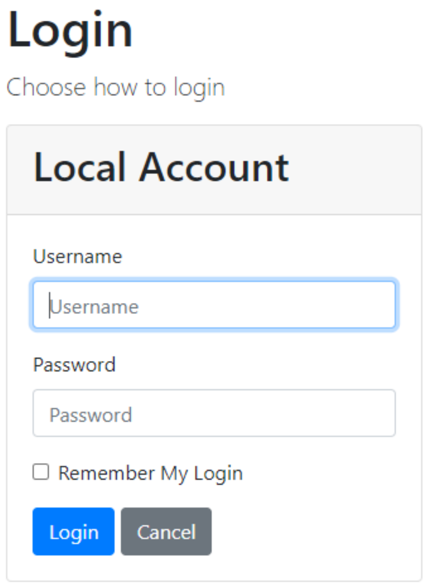
Click here to view the server side sessions.

Click here to view your pending CIBA login requests.

Here are links to the source code repository, and ready to use samples.

Well, this is a lot better now. In addition to this page, you can click these links to see the configuration page and the login page as well.

We've already seen the configuration, so let's click the first "here" link:



Login

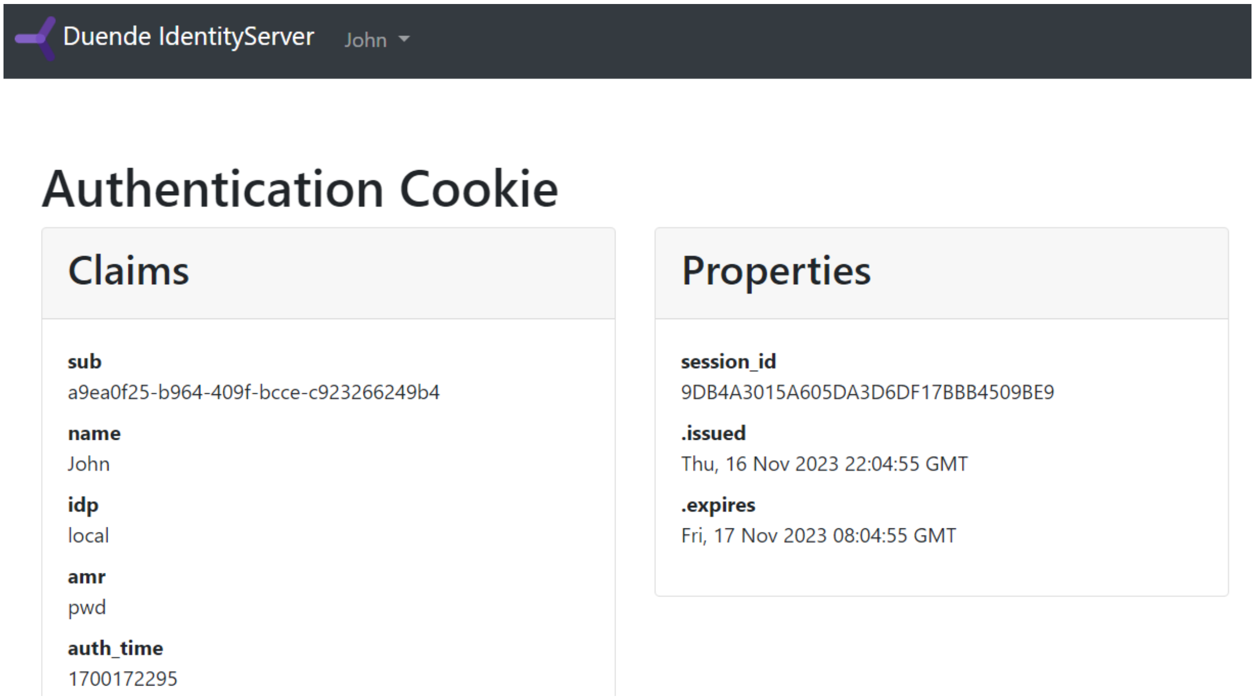
Choose how to login

Local Account

Username

Password

We can see the Login form. Once we enter the credentials from the `TestUsers` class, we will see the claims and properties:



Authentication Cookie

Claims

sub
a9ea0f25-b964-409f-bcce-c923266249b4

name
John

idp
local

amr
pwd

auth_time
1700172295

Properties

session_id
9DB4A3015A605DA3D6DF17BBB4509BE9

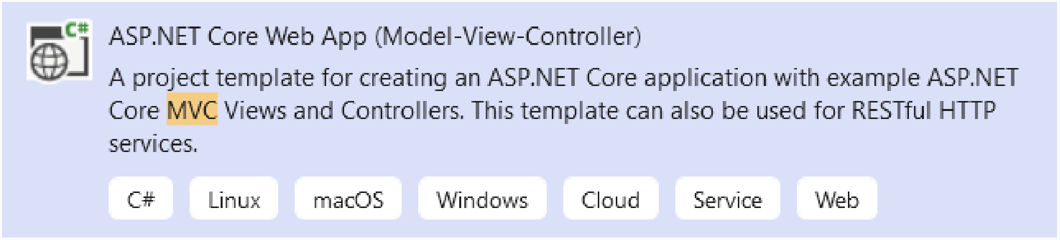
.issued
Thu, 16 Nov 2023 22:04:55 GMT

.expires
Fri, 17 Nov 2023 08:04:55 GMT

4 Creating a Client Application

In the first part of this book, we talked about different client types – confidential and public. Of course, we need a client application to consume our API. Therefore, we are going to create a new confidential client application.

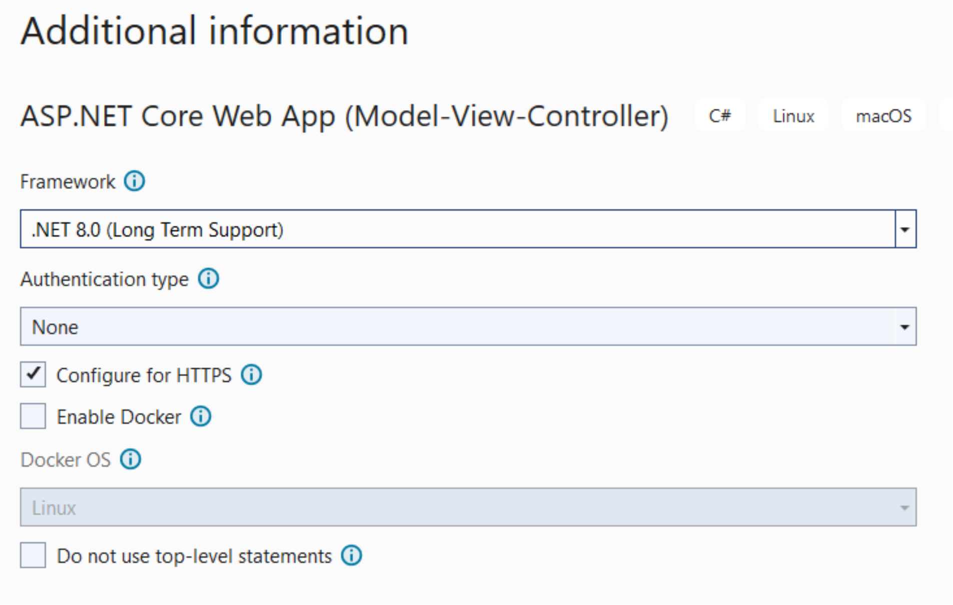
So, let's start by creating a new ASP.NET Core project:



ASP.NET Core Web App (Model-View-Controller)

A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

We will name it **CompanyEmployees.Client** and choose the .NET 8.0 version:



Additional information

ASP.NET Core Web App (Model-View-Controller)

Framework: .NET 8.0 (Long Term Support)

Authentication type: None

☑ Configure for HTTPS

☐ Enable Docker

Docker OS: Linux

☐ Do not use top-level statements

After we created a project, let's modify the **launchSettings.json** file:

```
"profiles": {
    "https": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5010",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
```

Now, we need to modify the **Program** class, to configure the HttpClient service:

```
builder.Services.AddHttpClient("APIClient", client =>
{
       client.BaseAddress = new Uri("https://localhost:5001/");
       client.DefaultRequestHeaders.Clear();
       client.DefaultRequestHeaders.Add(HeaderNames.Accept, "application/json");
});
```

Here, we register our HttpClient service and configure default values for the base address (our API address) and headers. For the HeaderNames class, we have to use the **Microsoft.Net.Http.Headers** namespace.

We need the ViewModel class, so, let's create it in the **Models** folder:

```
public class CompanyViewModel
{
    public string? Name { get; set; }
    public string? FullAddress { get; set; }
}
```

The next step is to modify the Home controller.

The first thing we are going to do is to inject the **IHttpClientFactory** interface since we need it to use our already registered **HttpClient** class:

```
private readonly IHttpClientFactory _httpClientFactory;

public HomeController(IHttpClientFactory httpClientFactory)
{
    _httpClientFactory = httpClientFactory;
}
```

Right after that, we are going to create a new action in the same controller class:

```
public async Task<IActionResult> Companies()
{
    var httpClient = _httpClientFactory.CreateClient("APIClient");

    var response = await httpClient.GetAsync("api/companies").ConfigureAwait(false);

    response.EnsureSuccessStatusCode();

    var companiesString = await response.Content.ReadAsStringAsync();
    var companies = JsonSerializer.Deserialize<List<CompanyViewModel>>(companiesString,
new JsonSerializerOptions { PropertyNameCaseInsensitive = true});

    return View(companies);
}
```

We use the **_httpClientFactory** object to create our API client and use that client with the **GetAsync** method to send a request to the API endpoint. Then, we read the content, convert it to a list and return a view.

Now, let's create that view:

```
@model IEnumerable<CompanyEmployees.Client.Models.CompanyViewModel>

@{
    ViewData["Title"] = "Companies";
}

<h1>Companies</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FullAddress)
            </th>
            <th>Actions</th>
        </tr>
    </thead>
    <tbody>
    @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FullAddress)
                </td>
                <td>
                    @Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) |
                    @Html.ActionLink("Details", "Details", new { /* id=item.PrimaryKey */
}) |
                    @Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */
})
                </td>
            </tr>
    }
    </tbody>
</table>
```

Excellent.

To finish the client creation, let's modify the `_Layout.cshtml` file, to include this view in the menu:

```
<div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Companies">Companies</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
        </li>
     </ul>
 </div>
```

As you can see, we just modify the Home link to the Companies link, nothing else.

4.1 Regarding the API

For the API, we are using the project from our main book's source code. You can find it in this book's source code folder called **04-ClientApplication/CompanyEmployees.API**. If you have read our main book Ultimate ASP.NET Core Web API, you probably have the database created. If you don't, all you have to do is modify the connection string in the **appsettings.json** file (or leave it as is):

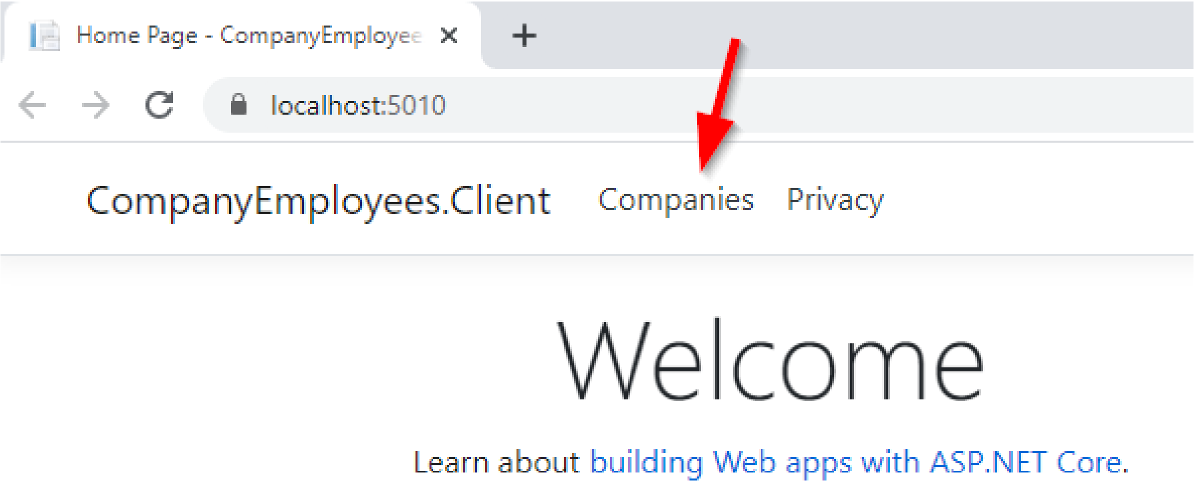
```
"ConnectionStrings": {
    "sqlConnection": "server=.; database=CompanyEmployee; Integrated Security=true;
TrustServerCertificate=true"
},
```

Then run the **Update-Database** command to execute migrations.

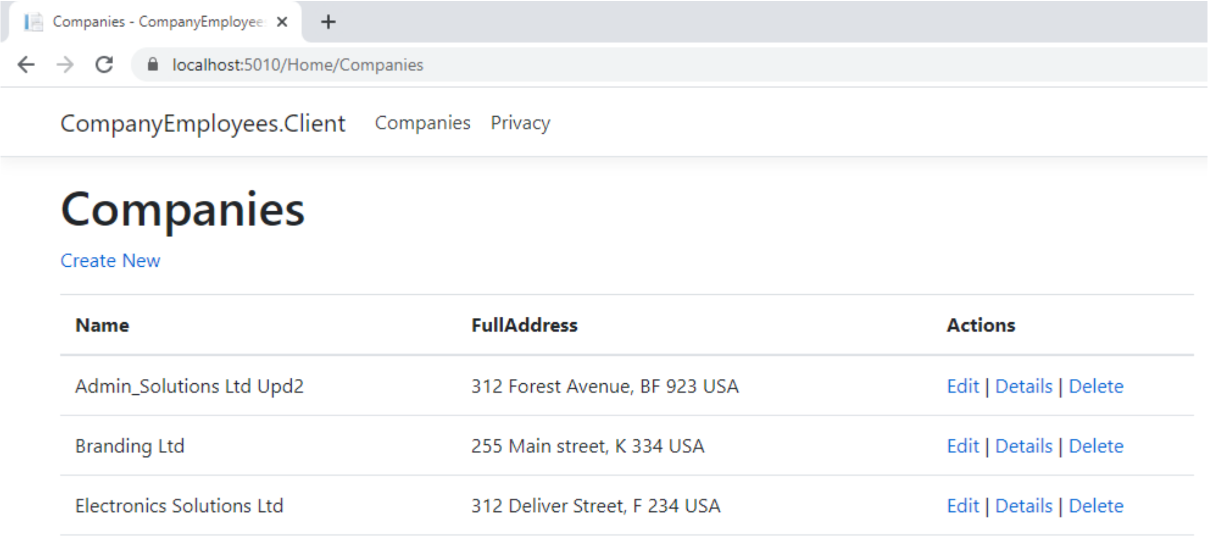
After that, you will have your database created and populated with initial data.

4.2 Testing the Client Application

After all of these changes, we can start our API and Client application. As soon as our client starts, we are going to see the Home screen:


Home Page - CompanyEmployee
localhost:5010
CompanyEmployees.Client
Companies
Privacy
Welcome
Learn about building Web apps with ASP.NET Core.

Now, if we click the Companies link, we should be able to see our companies data:



Companies

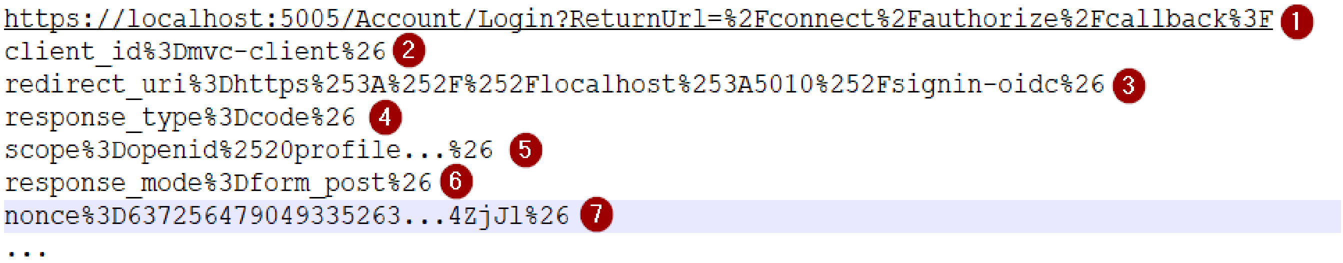
Create New

Name	FullAddress	Actions
Admin_Solutions Ltd Upd2	312 Forest Avenue, BF 923 USA	Edit | Details | Delete
Branding Ltd	255 Main street, K 334 USA	Edit | Details | Delete
Electronics Solutions Ltd	312 Deliver Street, F 234 USA	Edit | Details | Delete

That's it for now, regarding the client application. We have the API prepared, the IDP project ready, and the client application consuming our API. Now it's time to add the security logic for our applications.

5 Securing the Web Application

In the process of securing our application, we have to create a URI that will direct us to the Identity Provider project. That URI consists of multiple elements. Each of them contains important information for the security process. We have extracted and simplified (for readability) parts of such a URI:



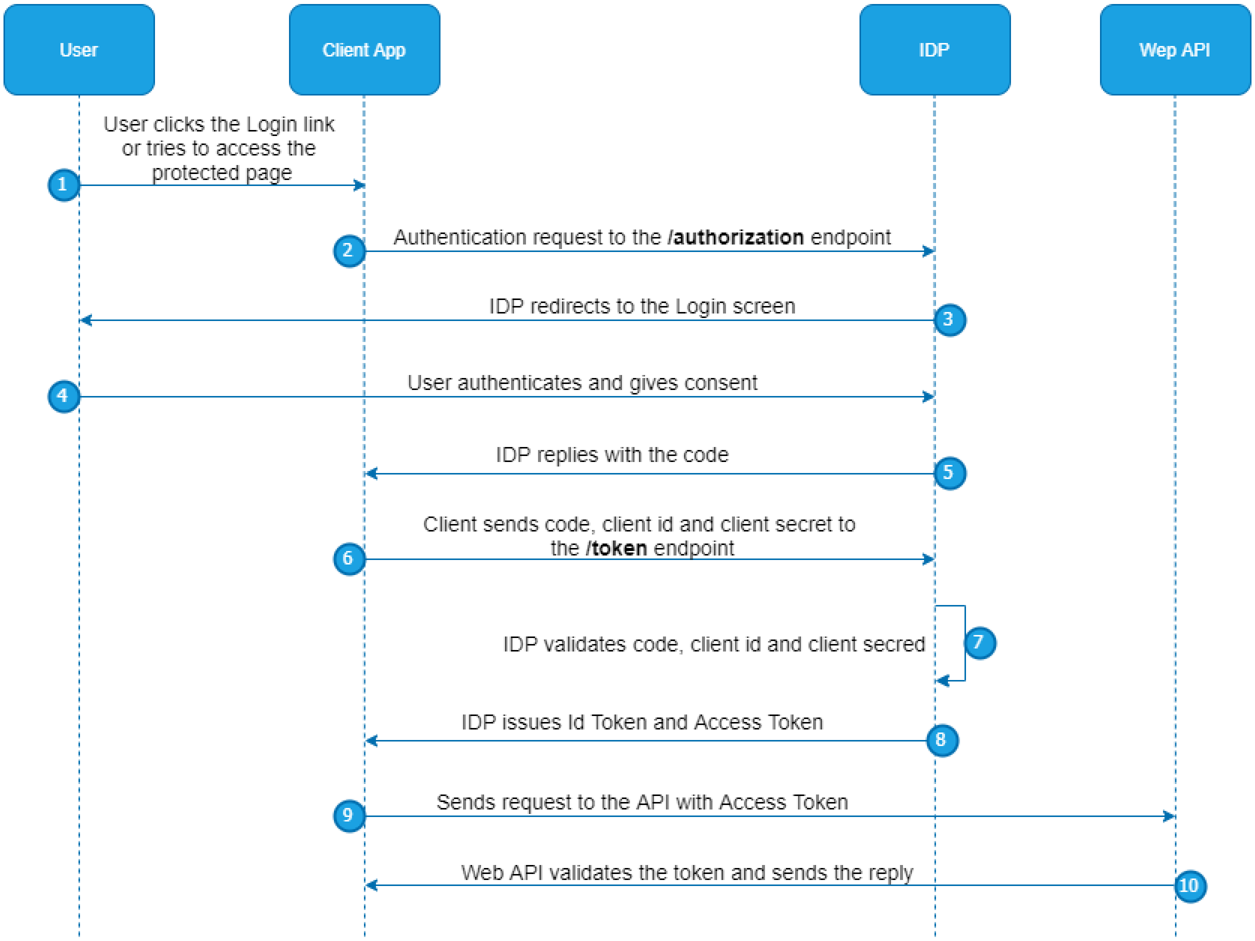
```
https://localhost:5005/Account/Login?ReturnUrl=%2Fconnect%2Fauthorize%2Fcallback%3F ❶
client_id%3Dmvc-client%26 ❷
redirect_uri%3Dhttps%253A%252F%252Flocalhost%253A5010%252Fsignin-oidc%26 ❸
response_type%3Dcode%26 ❹
scope%3Dopenid%2520profile...%26 ❺
response_mode%3Dform_post%26 ❻
nonce%3D637256479049335263...4ZjJl%26 ❼
...
```

So, let’s find out what each of these parts means:

1. The first part is the **/authorization** endpoint URI at the level of the IDP.
2. Next, we have a client_id that is an identifier of our client MVC application.
3. Then, there is the redirection URI that points to our client application. This URI is required for our IDP project so that we know where to deliver the code.
4. The response type states which flow we are using for the security actions. We are going to use the Authorization Code Flow and the **code** response type suggests just that.
5. We have the scopes as well. The application requires access to the OpenId and Profile scopes. If you remember, we have enabled them at the IDP level.

6. With the help of the response_mode part, we decide in what way we want to deliver the information to the browser (URI or Form POST)
7. The nonce part is here for validation purposes. IdentityServer will include this nonce in the identity token while sending it to the client.

Now, we can inspect the diagram, to see how the authorization code flow works:


User
Client App
IDP
Wep API
1 User clicks the Login link or tries to access the protected page
2 Authentication request to the /authorization endpoint
3 IDP redirects to the Login screen
4 User authenticates and gives consent
5 IDP replies with the code
6 Client sends code, client id and client secret to the /token endpoint
7 IDP validates code, client id and client secred
8 IDP issues Id Token and Access Token
9 Sends request to the API with Access Token
10 Web API validates the token and sends the reply

As we can see:

1. As soon as a user tries to login or access the protected page
2. The client sends an authentication request to the **/authorization** endpoint with the response_type code and other parameters we saw in the URI

3. The IDP shows the Login page to the user
4. As soon as the user provides credentials and gives consent
5. The IDP replies with the code via URI redirection or the Form POST. This is the front channel of communication.
6. The client then calls the **/token** endpoint through the back channel by providing the code, client id and client secret.
7. After IDP validates the code and the client credentials
8. It issues the id token and the access token (it can issue the refresh token as well if requested)
9. The client application then uses that access token to attach it to the HTTP request, usually as a Bearer token. It needs the access token for the verification process against the Web API.
10. Finally, the Web API replies after it successfully validates the token.

5.1 Configuring the IdentityServer to Use the Authorization Code Flow

Now it's time to configure the IDP project to support the use of the authorization code flow. To do that, we have to modify the **Config** class:

```
public static IEnumerable<Client> Clients =>
    new Client[]
    {
        new Client
        {
            ClientName = "CompanyEmployeeClient",
            ClientId = "companyemployeeclient",
            AllowedGrantTypes = GrantTypes.Code,
            RedirectUris = new List<string>{ "https://localhost:5010/signin-oidc" },
            AllowedScopes = { IdentityServerConstants.StandardScopes.OpenId,
IdentityServerConstants.StandardScopes.Profile },
            ClientSecrets = { new Secret("CompanyEmployeeClientSecret".Sha512()) },
            RequirePkce = false,
            RequireConsent = true
        }
    };
```

We register our client with a name and an id (**ClientName** and **ClientId** properties). With the **AllowedGrantTypes** property, we define a flow we want to use. Here, we are using the `Code` that stands for the authorization code flow. This flow is redirection-based (tokens are delivered to the browser in the URI via redirection) and therefore, we have to populate the `RedirectUris` property. This address is our client application's address, with the addition of `/signin-oidc`. Then, we add allowed scopes, and these scopes are already supported in the `GetIdentityResource` method. Finally, we add a secret that is hashed with the `Sha512` algorithm and for now, disable the PKCE protection.

With this configuration, we have enabled the Consent screen to better understand the process happening behind the scene. But if you don't want to give consent every time, you can just remove the `RequireConsent` property since by default it is set to false.

This is all we require at the IDP level.

5.2 Securing the Client Application with the Authorization Code Flow

In the client application, we need to provide a way to store the user's identity. For that, we are going to modify the `Program` class, right below the HttpClient configuration:

```
builder.Services.AddAuthentication(opt =>
{
       opt.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
       opt.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
}).AddCookie(CookieAuthenticationDefaults.AuthenticationScheme);
```

Here, we register authentication as a service and populate the `DefaultScheme` and `DefaultChallengeScheme` properties. Finally, we call the `AddCookie` method with the name of the scheme, to register the cookie handler and the cookie-based authentication for our default scheme. Once the identity token has been validated and transformed into a claims identity, it will be stored in a cookie, which then can be used for each request to the web application.

For the **CookieAuthenticationDefaults** class, we have to include **Microsoft.AspNetCore.Authentication.Cookies** namespace. And for the **OpenIdConnectDefaults** class, we have to install the **Microsoft.AspNetCore.Authentication.OpenIdConnect** package.

Next, we have to add the OpenID Connect support:

```
builder.Services.AddAuthentication(opt =>
{
    opt.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    opt.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
}).AddCookie(CookieAuthenticationDefaults.AuthenticationScheme)
  .AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, opt =>
  {
      opt.SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
      opt.Authority = "https://localhost:5005";
      opt.ClientId = "companyemployeeclient";
      opt.ResponseType = OpenIdConnectResponseType.Code;
      opt.SaveTokens = true;
      opt.ClientSecret = "CompanyEmployeeClientSecret";
      opt.UsePkce = false;
  });
```

The first parameter in the **AddOpenIdConnect** cookie is the same as the value for the **DefaultChallengeScheme**, which means that the OpenID Connect will be set as default for the authentication actions.
The **SignInScheme** property has the same value as our **DefaultScheme**.
The **Authority** property has the value of our IdentityServer address.
The **ClientId** and **ClientSecret** properties have to be the same as the id and secret from the **Config** class for this client. We set the **SaveTokens** property to true to store the token after successful authorization. Now, let’s look at the **ResponseType** property. We set it to a code value and therefore, we expect the code to be returned from the **/authorization** endpoint. The **OpenIdConnectResponseType** class

lives in the **Microsoft.IdentityModel.Protocols.OpenIdConnect** namespace.

The Authorization Code flow requires the **UsePkce** property to be enabled, but for now, we are going to disable it (we'll enable it later on). Also, in the **Config** class, we have defined the **OpenId** and the **Profile** scopes as the **AllowedScopes** (Client configuration part), but here, we don't specify that. That's because these two scopes are requested by default.

Now, we have to navigate to enable authentication:

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

The last thing we have to do is to ensure that an unauthenticated user can't access the Companies action. We are going to do that by applying the Authorize attribute just before that action:

```
[Authorize]
public async Task<IActionResult> Companies()
{
    ...
}
```

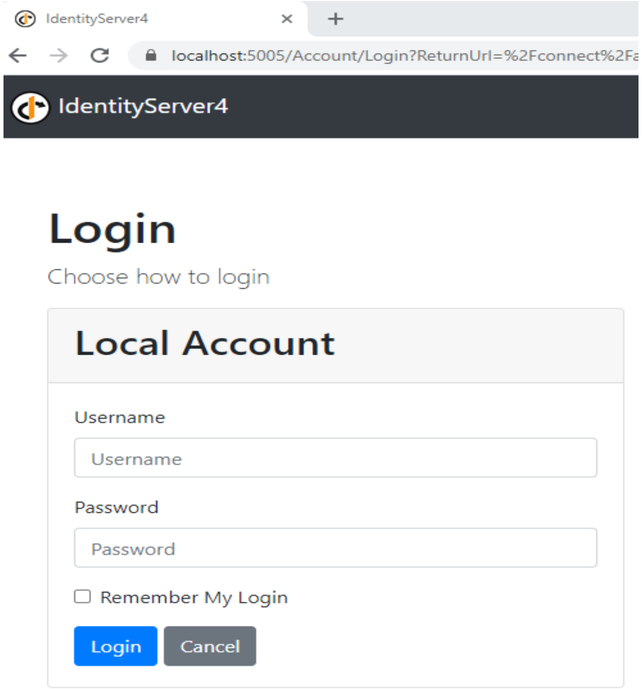
For this attribute to be available, we have to include additional namespace:

```
using Microsoft.AspNetCore.Authorization;
```

Now we can test this.

5.3 Testing the Functionality

Right now, we can start the API, IDP, and Client application. Once the Home screen is shown, we can click the Companies link:



IdentityServer4

localhost:5005/Account/Login?ReturnUrl=%2Fconnect%2Fa

IdentityServer4

Login

Choose how to login

Local Account

Username

Username

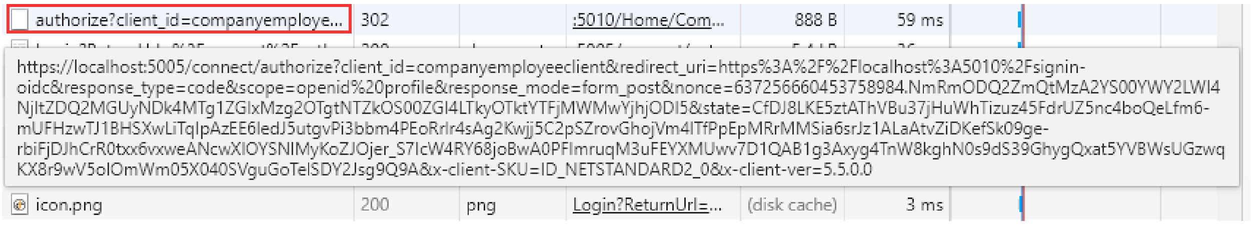
Password

Password

□ Remember My Login

Login Cancel

We get redirected to the Login screen on the IDP level. If we inspect the URI:

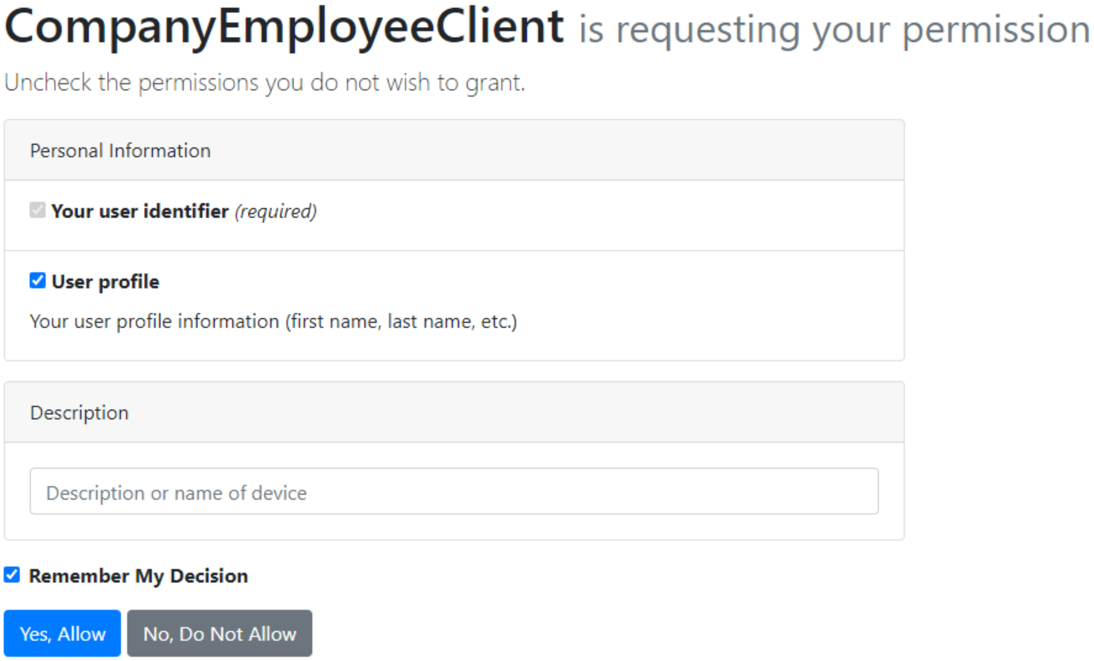


```
authorize?client_id=companyemploye...   302         :5010/Home/Com...    888 B       59 ms
https://localhost:5005/connect/authorize?client_id=companyemployeeclient&redirect_uri=https%3A%2F%2Flocalhost%3A5010%2Fsignin-
oidc&response_type=code&scope=openid%20profile&response_mode=form_post&nonce=637256660453758984.NmRmODQ2ZmQtMzA2YS00YWY2LWI4
NjltZDQ2MGUyNDk4MTg1ZGIxMzg2OTgtNTZkOS00ZGI4LTkyOTktYTFjMWMwYjhjODI5&state=CfDJ8LKE5ztAThVBu37jHuWhTizuz45FdrUZ5nc4boQeLfm6-
mUFHzwTJ1BHSXwLiTqlpAzEE6ledJ5utgvPi3bbm4PEoRrlr4sAg2Kwjj5C2pSZrovGhojVm4lTfPpEpMRrMMSia6srJz1ALaAtvZiDKefSk09ge-
rbiFjDJhCrR0txx6vxweANcwXlOYSNlMyKoZJOjer_S7IcW4RY68joBwA0PFlmruqM3uFEYXMUwv7D1QAB1g3Axyg4TnW8kghN0s9dS39GhygQxat5YVBWsUGzwq
KX8r9wV5olOmWm05X040SVguGoTelSDY2Jsg9Q9A&x-client-SKU=ID_NETSTANDARD2_0&x-client-ver=5.5.0.0
icon.png                                200    png  Login?ReturnUrl=...  (disk cache)  3 ms
```

We can see all we talked about at the beginning of this chapter.

You can also inspect the console logs, to find additional information.

Once we enter valid credentials (John – JohnPassword), we are going to see the consent screen:



CompanyEmployeeClient is requesting your permission

Uncheck the permissions you do not wish to grant.

Personal Information

☑ **Your user identifier** *(required)*

☑ **User profile**

Your user profile information (first name, last name, etc.)

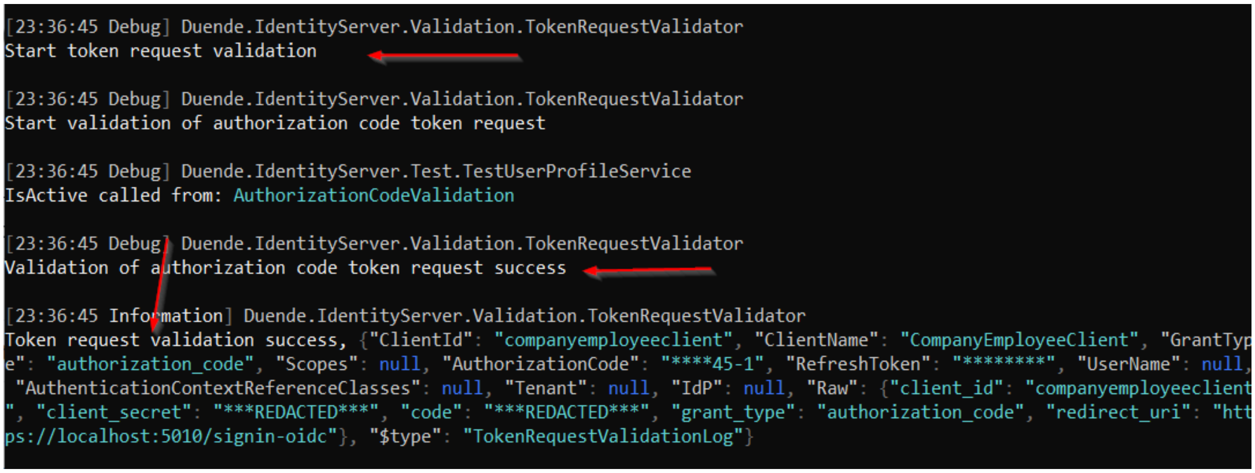
Description

Description or name of device

☑ **Remember My Decision**

We can see that the Client application requests our permission for the data we specified in the allowed scopes in the configuration (OpenId and Profile). When we click the Yes button, we are going to see our requested data.

Additionally, if we inspect the console log:



```
[23:36:45 Debug] Duende.IdentityServer.Validation.TokenRequestValidator
Start token request validation

[23:36:45 Debug] Duende.IdentityServer.Validation.TokenRequestValidator
Start validation of authorization code token request

[23:36:45 Debug] Duende.IdentityServer.Test.TestUserProfileService
IsActive called from: AuthorizationCodeValidation

[23:36:45 Debug] Duende.IdentityServer.Validation.TokenRequestValidator
Validation of authorization code token request success

[23:36:45 Information] Duende.IdentityServer.Validation.TokenRequestValidator
Token request validation success, {"ClientId": "companyemployeeclient", "ClientName": "CompanyEmployeeClient", "GrantTyp
e": "authorization_code", "Scopes": null, "AuthorizationCode": "****45-1", "RefreshToken": "********", "UserName": null,
 "AuthenticationContextReferenceClasses": null, "Tenant": null, "IdP": null, "Raw": {"client_id": "companyemployeeclient
", "client_secret": "***REDACTED***", "code": "***REDACTED***", "grant_type": "authorization_code", "redirect_uri": "htt
ps://localhost:5010/signin-oidc"}, "$type": "TokenRequestValidationLog"}
```

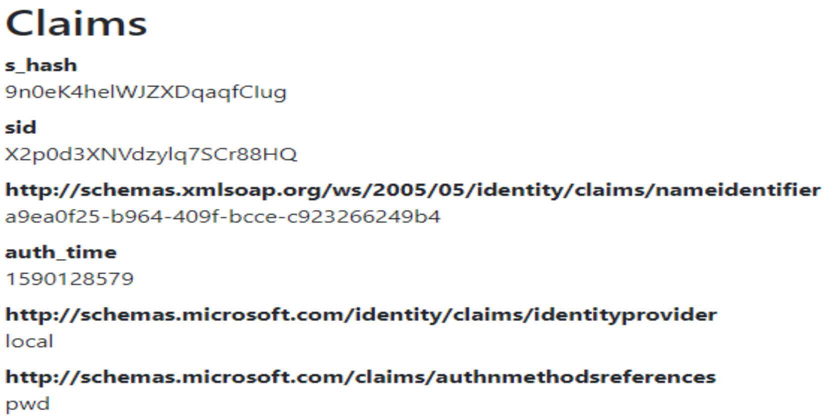
We can see the validation process of the authorization code. So, the token was sent via the back channel to the `/token` endpoint and the validation was successful.

5.4 Inspecting Claims

To inspect the claims from the token, we are going to modify the Privacy view file in the client application:

```
@using Microsoft.AspNetCore.Authentication

<h2>Claims</h2>

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

<h2>Properties</h2>

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

Now, we can start the client application and navigate to the Companies page. We have to log in. After that, let's click the Privacy link:



Claims

s_hash
9n0eK4helWJZXDqaqfClug

sid
X2p0d3XNVdzylq7SCr88HQ

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier
a9ea0f25-b964-409f-bcce-c923266249b4

auth_time
1590128579

http://schemas.microsoft.com/identity/claims/identityprovider
local

http://schemas.microsoft.com/claims/authnmethodsreferences
pwd

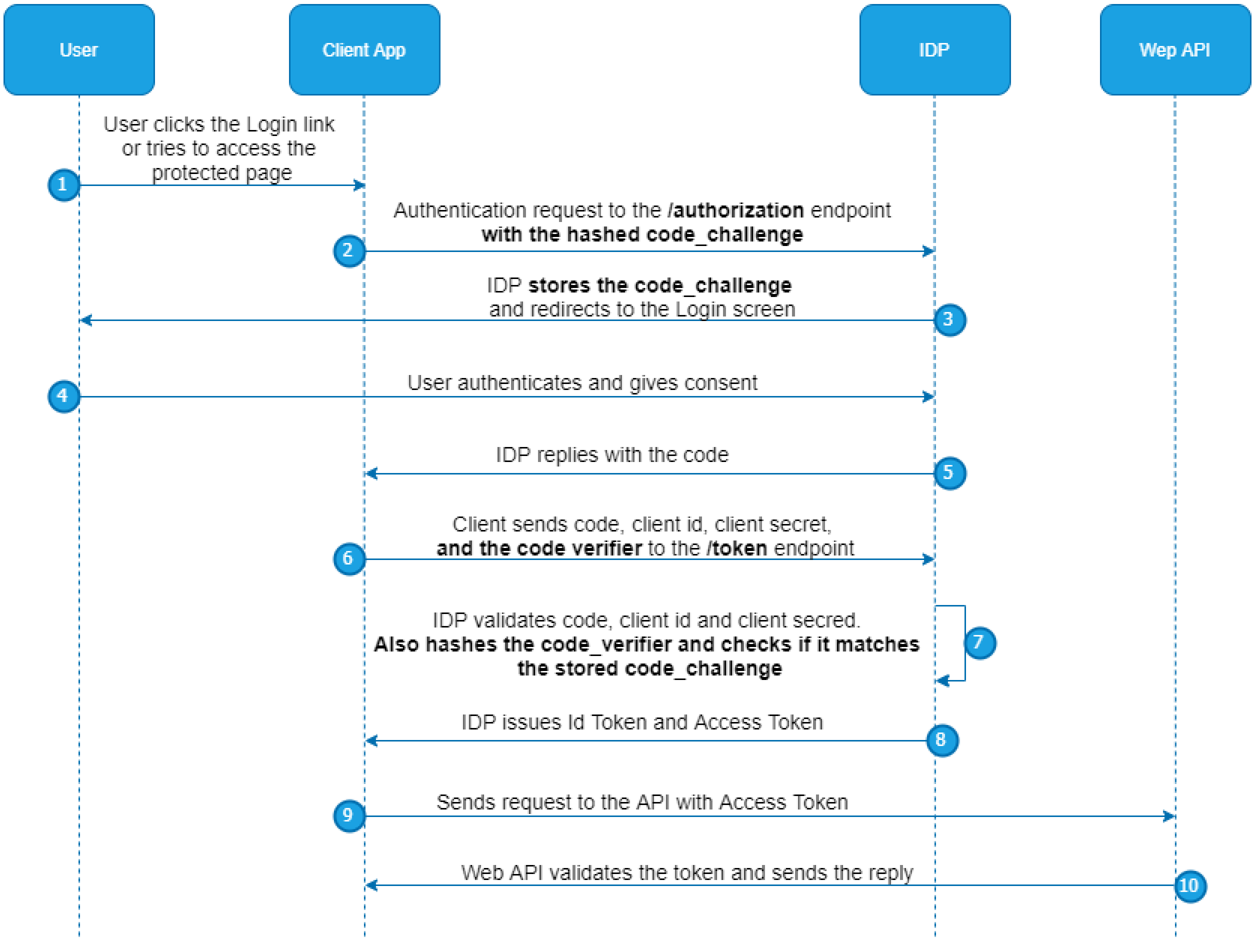
Here, we can see our claims. Bellow them, we can find the Properties as well.

5.5 Using PKCE (Proof Key for Code Exchange)

Right now, there is a problem with this Authorization Code Flow. The code is vulnerable to a code injection attack. The attacker can access that code and use it to switch sessions with the victim. This means the attacker will have all the privileges as the victim.

The recommended way of solving this problem is by using PKCE. With PKCE configured, with each request to the /authorization endpoint, the secret is created by the client. Then, when calling the /token endpoint, this secret is verified and IDP will return a token only if the secret matches.

So, let's see how the diagram looks now, with the PKCE protection:


User
Client App
IDP
Wep API
1 User clicks the Login link or tries to access the protected page
2 Authentication request to the **/authorization** endpoint **with the hashed code_challenge**
3 IDP **stores the code_challenge** and redirects to the Login screen
4 User authenticates and gives consent
5 IDP replies with the code
6 Client sends code, client id, client secret, **and the code verifier** to the **/token** endpoint
7 IDP validates code, client id and client secred. **Also hashes the code_verifier and checks if it matches the stored code_challenge**
8 IDP issues Id Token and Access Token
9 Sends request to the API with Access Token
10 Web API validates the token and sends the reply

So, when a client sends a request to the /authorization endpoint, it adds the hashed code_challenge. This code is stored at the IDP level. Later on, the client sends the code_verifier, next to the client's credentials and code. IDP hashes the code_verifier and compares it to the stored code_challenge. If it matches, IDP replies with the id token and access token.

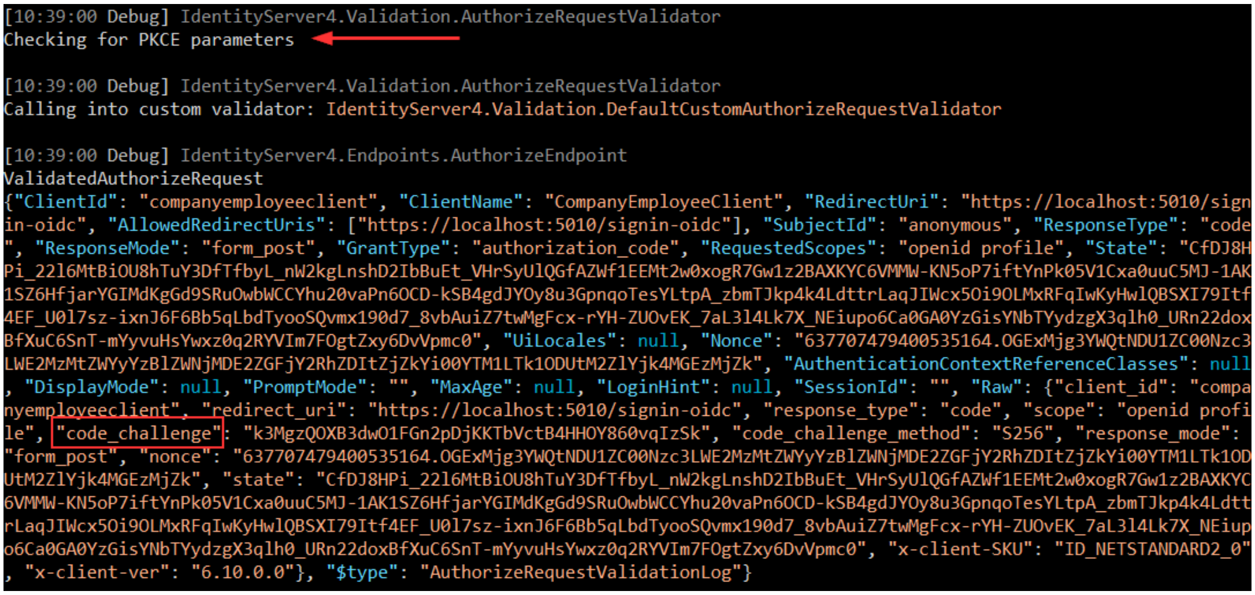
Now, when we understand the flow with PKCE, we can continue with the implementation.

The first thing we have to do is to modify the client configuration in the Config class:

```
new Client
{
    ClientName = "CompanyEmployeeClient",
    ClientId = "companyemployeeclient",
    AllowedGrantTypes = GrantTypes.Code,
    RedirectUris = new List<string>{ "https://localhost:5010/signin-oidc" },
    AllowedScopes = { IdentityServerConstants.StandardScopes.OpenId,
IdentityServerConstants.StandardScopes.Profile },
    ClientSecrets = { new Secret("CompanyEmployeeClientSecret".Sha512()) },
    RequirePkce = true,
    RequireConsent = true
}
```

The next thing we have to do is to remove the **opt.UsePkce = false** code from the OpenIdConnect configuration in the client application. The default value for the UsePkce property is true.

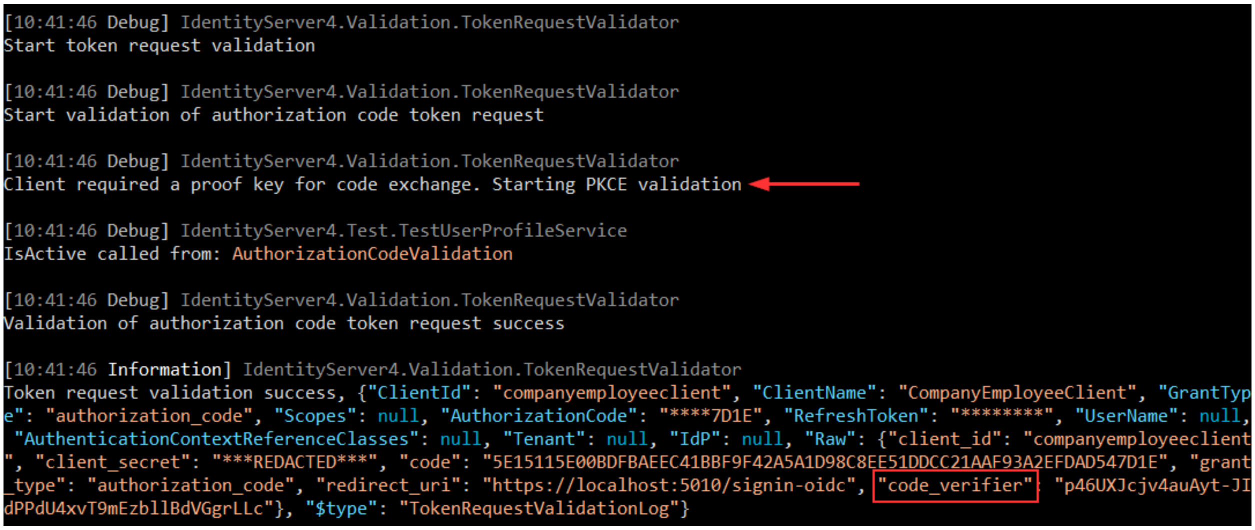
Let’s start our applications and test this by clicking the Companies link:



```
[10:39:00 Debug] IdentityServer4.Validation.AuthorizeRequestValidator
Checking for PKCE parameters

[10:39:00 Debug] IdentityServer4.Validation.AuthorizeRequestValidator
Calling into custom validator: IdentityServer4.Validation.DefaultCustomAuthorizeRequestValidator

[10:39:00 Debug] IdentityServer4.Endpoints.AuthorizeEndpoint
ValidatedAuthorizeRequest
{"ClientId": "companyemployeeclient", "ClientName": "CompanyEmployeeClient", "RedirectUri": "https://localhost:5010/sign
in-oidc", "AllowedRedirectUris": ["https://localhost:5010/signin-oidc"], "SubjectId": "anonymous", "ResponseType": "code
", "ResponseMode": "form_post", "GrantType": "authorization_code", "RequestedScopes": "openid profile", "State": "CfDJ8H
Pi_2216MtBiOU8hTuY3DfTfbyL_nW2kgLnshD2IbBuEt_VHrSyUlQGfAZWf1EEMt2w0xogR7Gw1z2BAXKYC6VMMW-KN5oP7iftYnPk05V1Cxa0uuC5MJ-1AK
1SZ6HfjarYGIMdKgGd9SRuOwbWCCYhu20vaPn6OCD-kSB4gdJYOy8u3GpnqoTesYLtpA_zbmTJkp4k4LdttrLaqJIWcx5Oi9OLMxRFqIwKyHwlQBSXI79Itf
4EF_U0l7sz-ixnJ6F6Bb5qLbdTyooSQvmx190d7_8vbAuiZ7twMgFcx-rYH-ZUOvEK_7aL3l4Lk7X_NEiupo6Ca0GA0YzGisYNbTYydzgX3qlh0_URn22dox
BfXuC6SnT-mYyvuHsYwxz0q2RYVIm7FOgtZxy6DvVpmc0", "UiLocales": null, "Nonce": "637707479400535164.OGExMjg3YWQtNDU1ZC00Nzc3
LWE2MzMtZWYyYzBlZWNjMDE2ZGFjY2RhZDItZjZkYi00YTM1LTk1ODUtM2ZlYjk4MGEzMjZk", "AuthenticationContextReferenceClasses": null
, "DisplayMode": null, "PromptMode": "", "MaxAge": null, "LoginHint": null, "SessionId": "", "Raw": {"client_id": "compa
nyemployeeclient", "redirect_uri": "https://localhost:5010/signin-oidc", "response_type": "code", "scope": "openid profi
le", "code_challenge": "k3MgzQOXB3dwO1FGn2pDjKKTbVctB4HHOY860vqIzSk", "code_challenge_method": "S256", "response_mode":
"form_post", "nonce": "637707479400535164.OGExMjg3YWQtNDU1ZC00Nzc3LWE2MzMtZWYyYzBlZWNjMDE2ZGFjY2RhZDItZjZkYi00YTM1LTk1OD
UtM2ZlYjk4MGEzMjZk", "state": "CfDJ8HPi_2216MtBiOU8hTuY3DfTfbyL_nW2kgLnshD2IbBuEt_VHrSyUlQGfAZWf1EEMt2w0xogR7Gw1z2BAXKYC
6VMMW-KN5oP7iftYnPk05V1Cxa0uuC5MJ-1AK1SZ6HfjarYGIMdKgGd9SRuOwbWCCYhu20vaPn6OCD-kSB4gdJYOy8u3GpnqoTesYLtpA_zbmTJkp4k4Ldtt
rLaqJIWcx5Oi9OLMxRFqIwKyHwlQBSXI79Itf4EF_U0l7sz-ixnJ6F6Bb5qLbdTyooSQvmx190d7_8vbAuiZ7twMgFcx-rYH-ZUOvEK_7aL3l4Lk7X_NEiup
o6Ca0GA0YzGisYNbTYydzgX3qlh0_URn22doxBfXuC6SnT-mYyvuHsYwxz0q2RYVIm7FOgtZxy6DvVpmc0", "x-client-SKU": "ID_NETSTANDARD2_0"
, "x-client-ver": "6.10.0.0"}, "$type": "AuthorizeRequestValidationLog"}
```

We can see the check for PKCE parameters and, in the request, we can find the code_challenge.

After we log in, we can inspect the logs once again:



```
[10:41:46 Debug] IdentityServer4.Validation.TokenRequestValidator
Start token request validation

[10:41:46 Debug] IdentityServer4.Validation.TokenRequestValidator
Start validation of authorization code token request

[10:41:46 Debug] IdentityServer4.Validation.TokenRequestValidator
Client required a proof key for code exchange. Starting PKCE validation

[10:41:46 Debug] IdentityServer4.Test.TestUserProfileService
IsActive called from: AuthorizationCodeValidation

[10:41:46 Debug] IdentityServer4.Validation.TokenRequestValidator
Validation of authorization code token request success

[10:41:46 Information] IdentityServer4.Validation.TokenRequestValidator
Token request validation success, {"ClientId": "companyemployeeclient", "ClientName": "CompanyEmployeeClient", "GrantTyp
e": "authorization_code", "Scopes": null, "AuthorizationCode": "****7D1E", "RefreshToken": "********", "UserName": null,
 "AuthenticationContextReferenceClasses": null, "Tenant": null, "IdP": null, "Raw": {"client_id": "companyemployeeclient
", "client_secret": "***REDACTED***", "code": "5E15115E00BDFBAEEC41BBF9F42A5A1D98C8EE51DDCC21AAF93A2EFDAD547D1E", "grant
_type": "authorization_code", "redirect_uri": "https://localhost:5010/signin-oidc", "code_verifier": "p46UXJcjv4auAyt-JI
dPPdU4xvT9mEzbllBdVGgrLLc"}, "$type": "TokenRequestValidationLog"}
```

The PKCE validation is here, we can see the code_verifier and that the validation process succeeded.

5.6 Login, Logout, and Additional User Info

Right now, we can navigate to the Login view only if we try to access a protected page. But, we want to be able to click the Login link and navigate to the Login page as well. To do that, let's create a new Auth controller in the client application and add a Login action:

```
public class AuthController : ControllerBase
{
    public IActionResult Login()
    {
        return Challenge(new AuthenticationProperties
        {
            RedirectUri = "/"
        });
    }
}
```

The **AuthenticationProperties** class lives inside the **Microsoft.AspNetCore.Authentication** namespace.

Then, let's modify the **_Layout** file:

```
<div class="navbar-collapse collapse">
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Companies">Companies</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
        </li>
    </ul>
    @if (!User.Identity.IsAuthenticated)
    {
        <section>
            <a class="nav-link text-dark" asp-area="" asp-controller="Auth" asp-
action="Login">Login</a>
        </section>
    }
    else
    {
        <section>
            <a class="nav-link text-dark" asp-area="" asp-controller="Auth" asp-
action="Logout">Logout</a>
        </section>
    }
```

```
</div>
```

Here, we create a login link if the user is not authenticated. Otherwise, we show the Logout link.

If we start the client application and we are not logged in, we are going to see the Login link. Once we click it, we are going to be redirected to the login page. After a successful login, the Logout link will be available. But, we don't have the Logout action, so let's add it to the Auth controller:

```
public async Task Logout()
{
    await HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
    await HttpContext.SignOutAsync(OpenIdConnectDefaults.AuthenticationScheme);
}
```

Let's make a few more changes to the IDP application.

First, let's add one more property in the Config class for the Client configuration:

```
RequirePkce = true,
PostLogoutRedirectUris = new List<string> { "https://localhost:5010/signout-callback-oidc" }
```

Next, let's open the **`AccountOptions`** class (QuickStart/Account folder), for Duende it is **`LogOutOptions`** class (Pages/Account/Logout folder) and set the **`AutomaticRedirectAfterSignOut`** property to true:

```
public static bool AutomaticRedirectAfterSignOut = true;
```

With these settings in place, after successful logout action, our application will redirect the user to the Home page. You can try it for yourself.

But now, we still have one more problem. On the Login screen, if we click the Cancel button, we will get an error page.

There are multiple ways to solve this, but the easiest one is to modify the Login action in the Account controller at the IDP level:

```
public async Task<IActionResult> Login(LoginInputModel model, string button)
{
    ...
    if (button != "login")
    {
        if (context != null)
        {
            await _interaction.DenyAuthorizationAsync(context,
AuthorizationError.AccessDenied);

            if (context.IsNativeClient())
            {
                return this.LoadingPage("Redirect", model.ReturnUrl);
            }

            return Redirect(context.Client.ClientUri);
        }

...
```

Fot the Duende project we have to modify the Index.cshtml.cs file inside the Pages\Account\Login folder:

```
public async Task<IActionResult> OnPost()
{
    // check if we are in the context of an authorization request
    var context = await _interaction.GetAuthorizationContextAsync(Input.ReturnUrl);

    if (Input.Button != "login")
    {
        if (context != null)
        {
            await _interaction.DenyAuthorizationAsync(context,
AuthorizationError.AccessDenied);

            if (context.IsNativeClient())
            {
                return this.LoadingPage(Input.ReturnUrl);
            }

            return Redirect(context.Client.ClientUri);
        }

...
```

And add the `ClientUri` property in the Client configuration:

```
new Client
{
        ...
        RequireConsent = true,
        ClientUri = "https://localhost:5010"
}
```

As soon as we click the Cancel button, we are going to be redirected to the Home page.

5.7 Additional Claims

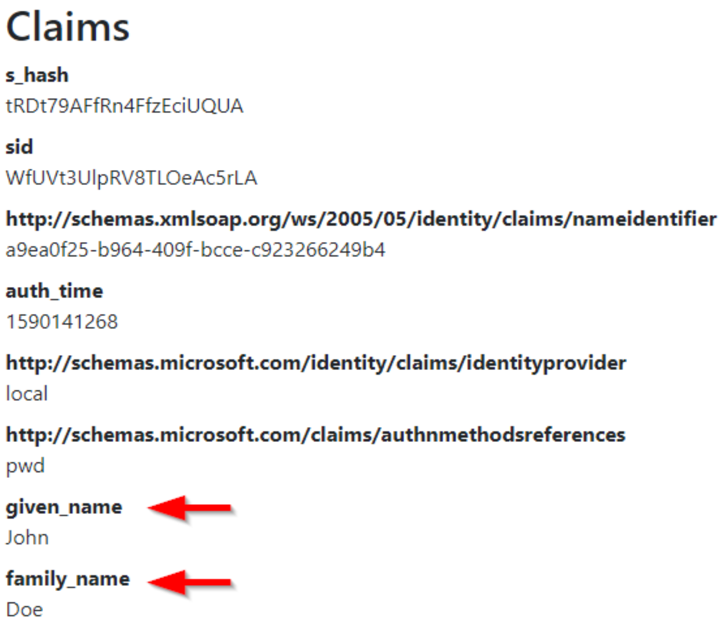
If we look again at the Consent screen picture, we are going to see that we allowed MVC Client to use the id and the user profile information. But, if we inspect the content on the Privacy page, we are going to see we are missing the given_name and the family_name claims – from the Profile scope.

We can include these claims in the id token but, with too much information in the id token, it can become quite large and cause issues due to URI length restrictions. So, we are going to get these claims another way, by modifying the **OpenID Connect** configuration:

```
.AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, opt =>
{
    opt.SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    opt.Authority = "https://localhost:5005";
    opt.ClientId = "companyemployeeclient";
    opt.ResponseType = OpenIdConnectResponseType.Code;
    opt.SaveTokens = true;
    opt.ClientSecret = "CompanyEmployeeClientSecret";
    opt.GetClaimsFromUserInfoEndpoint = true;
});
```

With this modification, we allow our middleware to communicate with the **/userinfo** endpoint to retrieve additional user data.

At this point, we can log in again and inspect the Privacy page:



Claims

s_hash
tRDt79AFfRn4FfzEciUQUA

sid
WfUVt3UlpRV8TLOeAc5rLA

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier
a9ea0f25-b964-409f-bcce-c923266249b4

auth_time
1590141268

http://schemas.microsoft.com/identity/claims/identityprovider
local

http://schemas.microsoft.com/claims/authnmethodsreferences
pwd

given_name
John

family_name
Doe

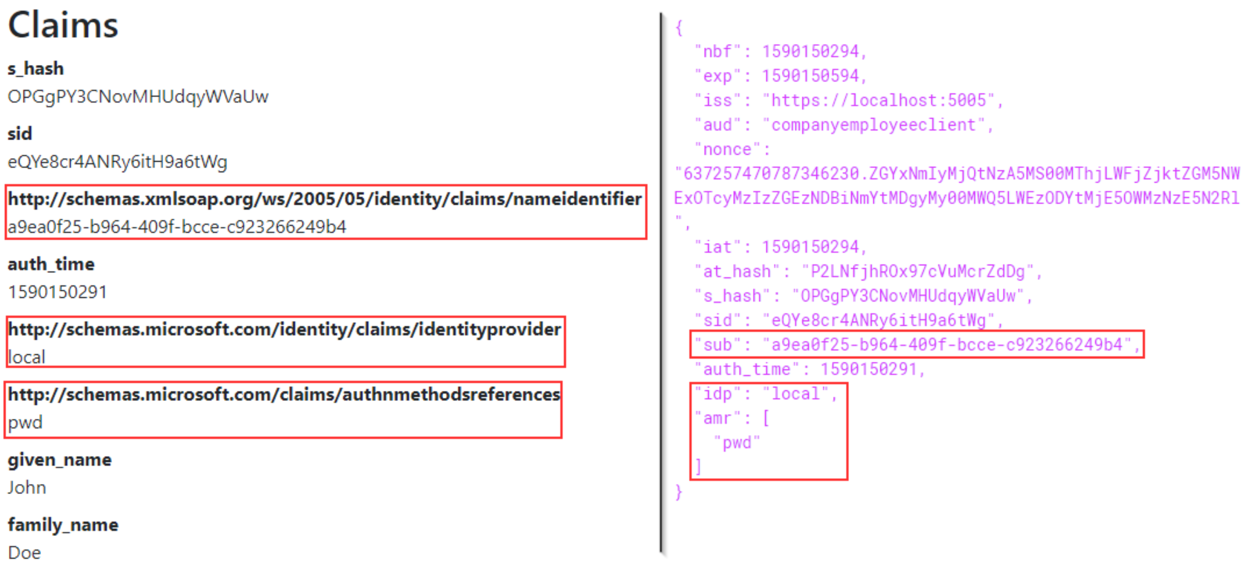
Excellent. We can see our additional claims.

6 Working with Claims and Authorization

We can use claims to show identity-related information in our application, but we can use them for the authorization process as well. In this section, we are going to learn how to modify our claims and add new ones. We are also going to learn about the Authorization process and how to use Roles to protect our endpoints.

6.1 Modifying Claims

If we inspect our decoded **id_token** with the claims on the Privacy page, we are going to find some naming differences:



Claims

s_hash
OPGgPY3CNovMHUdqyWVaUw

sid
eQYe8cr4ANRy6itH9a6tWg

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier
a9ea0f25-b964-409f-bcce-c923266249b4

auth_time
1590150291

http://schemas.microsoft.com/identity/claims/identityprovider
local

http://schemas.microsoft.com/claims/authnmethodsreferences
pwd

given_name
John

family_name
Doe

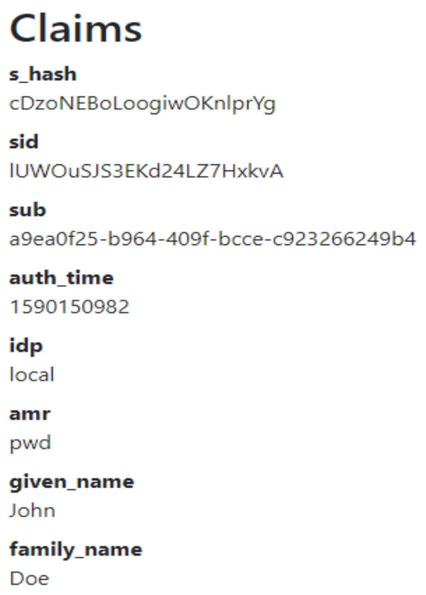
```
{
  "nbf": 1590150294,
  "exp": 1590150594,
  "iss": "https://localhost:5005",
  "aud": "companyemployeeclient",
  "nonce":
"637257470787346230.ZGYxNmIyMjQtNzA5MS00MThjLWFjZjktZGM5NW
ExOTcyMzIzZGEzNDBiNmYtMDgyMy00MWQ5LWEzODYtMjE5OWMzNzE5N2Rl
",
  "iat": 1590150294,
  "at_hash": "P2LNfjhROx97cVuMcrZdDg",
  "s_hash": "OPGgPY3CNovMHUdqyWVaUw",
  "sid": "eQYe8cr4ANRy6itH9a6tWg",
  "sub": "a9ea0f25-b964-409f-bcce-c923266249b4",
  "auth_time": 1590150291,
  "idp": "local",
  "amr": [
    "pwd"
  ]
}
```

What we want is to ensure that our claims stay the same as we define them, instead of being mapped to different claims. For example, the nameidentifier claim is mapped to the sub claim, and we want it to stay the sub claim. To do that, we have to slightly modify the **Program** class in the client project:

```
var builder = WebApplication.CreateBuilder(args);

JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();
```

The **JwtSecurityTokenHandler** lives inside the **System.IdentityModel.Tokens.Jwt** namespace.

Now, we can start our application, log out from the client, log in again and check the Privacy page:



Claims

s_hash
cDzoNEBoLoogiwOKnlprYg

sid
lUWOuSJS3EKd24LZ7HxkvA

sub
a9ea0f25-b964-409f-bcce-c923266249b4

auth_time
1590150982

idp
local

amr
pwd

given_name
John

family_name
Doe

We can see our claims are the same as we defined them at the IDP (Identity Provider) level.

If there are some claims we don't want to have in the token, we can remove them. To do that, we have to use the **ClaimActions** property in the OIDC (OpenIdConnect) configuration:

```
.AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, opt =>
{
    opt.SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    opt.Authority = "https://localhost:5005";
    opt.ClientId = "companyemployeeclient";
    opt.ResponseType = OpenIdConnectResponseType.Code;
    opt.SaveTokens = true;
    opt.ClientSecret = "CompanyEmployeeClientSecret";
    opt.GetClaimsFromUserInfoEndpoint = true;
    opt.ClaimActions.DeleteClaim("sid");
    opt.ClaimActions.DeleteClaim("idp");
});
```

The **DeleteClaim** method exists in the **Microsoft.AspNetCore.Authentication** namespace.

As a parameter, we pass a claim we want to remove. Now, if we start our client again and navigate to the Privacy page, these claims will be missing for sure (Log out and log in before checking the Privacy page).

If you don’t want to use the **DeleteClaim** method for each claim you want to remove, you can always use the **DeleteClaims** method:

```
opt.ClaimActions.DeleteClaims(new string[] { "sid", "idp" });
```

6.2 Adding Additional Claims

If we want to add additional claims to our token (address, for example), we can do that in a few simple steps. The first step is to support a new identity resource in the **Config** class in the IDP project :

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

Then, we have to add the Address scope to the AllowedScopes property for the Client configuration:

```
AllowedScopes =
{
    IdentityServerConstants.StandardScopes.OpenId,
    IdentityServerConstants.StandardScopes.Profile,
    IdentityServerConstants.StandardScopes.Address
},
```

After this, we have to open the TestUsers class and add a new claim for both users:

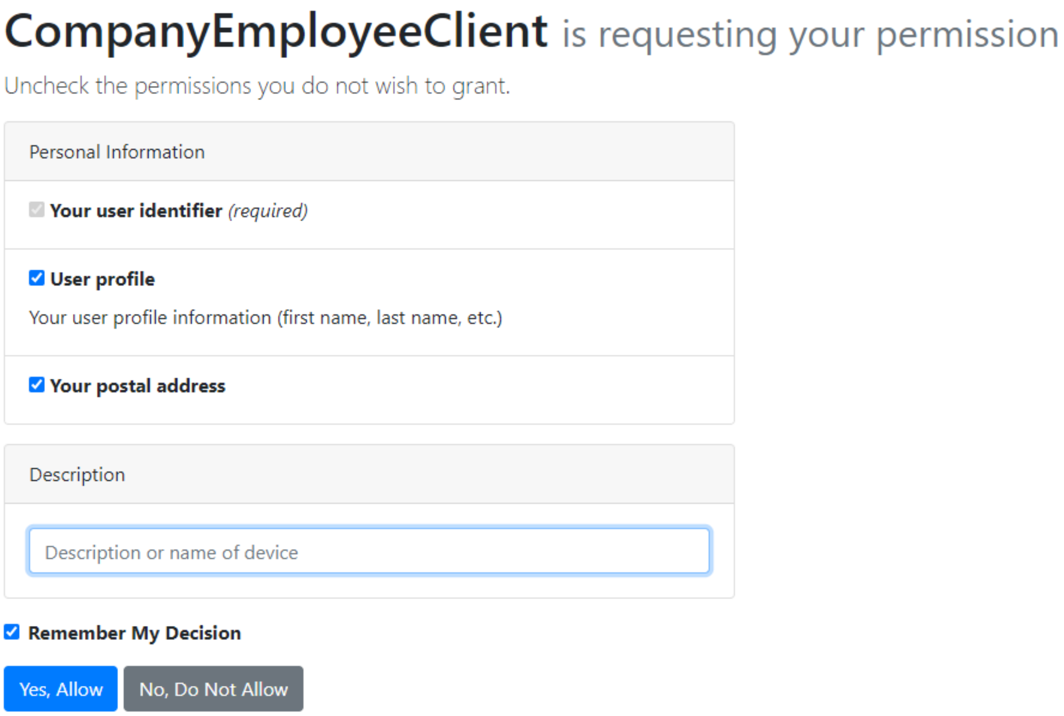
```
public static List<TestUser> Users =>
    new List<TestUser>
    {
        new TestUser
        {
            ...
            Claims = new List<Claim>
            {
                new Claim("given_name", "John"),
                new Claim("family_name", "Doe"),
                new Claim("address", "John Doe's Boulevard 323")
            }
        },
        new TestUser
        {
            ...
            Claims = new List<Claim>
            {
                new Claim("given_name", "Jane"),
                new Claim("family_name", "Doe"),
                new Claim("address", "Jane Doe's Avenue 214")
            }
        }
    };
```

By doing so, we are done with the IDP changes.

Now, let's move on to the client project and modify the OpenIdConnect configuration:

```
.AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, opt =>
{
    ...
    opt.ClaimActions.DeleteClaim("sid");
    opt.ClaimActions.DeleteClaim("idp");
    opt.Scope.Add("address");
});
```

Now, we can log in again:



CompanyEmployeeClient is requesting your permission

Uncheck the permissions you do not wish to grant.

Personal Information

☑ **Your user identifier** *(required)*

☑ **User profile**

Your user profile information (first name, last name, etc.)

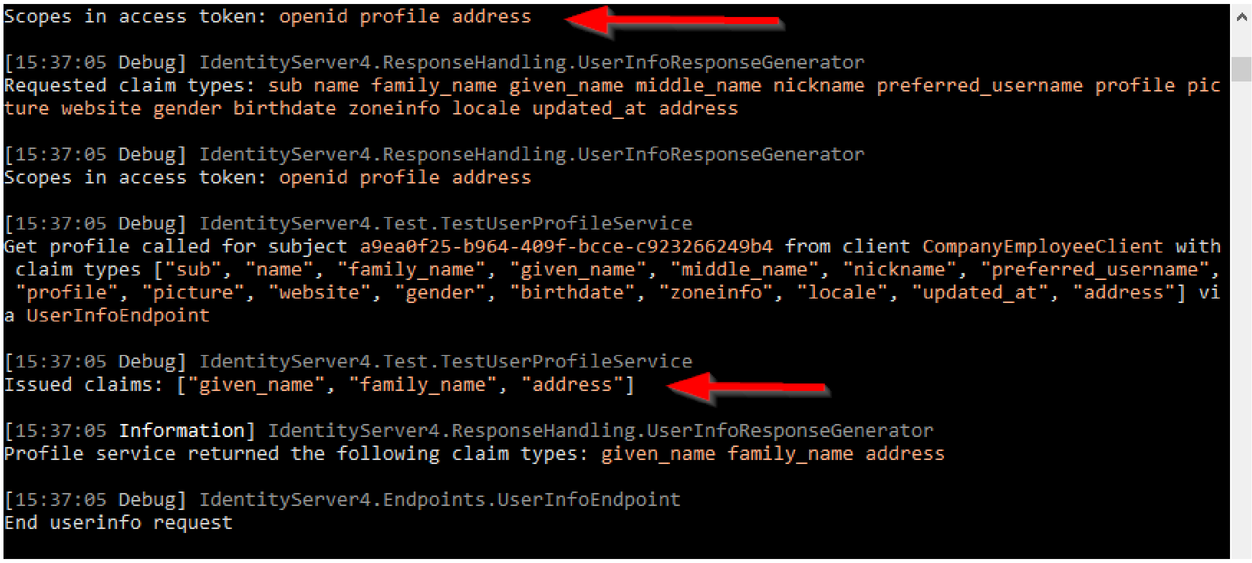
☑ **Your postal address**

Description

Description or name of device

☑ **Remember My Decision**

We can see a new address scope. But, if we inspect the Privacy page, we won't be able to find the address claim there. That's because we didn't map it to our claims. But, what we can do is inspect the console logs to make sure the IdentityServer returned our new claim:



```
Scopes in access token: openid profile address

[15:37:05 Debug] IdentityServer4.ResponseHandling.UserInfoResponseGenerator
Requested claim types: sub name family_name given_name middle_name nickname preferred_username profile pic
ture website gender birthdate zoneinfo locale updated_at address

[15:37:05 Debug] IdentityServer4.ResponseHandling.UserInfoResponseGenerator
Scopes in access token: openid profile address

[15:37:05 Debug] IdentityServer4.Test.TestUserProfileService
Get profile called for subject a9ea0f25-b964-409f-bcce-c923266249b4 from client CompanyEmployeeClient with
 claim types ["sub", "name", "family_name", "given_name", "middle_name", "nickname", "preferred_username",
 "profile", "picture", "website", "gender", "birthdate", "zoneinfo", "locale", "updated_at", "address"] vi
a UserInfoEndpoint

[15:37:05 Debug] IdentityServer4.Test.TestUserProfileService
Issued claims: ["given_name", "family_name", "address"]

[15:37:05 Information] IdentityServer4.ResponseHandling.UserInfoResponseGenerator
Profile service returned the following claim types: given_name family_name address

[15:37:05 Debug] IdentityServer4.Endpoints.UserInfoEndpoint
End userinfo request
```

If we want to include it, we can modify the OIDC configuration:

```
opt.ClaimActions.MapUniqueJsonKey("address", "address");
```

After we log in again, we can find the address claim on the Privacy page.

The important thing to mention here is: if you need a claim just for a part of the application (not for the entire application), the best practice is not to map it. You can always get it with the **IdentityModel** package, by sending the request to the **/userinfo** endpoint. By doing that, you ensure your cookies are small in size and that you always get up-to-date information from the userinfo endpoint.

6.3 Manually Calling the UserInfo Endpoint

Now, let’s see how we can extract the address claim from the **/userinfo** endpoint.

The first thing we have to do is to remove the **MapUniqueJsonKey(„address“, „address“)** statement from the OIDC configuration.

Then, we need to install the IdentityModel package:

IdentityModel by Dominick Baier,Brock Allen, 36.2M downloads
OpenID Connect & OAuth 2.0 client library

After the installation, we are going to configure another HttpClient in the **Program** class, just after our APIClient. But this time, for the IDP level:

```
builder.Services.AddHttpClient("IDPClient", client =>
{
    client.BaseAddress = new Uri("https://localhost:5005/");
    client.DefaultRequestHeaders.Clear();
    client.DefaultRequestHeaders.Add(HeaderNames.Accept, "application/json");
});
```

As soon as we create our IDPClient, we have to modify the **Privacy** action in the **Home** controller:

```
public async Task<IActionResult> Privacy()
{
    var idpClient = _httpClientFactory.CreateClient("IDPClient");
    var metaDataResponse = await idpClient.GetDiscoveryDocumentAsync();

    var accessToken = await
HttpContext.GetTokenAsync(OpenIdConnectParameterNames.AccessToken);

    var response = await idpClient.GetUserInfoAsync(new UserInfoRequest
    {
        Address = metaDataResponse.UserInfoEndpoint,
        Token = accessToken
    });

    if (response.IsError)
    {
        throw new Exception("Problem while fetching data from the UserInfo endpoint",
response.Exception);
    }

    var addressClaim = response.Claims.FirstOrDefault(c => c.Type.Equals("address"));

    User.AddIdentity(new ClaimsIdentity(new List<Claim> { new
Claim(addressClaim.Type.ToString(), addressClaim.Value.ToString()) }));

    return View();
}
```

So, we create a new client object and fetch the response from the IdentityServer with the **GetDiscoveryDocumentAsync** method. This response contains our required **/userinfo** endpoint's address. After that, we use the **UserInfo** address and extracted the access token to fetch the required user information. If the response is successful, we extract the address claim from the claims list and just add it to the **User.Claims** list (this is the list of Claims we iterate through in the Privacy view).

Also, we need these namespaces:

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.IdentityModel.Protocols.OpenIdConnect;
using System.Security.Claims;
```

Now, if we log in again and navigate to the Privacy page, the address claim will still be there. But this time, we extracted it manually. So basically, we can use this code only when we need it in our application.

6.4 Implementing Role-Based Authorization

Up until now, we have been working with Authentication by providing proof of who we are and the id token helped us in the process. So it makes sense to continue with the Authorization actions and take a look at Role-Based Authorization actions.

To begin with this type of authorization, we have to provide a role claim in the TestUser class at the IDP level:

```
new TestUser
{
    ...
    Claims = new List<Claim>
    {
        ...
        new Claim("address", "John Doe's Boulevard 323"),
        new Claim("role", "Administrator")
    }
},
new TestUser
{
    ...
    Claims = new List<Claim>
    {
        ...
        new Claim("address", "Jane Doe's Avenue 214"),
        new Claim("role", "Visitor")
    }
}
```

We’ve finished working with users. Now, let’s move on to the **Config** class and create a new identity scope in the **Ids** property:

```
public static IEnumerable<IdentityResource> Ids =>
    new IdentityResource[]
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Profile(),
        new IdentityResources.Address(),
        new IdentityResource("roles", "User role(s)", new List<string> { "role" })
    };
```

Since a role is not a standard identity resource, we have to create it ourselves. We add a scope name for the resource, then add a display name, and finally a list of claims that must be returned when the application asks for this “roles” scope.

Additionally, we have to add a new allowed scope for our client application:

```
new Client
{
    ...
    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        IdentityServerConstants.StandardScopes.Address ,
        "roles"
    },
    ...
}
```

With all of these in place, we are done with the IDP level modifications. Now, we can move on to the client application by updating the OIDC configuration to support roles scope:

```
.AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, opt =>
{
    ...
    opt.Scope.Add("address");
    opt.Scope.Add("roles");
    opt.ClaimActions.MapUniqueJsonKey("role", "role");
});
```

It isn’t enough just to add the **roles** scope to the configuration, we need to map it as well to have it in a claims list.

So, what we want to do with this role claim is to allow only the user with the Administrator role to use Create, Update, Details, and Delete actions. To do that, let’s modify the **Companies** view:

```
@if (User.IsInRole("Administrator"))
{
    <p>
        <a asp-action="Create">Create New</a>
    </p>
}
<table class="table">
    ...
    <tbody>
```

```
        @foreach (var item in Model)
        {
            <tr>
                ...
                @if (User.IsInRole("Administrator"))
                {
                    <td>
                        @Html.ActionLink("Edit", "Edit", new {}) |
                        @Html.ActionLink("Details", "Details", new {}) |
                        @Html.ActionLink("Delete", "Delete", new {})
                    </td>
                }
            </tr>
        }
    </tbody>
</table>
```

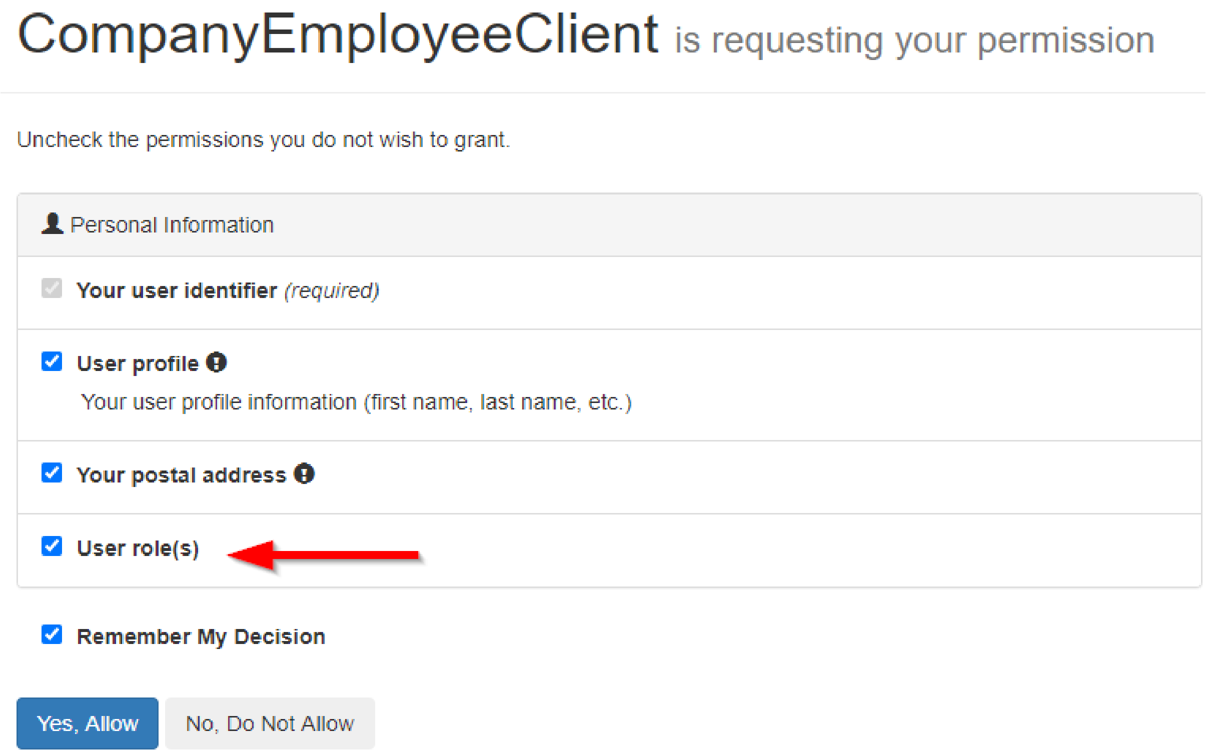
Here, we wrap the actions we want to allow only to the administrator with the check if the user is in the Administrator role. We do that by using the **User** object and **IsInRole** method where we pass the value of the role.

Finally, we have to state where our framework can find the user's role:

```
.AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, opt =>
{
    ...
    opt.Scope.Add("roles");
    opt.ClaimActions.MapUniqueJsonKey("role", "role");

    opt.TokenValidationParameters = new TokenValidationParameters
    {
        RoleClaimType = JwtClaimTypes.Role
    };
});
```

The **TokenValidationParameters** class exists in the **Microsoft.IdentityModel.Tokens** namespace.

Now, we can start our applications and log in with Jane's account:



CompanyEmployeeClient is requesting your permission

Uncheck the permissions you do not wish to grant.

Personal Information

☑ **Your user identifier** *(required)*

☑ **User profile**
Your user profile information (first name, last name, etc.)

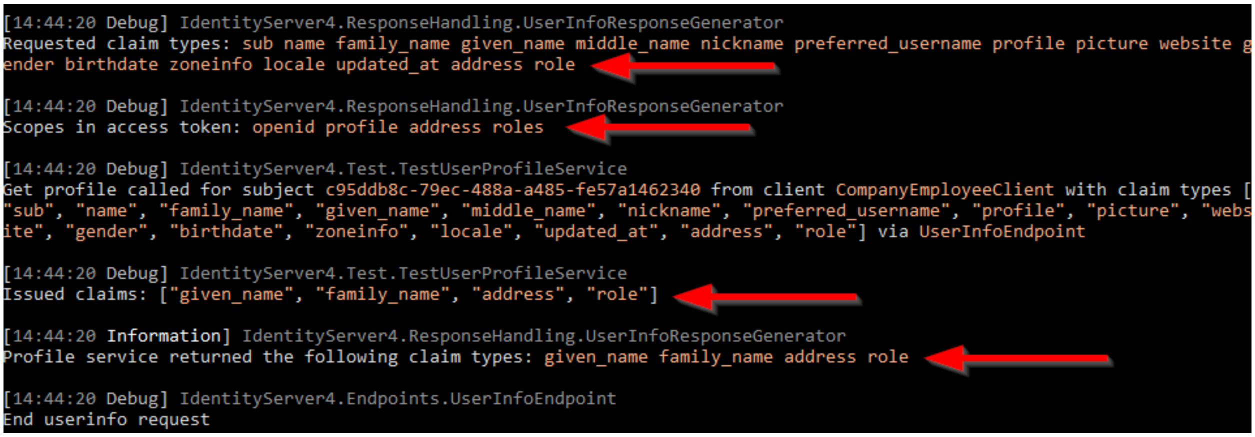
☑ **Your postal address**

☑ **User role(s)**

☑ **Remember My Decision**

We can see an additional scope in the Consent screen.

As soon as we allow this, the Home screen will appear. Let's just inspect the console logs:

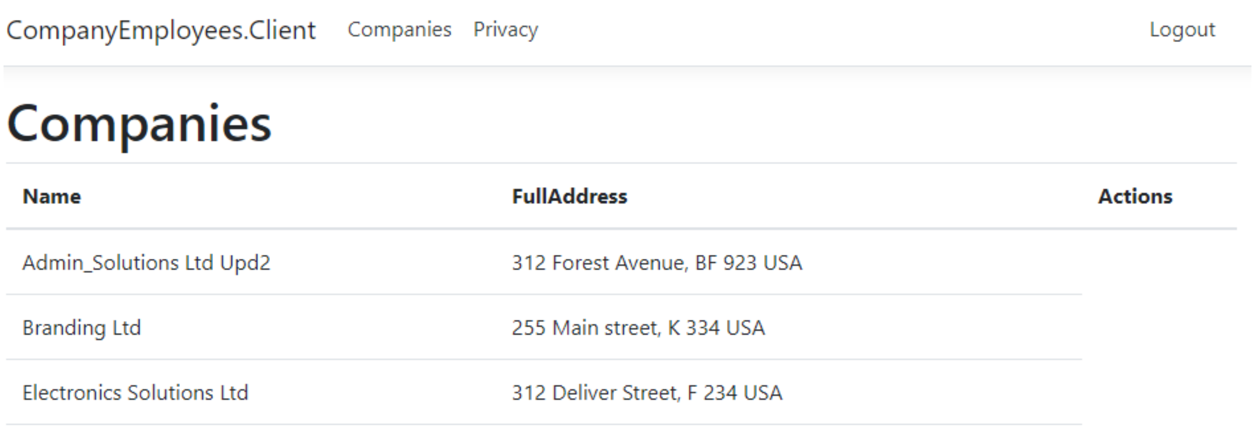


```
[14:44:20 Debug] IdentityServer4.ResponseHandling.UserInfoResponseGenerator
Requested claim types: sub name family_name given_name middle_name nickname preferred_username profile picture website g
ender birthdate zoneinfo locale updated_at address role

[14:44:20 Debug] IdentityServer4.ResponseHandling.UserInfoResponseGenerator
Scopes in access token: openid profile address roles

[14:44:20 Debug] IdentityServer4.Test.TestUserProfileService
Get profile called for subject c95ddb8c-79ec-488a-a485-fe57a1462340 from client CompanyEmployeeClient with claim types [
"sub", "name", "family_name", "given_name", "middle_name", "nickname", "preferred_username", "profile", "picture", "webs
ite", "gender", "birthdate", "zoneinfo", "locale", "updated_at", "address", "role"] via UserInfoEndpoint

[14:44:20 Debug] IdentityServer4.Test.TestUserProfileService
Issued claims: ["given_name", "family_name", "address", "role"]

[14:44:20 Information] IdentityServer4.ResponseHandling.UserInfoResponseGenerator
Profile service returned the following claim types: given_name family_name address role

[14:44:20 Debug] IdentityServer4.Endpoints.UserInfoEndpoint
End userinfo request
```

We can see the role claim was issued and returned.

For Duende the logs are organized a bit differently, but you will be able to find the roles claim.

Now, let’s navigate to the Companies view:



CompanyEmployees.Client Companies Privacy Logout

Companies

Name	FullAddress	Actions
Admin_Solutions Ltd Upd2	312 Forest Avenue, BF 923 USA	
Branding Ltd	255 Main street, K 334 USA	
Electronics Solutions Ltd	312 Deliver Street, F 234 USA	

We can’t find additional actions on this page, but if we log out and log in with John’s account, we will be able to find the missing actions.

Excellent.

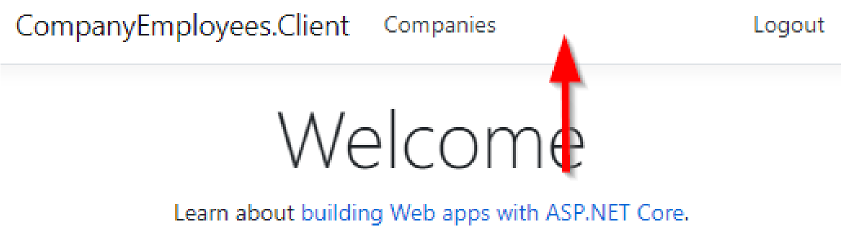
But can we protect our endpoints with roles as well? Of course. Let’s see how it’s done.

6.5 Protecting Endpoints with Roles

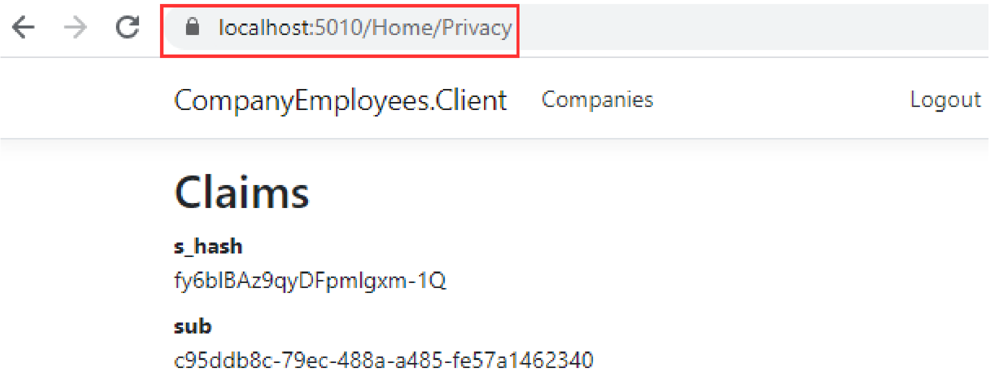
Le’s say, for example, only the Administrator users can access the Privacy page. Well, with the same action from the previous part, we can show the Privacy link in the **_Layout** view:

```
@if (User.IsInRole("Administrator"))
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
    </li>
}
```

Now, if we log in as Jane, we won’t be able to see the privacy link:



Even though we can't see the Privacy link, we still have access to the Privacy page by entering a valid URI address:



Claims

s_hash

fy6blBAz9qyDFpmlgxm-1Q

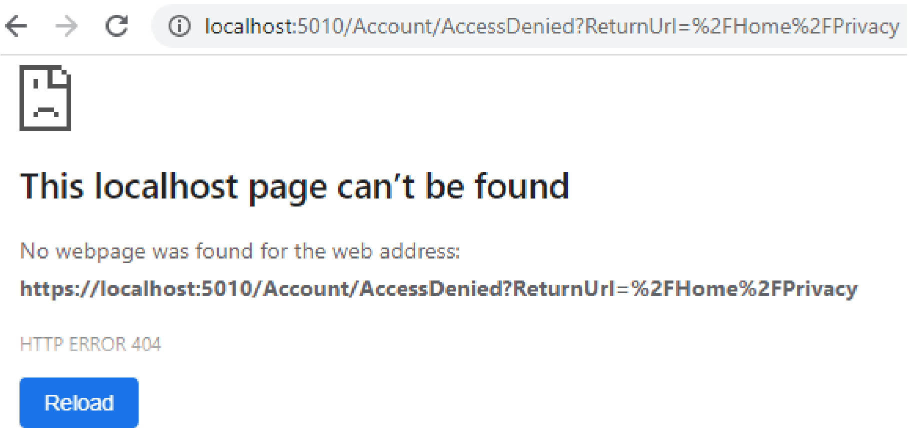
sub

c95ddb8c-79ec-488a-a485-fe57a1462340

So, what we have to do is to protect our `Privacy` endpoint with the user's role:

```
[Authorize(Roles = "Administrator")]
public async Task<IActionResult> Privacy()
```

Now, if we log out, log in again as Jane and try to use the URI address to access the privacy page, we won't be able to do that:



This localhost page can't be found

No webpage was found for the web address: **https://localhost:5010/Account/AccessDenied?ReturnUrl=%2FHome%2FPrivacy**

HTTP ERROR 404

The application redirects us to the **/Account/AccessDenied** page, but we get a 404 error code because we don't have that page.

So, let's create it.

The first thing we are going to do is create the **AccessDenied** action in the Auth controller:

```
public IActionResult AccessDenied()
{
    return View();
}
```

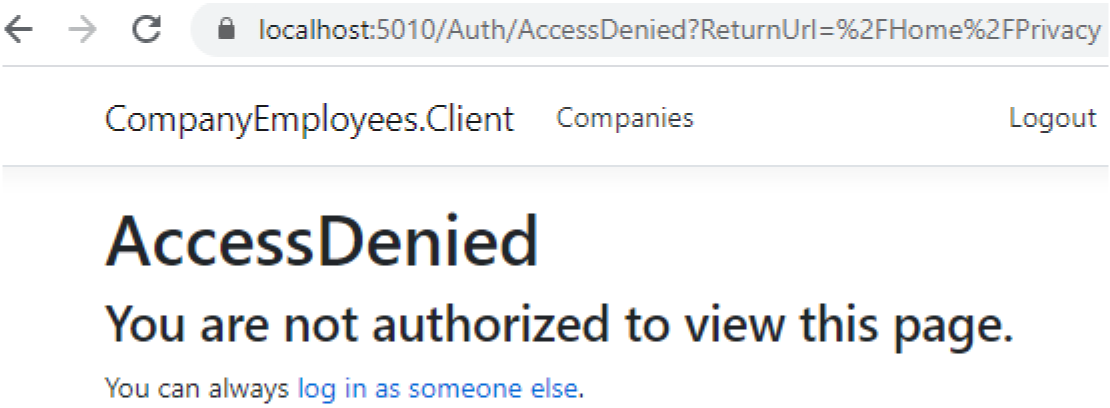
Then, let's create a view for this action:

```
@{
    ViewData["Title"] = "AccessDenied";
}

<h1>AccessDenied</h1>

<h3>You are not authorized to view this page.</h3>

<p>
    You can always <a asp-controller="Auth" asp-action="Logout">log in as someone
else</a>.
</p>
```

We have to pay attention to one thing. As we saw, if a user doesn't have access to a page, they are redirected by default to Account/AccessDenied address. But, since we don't have the Account controller, we have to override this setting. To do that, let's modify the **AddCookie** method in the **Program** class:

```
.AddCookie(CookieAuthenticationDefaults.AuthenticationScheme, opt => {
    opt.AccessDeniedPath = "/Auth/AccessDenied";
})
```

Now, if we log in as Jane and try to navigate to the /Home/Privacy URI, we are going to be navigated to our newly created page:



localhost:5010/Auth/AccessDenied?ReturnUrl=%2FHome%2FPrivacy

AccessDenied

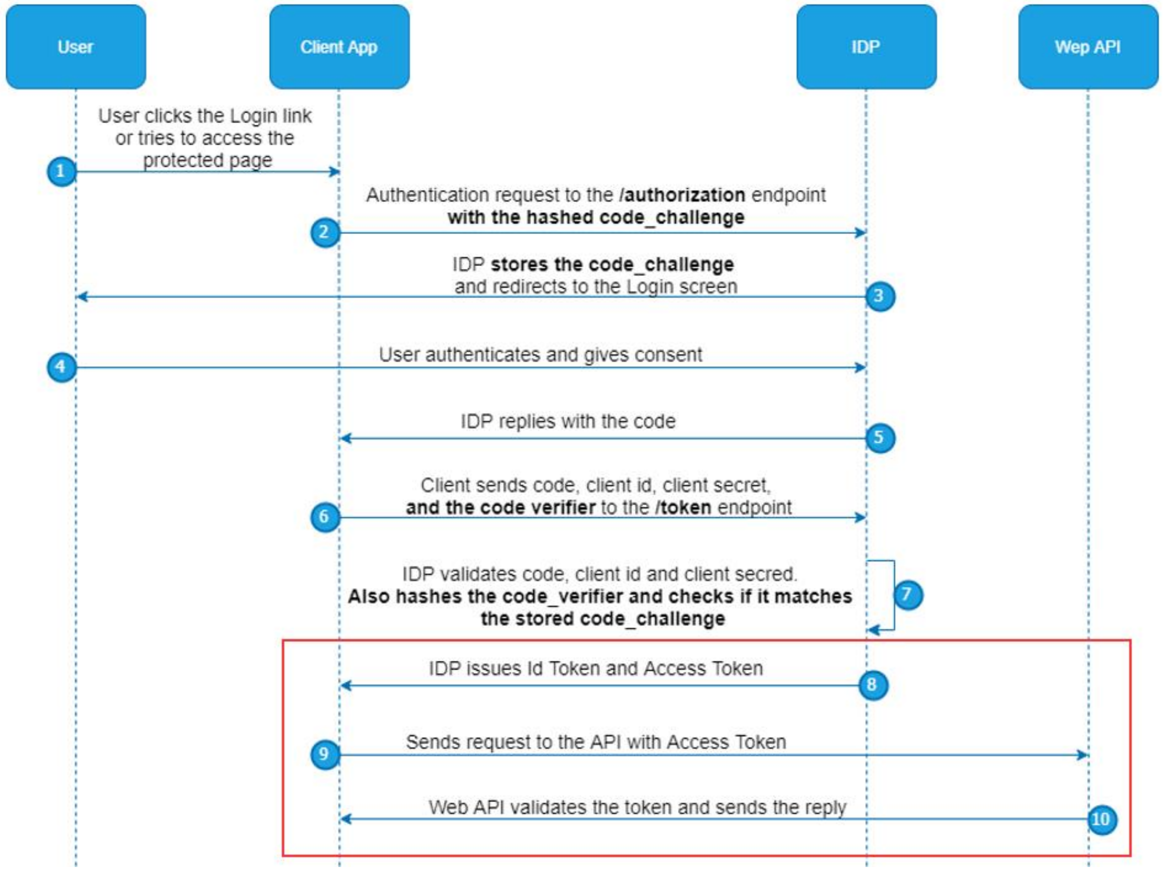
You are not authorized to view this page.

You can always log in as someone else.

And, of course, if we click the "log in as someone else" link, we are going to be logged out and navigated to the Home page.

7 SECURING THE WEB API

For this section of the book, we are going to pay closer attention to the last three steps of our Authorization Code Flow diagram:


User
Client App
IDP
Wep API
1 User clicks the Login link or tries to access the protected page
2 Authentication request to the /authorization endpoint with the hashed code_challenge
3 IDP stores the code_challenge and redirects to the Login screen
4 User authenticates and gives consent
5 IDP replies with the code
6 Client sends code, client id, client secret, and the code verifier to the /token endpoint
7 IDP validates code, client id and client secred. Also hashes the code_verifier and checks if it matches the stored code_challenge
8 IDP issues Id Token and Access Token
9 Sends request to the API with Access Token
10 Web API validates the token and sends the reply

As we can see in step eight, IDP provides the access token and the id token. Then, the client application stores the access token and sends it as a Bearer token with each request to the API. At the API level, this token is validated and access is granted to the protected resources. In this section, we are going to cover these three steps.

7.1 Implementing Web API Security

The first thing we want to do is to add a new API scope in the `Config` class at the IDP level:

```
public static IEnumerable<ApiScope> ApiScopes =>
    new ApiScope[]
    {
        new ApiScope("companyemployeeapi.scope", "CompanyEmployee API Scope")
    };
```

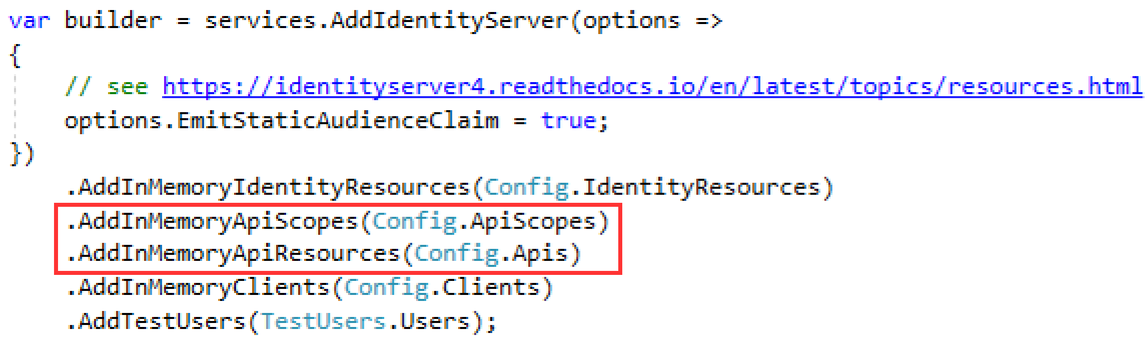
After that, we have to configure the API Resource:

```
public static IEnumerable<ApiResource> Apis =>
      new ApiResource[]
      {
            new ApiResource("companyemployeeapi", "CompanyEmployee API")
            {
                  Scopes = { "companyemployeeapi.scope" }
            }
      };
```

Then, in the AllowedScopes, we want to add this API resource:

```
AllowedScopes =
{
    IdentityServerConstants.StandardScopes.OpenId,
    IdentityServerConstants.StandardScopes.Profile,
    IdentityServerConstants.StandardScopes.Address ,
    "roles",
    "companyemployeeapi.scope"
},
```

If we check the code in the **ConfigureServices** method, we are going to see that IDP is already configured to use API resources:



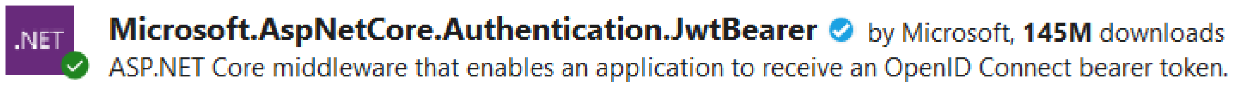
```
var builder = services.AddIdentityServer(options =>
{
    // see https://identityserver4.readthedocs.io/en/latest/topics/resources.html
    options.EmitStaticAudienceClaim = true;
})
    .AddInMemoryIdentityResources(Config.IdentityResources)
    .AddInMemoryApiScopes(Config.ApiScopes)
    .AddInMemoryApiResources(Config.Apis)
    .AddInMemoryClients(Config.Clients)
    .AddTestUsers(TestUsers.Users);
```

And that’s all we have to modify at the level of IDP. We can move on to the client application.

In the **Program** class, we are going to add a new scope to the OIDC configuration:

```
AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, opt =>
{
    ...
    opt.ClaimActions.MapUniqueJsonKey("role", "role");
    opt.Scope.Add("companyemployeeapi.scope");

    opt.TokenValidationParameters = new TokenValidationParameters
    {
        RoleClaimType = JwtClaimTypes.Role
    };
});
```

Now, in the API project, we want to install an access token validation package:



Microsoft.AspNetCore.Authentication.JwtBearer by Microsoft, **145M** downloads
ASP.NET Core middleware that enables an application to receive an OpenID Connect bearer token.

After the installation completes, we want to register the authentication handler in the **ServiceExtensions** class in the **Extensions** folder:

```
public static void ConfigureAuthenticationHandler(this IServiceCollection services)
       => services.AddAuthentication("Bearer")
       .AddJwtBearer("Bearer", options =>
       {
              options.Authority = "https://localhost:5005";
              options.Audience = "companyemployeeapi";
       });
```

We are calling the **AddAuthentication** method and passing a scheme **Bearer**. Then, we need to enable the JWT bearer authentication using the **AddJwtBearer** method. It accepts a scheme name and an action delegate as parameters, and by using that delegate, we configure the **Authority** (the address of our IDP project) and the **Audience**.

Now, we have to invoke this extension method in the **Program** class:

```
builder.Services.ConfigureAuthenticationHandler();

builder.Services.AddControllers(config =>
```

Additionally, we have to add authentication to the application's pipeline:

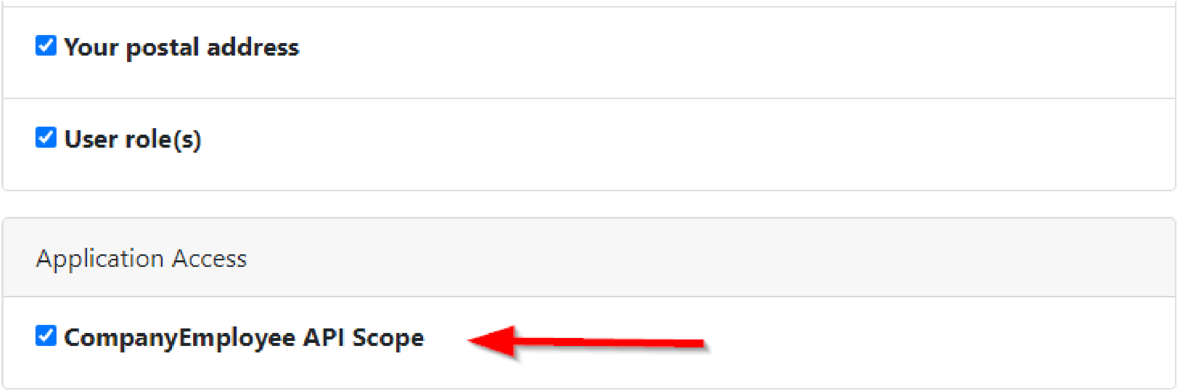
```
app.UseAuthentication();
app.UseAuthorization();
```

Lastly, we want to protect our companies endpoint. To do that, we are going to add the **[Authorize]** attribute just before the **GetCompanies** action in the **Companies** controller:

```
[HttpGet]
[Authorize]
public async Task<IActionResult> GetCompanies()
```

Excellent.

Now, let's start our applications and log in as John:



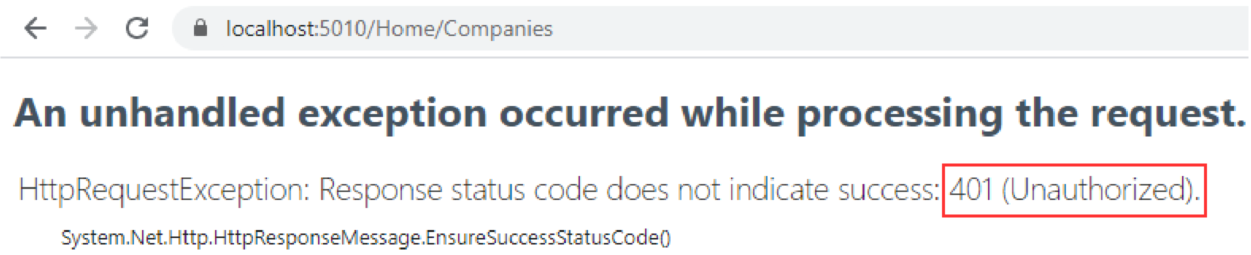
☑ Your postal address

☑ User role(s)

Application Access

☑ CompanyEmployee API Scope

On the consent screen, we can see the new API scope. As soon as we allow this and navigate to the Companies page, we get a 401 Unauthorized response:



localhost:5010/Home/Companies

An unhandled exception occurred while processing the request.

HttpRequestException: Response status code does not indicate success: 401 (Unauthorized).

System.Net.Http.HttpResponseMessage.EnsureSuccessStatusCode()

The API returns this response because we didn't send the access token in the request.

7.2 Providing an Access Token When Calling an API

Since we have to pass the access token with each call to the API, the best practice is to implement some reusable logic for applying that token to the request. That's exactly what we are going to do here.

In the client application, we are going to create a new **Handlers** folder and inside it a new **BearerTokenHandler** class:

```
public class BearerTokenHandler : DelegatingHandler
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    public BearerTokenHandler(IHttpContextAccessor httpContextAccessor)
    {
        _httpContextAccessor = httpContextAccessor;
    }

    protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage
request, CancellationToken cancellationToken)
    {
        var accessToken = await
_httpContextAccessor.HttpContext.GetTokenAsync(OpenIdConnectParameterNames.AccessToken);

        if (!string.IsNullOrWhiteSpace(accessToken))
            request.SetBearerToken(accessToken);

        return await base.SendAsync(request, cancellationToken);
    }
}
```

This code requires the following using statements:

```
using IdentityModel.Client;
using Microsoft.AspNetCore.Authentication;
using Microsoft.IdentityModel.Protocols.OpenIdConnect;
```

Now, let's explain the code.

Our class inherits from the **DelegatingHandler** class that allows us to delegate the processing of HTTP response messages to another handler. This is exactly what we need since we don't want to override the default way HTTP sends messages and receives them, but we want to add something to it.

Then, we inject the **IHttpContextAccessor** interface to help us access the **HttpContext** property. After that, we override the **SendAsync** method from the **DelegatingHandler** class. Inside it, we retrieve the access token by calling the **GetTokenAsync** method with one input parameter – the name of the token we want to retrieve. If the token exists, we attach it to the request by using the **SetBearerToken** method. Finally, we call the **SendAsync** method from the base class.

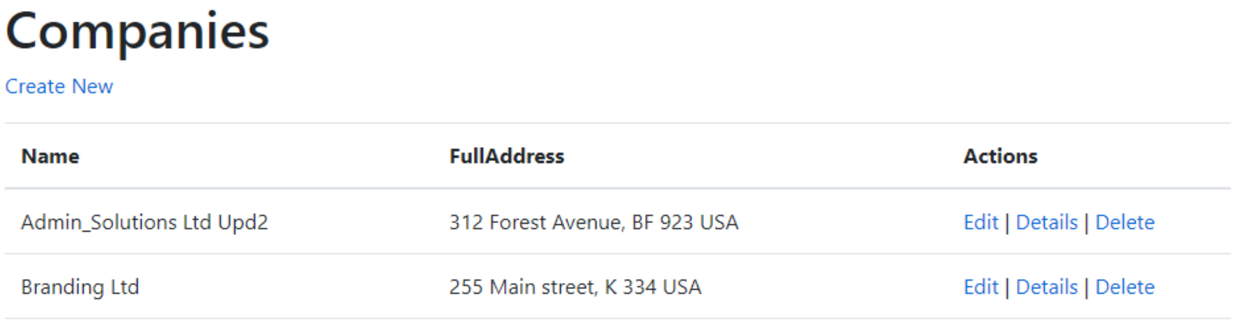
Now, we have to register the **IHttpContextAccessor** interface and our **BearerTokenHandler** class in the **Program** class:

```
builder.Services.AddHttpContextAccessor();
builder.Services.AddTransient<BearerTokenHandler>();
```

And to provide an additional message handler to our APIClient configuration:

```
 services.AddHttpClient("APIClient", client =>
{
    client.BaseAddress = new Uri("https://localhost:5001/");
    client.DefaultRequestHeaders.Clear();
    client.DefaultRequestHeaders.Add(HeaderNames.Accept, "application/json");
}).AddHttpMessageHandler<BearerTokenHandler>();
```

Now, let's start all the applications, log in as John and click on the Companies link in the menu:



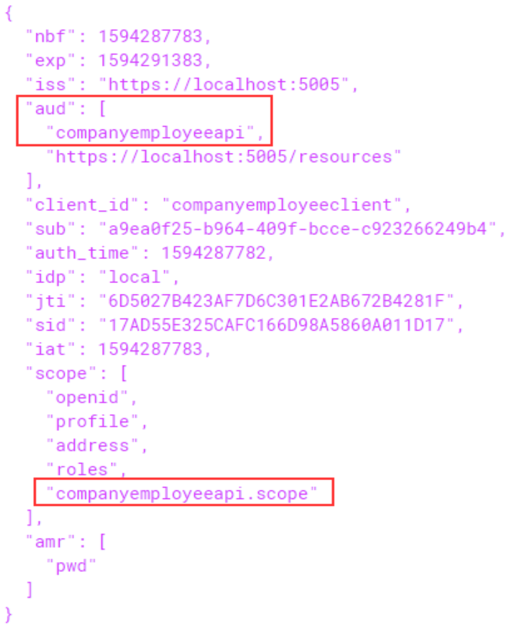
Companies

Create New

Name	FullAddress	Actions
Admin_Solutions Ltd Upd2	312 Forest Avenue, BF 923 USA	Edit | Details | Delete
Branding Ltd	255 Main street, K 334 USA	Edit | Details | Delete

This time, we can see our data, which means the client sent the access token to the API where it was validated, and the access to the protected resource was granted.

Now, if we navigate to the Privacy page, copy the access token and decode it, we will find the companyemployeeapi scope included in the list of scopes, and it is the value for the audience as well:



```
{
  "nbf": 1594287783,
  "exp": 1594291383,
  "iss": "https://localhost:5005",
  "aud": [
    "companyemployeeapi",
    "https://localhost:5005/resources"
  ],
  "client_id": "companyemployeeclient",
  "sub": "a9ea0f25-b964-409f-bcce-c923266249b4",
  "auth_time": 1594287782,
  "idp": "local",
  "jti": "6D5027B423AF7D6C301E2AB672B4281F",
  "sid": "17AD55E325CAFC166D98A5860A011D17",
  "iat": 1594287783,
  "scope": [
    "openid",
    "profile",
    "address",
    "roles",
    "companyemployeeapi.scope"
  ],
  "amr": [
    "pwd"
  ]
}
```

7.3 Redirecting the Client to the Unauthorized Page

If we log out from the client, log in again, but this time uncheck the CompanyEmployee API option and navigate to the Companies page, we are going to get the 401 response. But, since we already have the AccessDenied page, we can use it to create a better user experience.

To do that, we are going to modify the **Companies** action in the **Home** controller:

```
[Authorize]
public async Task<IActionResult> Companies()
{
    var httpClient = _httpClientFactory.CreateClient("APIClient");

    var response = await httpClient.GetAsync("api/companies").ConfigureAwait(false);

    if(response.IsSuccessStatusCode)
    {
        var companiesString = await response.Content.ReadAsStringAsync();
        var companies =
JsonSerializer.Deserialize<List<CompanyViewModel>>(companiesString, new
JsonSerializerOptions { PropertyNameCaseInsensitive = true });

        return View(companies);
    }
    else if (response.StatusCode == HttpStatusCode.Unauthorized || response.StatusCode ==
HttpStatusCode.Forbidden)
    {
        return RedirectToAction("AccessDenied", "Auth");
    }

    throw new Exception("There is a problem accessing the API.");
}
```

Now, if we uncheck the CompanyEmployee API option and navigate to the Companies page, we are going to be redirected to the AccessDenied page for sure.

7.4 Using Policies to Secure The Application

We have explained how to use Role Base Authorization (RBA) to protect an application. But, there is another approach – Attribute-based Access Control (ABAC), which is more suited for authorization with complex rules. When we talk about complex rules, we have something like this in mind – for example, the user has to be authenticated, have a certain role, and live in a specific country. So, as you can see, there are several rules for the authorization process to be completed, and the best way to configure that is by using the ABAC approach. This approach is also known as Policy-based Access Control because it uses policies to grant access to protected resources to users.

Let’s get started and learn how to use Policies in our client application.

We have to modify the configuration class at the IDP level first. So, let’s add a new identity resource:

```
public static IEnumerable<IdentityResource> Ids =>
    new IdentityResource[]
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Profile(),
        new IdentityResources.Address(),
        new IdentityResource("roles", "User role(s)", new List<string> { "role" }),
        new IdentityResource("country", "Your country", new List<string> { "country" })
    };
```

We want our client to be able to request this scope, so let’s modify the allowed scopes for the MVC client:

```
AllowedScopes =
{
    IdentityServerConstants.StandardScopes.OpenId,
    IdentityServerConstants.StandardScopes.Profile,
    IdentityServerConstants.StandardScopes.Address,
    "roles",
    "companyemployeeapi",
    "country"
},
```

The last thing we have to do at the IDP level is to modify our test users by adding a new claim in the claims list to both John and Jane:

```
new Claim("country", "USA")
```

Excellent. That’s it regarding the IDP level modifications.

Now, at the client level, we have to modify the OIDC configuration with familiar actions:

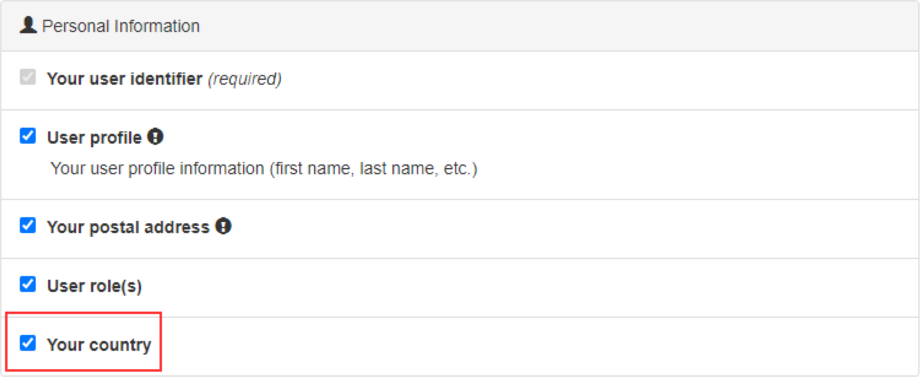
```
opt.Scope.Add("country");
opt.ClaimActions.MapUniqueJsonKey("country", "country");
```

Additionally, we have to create our policies right before the **AddControllersWithViews** method:

```
builder.Services.AddAuthorization(authOpt =>
{
    authOpt.AddPolicy("CanCreateAndModifyData", policyBuilder =>
    {
        policyBuilder.RequireAuthenticatedUser();
        policyBuilder.RequireRole("role", "Administrator");
        policyBuilder.RequireClaim("country", "USA");
    });
});

services.AddControllersWithViews();
```

We use the **AddAuthorization** method to add Authorization in the service collection. By using the **AddPolicy** method, we provide the policy name and all the policy conditions that are required so that the authorization process is completed. As you can see, we require the user to be an authenticated administrator from the USA.

So, once we log in, we are going to see an additional scope in our consent screen. If we navigate to the Privacy page, we are going to see the country claim:



Personal Information

☑ Your user identifier *(required)*

☑ User profile
Your user profile information (first name, last name, etc.)

☑ Your postal address

☑ User role(s)

☑ Your country

But, we are not using the policy we created for authorization. At least not yet. So, let's change that.

To do that, let's modify the **Companies** view:

```
@using  Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService
...
<h1>Companies</h1>

@if ((await AuthorizationService.AuthorizeAsync(User,
"CanCreateAndModifyData")).Succeeded)
{
    <p>
        <a asp-action="Create">Create New</a>
    </p>
}
<table class="table">
    ...
    <tbody>
        @foreach (var item in Model)
        {
            ...
                @if ((await AuthorizationService.AuthorizeAsync(User,
"CanCreateAndModifyData")).Succeeded)
                {
                    <td>
                        @Html.ActionLink("Edit", "Edit", new {}) |
                        @Html.ActionLink("Details", "Details", new {}) |
                        @Html.ActionLink("Delete", "Delete", new {})
                    </td>
                }
            </tr>
        }
    </tbody>
</table>
```

Additionally, to protect the **Privacy** action, we have to modify the Authorize attribute:

```
[Authorize(Policy = "CanCreateAndModifyData")]
public async Task<IActionResult> Privacy()
```

And that's it. We can test this with both John and Jane. For sure, with Jane's account, we won't be able to see Create, Update, Delete, and Details link, and we are going to be redirected to the AccessDenied page if we try to navigate to the Privacy page. You can try it out yourself.

Additional Note

To send the role claim to the API through the Access Token, we have to add a claim to the **ApiResource** in the **Config** class:

```
new ApiResource("companyemployeeapi", "CompanyEmployee API")
{
      Scopes = { "companyemployeeapi.scope" },
      UserClaims = new List<string> { "role" }
}
```

Then, you can add the **Roles** property to the **[Authorize]** attribute on the API level.

8 Handling Token Expiration

If we leave our browser open for some time and after that try to access API's protected resource, we will see that it's not possible anymore. The reason for that is the token's limited lifetime – the token has probably expired.

The lifetime for the identity token is five minutes by default, and, after that, it shouldn't be accepted in client applications for processing. This means the client application shouldn't use it to create the claims identity. In this situation, it's up to the client to create a logic regarding when access to the client application should expire. Usually, we want to keep the user logged in as long as they are active.

The situation with access tokens is different. They have a longer lifetime – one hour by default and after expiration, we have to provide a new one to access the API. We can be logged in to the client application but we won't have access to the API's resources. So, we can see that the client is not responsible for renewing the access token, this is the IDP's responsibility.

8.1 Working with Token Lifetime and Expiration

For each client, we can configure three types of lifetimes:

- IdentityTokenLifetime,
- AuthorizationCodeLifetime,
- AccessTokenLifetime

In our IDP application, we are going to leave the first two as is, with their default values. The default value for AccessTokenLifetime is one hour, and we are going to reduce that value in the Config class:

```
public static IEnumerable<Client> Clients =>
```

```
    new Client[]
    {
        new Client
        {
            ...
            AccessTokenLifetime = 120
        }
    };
```

We have set this value to 2 minutes (120 seconds).

It should be mentioned that the access token validation middleware adds 5 more minutes to the expiration lifetime to handle small offsets in out-of-sync clock times between the API and IDP, so if we log in, wait two minutes and try to access the API, we are going to be able to do that. We have to wait a little longer than that, at least 7 minutes. After that time expires, once we try to navigate to the Companies page, we are going to be redirected to the AccessDenied page.

8.2 Refreshing the Token

When a token expires, the flow could be triggered again and the user could be redirected to the login page and then to the consent page. But, doing this over and over again is not user-friendly at all. Luckily, with a confidential client, we don’t have to do this. This type of client application can use refresh tokens over the back channel, without user interaction. The refresh token is a credential to get new tokens, usually before the original token expires.

So, the flow is similar to the diagram we have already seen but with minor modifications. When the client application does the authentication with the user's credentials, it provides the refresh token as well in the request’s body. At the IDP level, this refresh token is validated and IDP sends back the id token, access token, and optionally the refresh token, so we could refresh again later on.

The offline access scope is required to support refresh tokens, so let's configure that in the Config class:

```
public static IEnumerable<Client> Clients =>
    new Client[]
    {
        new Client
        {
            ...
            AccessTokenLifetime = 120,
            AllowOfflineAccess = true
        }
    };
```

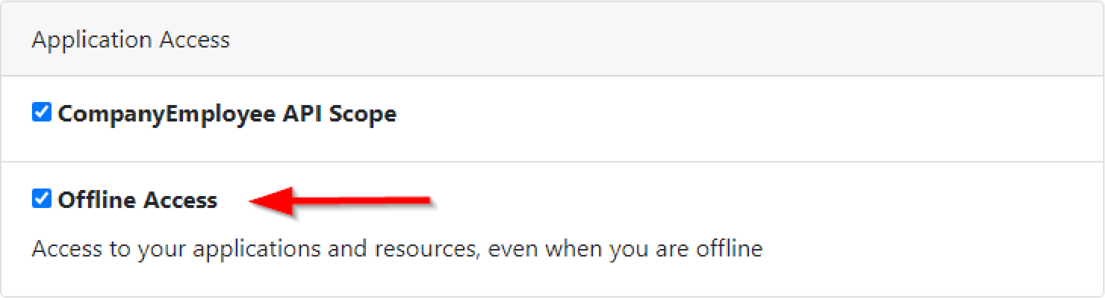
Additionally, we want our claims to be updated as soon as the token refreshes, and for that we have to configure one more property right after the AllowOfflineAcces property:

```
AllowOfflineAccess = true,
UpdateAccessTokenClaimsOnRefresh = true
```

That's it regarding the IDP. Now, we can move on to the client application and request this new offline_access scope in the OIDC configuration:

```
.AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, opt =>
{
    ...
    opt.Scope.Add("offline_access");
});
```

Excellent. We are ready to start our applications and log in. We are going to see a new Offline Access scope on the consent screen:



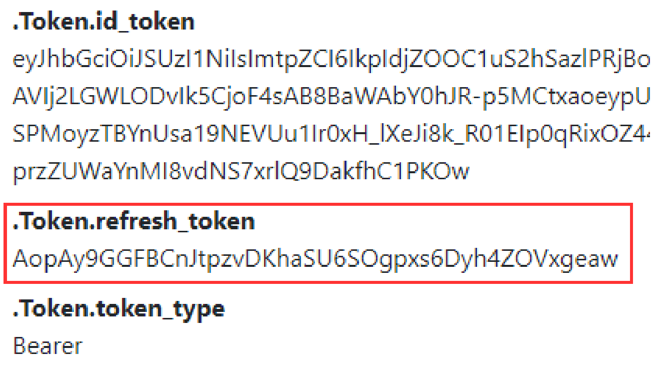
Application Access

☑ **CompanyEmployee API Scope**

☑ **Offline Access**

Access to your applications and resources, even when you are offline

If we navigate to the Privacy page, we are going to see the refresh token:



.Token.id_token

eyJhbGciOiJSUzI1NiIsImtpZCI6IkpIdjZOOC1uS2hSazlPRjBo
AVIj2LGWLODvIk5CjoF4sAB8BaWAbY0hJR-p5MCtxaoeypU
SPMoyzTBYnUsa19NEVUu1Ir0xH_lXeJi8k_R01EIp0qRixOZ4
przZUWaYnMI8vdNS7xrlQ9DakfhC1PKOw

.Token.refresh_token

AopAy9GGFBCnJtpzvDKhaSU6SOgpxs6Dyh4ZOVxgeaw

.Token.token_type

Bearer

Now, let’s use this refresh token in our application.

Certainly, we want to do this in a centralized location and we want to refresh the token a little bit before the original one expires.

That said, let’s modify the **BearerTokenHandler** class:

```
private readonly IHttpContextAccessor _httpContextAccessor;
private readonly IHttpClientFactory _httpClientFactory;

public BearerTokenHandler(IHttpContextAccessor httpContextAccessor, IHttpClientFactory
httpClientFactory)
{
    _httpContextAccessor = httpContextAccessor;
    _httpClientFactory = httpClientFactory;
}
```

We inject the **IHttpClientFactory** interface because we are going to need it to create an HttpClient object to access IDP.

Next, let’s modify the **SendAsync** method:

```
protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
CancellationToken cancellationToken)
{
    var accessToken = await GetAccessTokenAsync();

    if (!string.IsNullOrWhiteSpace(accessToken))
        request.SetBearerToken(accessToken);

    return await base.SendAsync(request, cancellationToken);
}
```

Here, we just replace the previous code for fetching the access token with a private **GetAccessTokenAsync** method. So, let’s inspect it:

```
public async Task<string> GetAccessTokenAsync()
{
    var expiresAtToken = await _httpContextAccessor
        .HttpContext.GetTokenAsync("expires_at");
    var expiresAtDateTimeOffset =
        DateTimeOffset.Parse(expiresAtToken, CultureInfo.InvariantCulture);

    if ((expiresAtDateTimeOffset.AddSeconds(-60)).ToUniversalTime() > DateTime.UtcNow)
        return await _httpContextAccessor
            .HttpContext.GetTokenAsync(OpenIdConnectParameterNames.AccessToken);

    var refreshResponse = await GetRefreshResponseFromIDP();

    var updatedTokens = GetUpdatedTokens(refreshResponse);

    var currentAuthenticateResult = await _httpContextAccessor
        .HttpContext
        .AuthenticateAsync(CookieAuthenticationDefaults.AuthenticationScheme);

    currentAuthenticateResult.Properties.StoreTokens(updatedTokens);

    await _httpContextAccessor.HttpContext.SignInAsync(
        CookieAuthenticationDefaults.AuthenticationScheme,
        currentAuthenticateResult.Principal,
        currentAuthenticateResult.Properties);

    return refreshResponse.AccessToken;
}
```

First, we extract the **expires_at** token and convert it to the **DateTimeOffset** value. If this offset time is greater than the current time, we just return the access token we already have. Additionally, we subtract sixty seconds because we want to trigger the refresh logic before the old token expires. But, if this check returns false, which means that the token is about to expire or it has already expired, we get the refresh token response from IDP and store all the updated tokens from the response in the **updatedTokens** list. For these actions, we use two private methods which we will inspect in a minute.

After storing the updated tokens, we call the **AuthenitcateAsync** method to get the authentication result, store the updated tokens in the **Properties** list and use this **Properties** list to sign in again. Finally, we just return the new access token.

Now, let’s look at the **GetRefreshResponseFromIDP** method:

```
private async Task<TokenResponse> GetRefreshResponseFromIDP()
{
    var idpClient = _httpClientFactory.CreateClient("IDPClient");
    var metaDataResponse = await idpClient.GetDiscoveryDocumentAsync();

    var refreshToken = await _httpContextAccessor
        .HttpContext
        .GetTokenAsync(OpenIdConnectParameterNames.RefreshToken);

    var refreshResponse = await idpClient.RequestRefreshTokenAsync(
        new RefreshTokenRequest
        {
            Address = metaDataResponse.TokenEndpoint,
            ClientId = "companyemployeeclient",
            ClientSecret = "CompanyEmployeeClientSecret",
            RefreshToken = refreshToken
        });

    return refreshResponse;
}
```

In this method, we create an HttpClient object at the IDP level. Then, we extract the metadata from the Discovery Document and fetch the refresh token from the **HttpContext**. Right after that, we create a RefreshTokenRequest object and use it to request the refresh token response with the **RequestRefreshTokenAsync** method. This response will contain the new access token, id token, refresh token and expires_at token as well.

That said, we can inspect the **GetUpdatedTokens** method:

```
private List<AuthenticationToken> GetUpdatedTokens(TokenResponse refreshResponse)
{
    var updatedTokens = new List<AuthenticationToken>
    {
        new AuthenticationToken
        {
            Name = OpenIdConnectParameterNames.IdToken,
            Value = refreshResponse.IdentityToken
        },
        new AuthenticationToken
        {
            Name = OpenIdConnectParameterNames.AccessToken,
            Value = refreshResponse.AccessToken
        },
        new AuthenticationToken
        {
            Name = OpenIdConnectParameterNames.RefreshToken,
            Value = refreshResponse.RefreshToken
        },
        new AuthenticationToken
        {
            Name = "expires_at",
            Value = (DateTime.UtcNow + TimeSpan.FromSeconds(refreshResponse.ExpiresIn)).
                ToString("o", CultureInfo.InvariantCulture)
        }
    };

    return updatedTokens;
}
```

Here we take all the updated tokens from the **refreshResponse** object and store them into a single list.

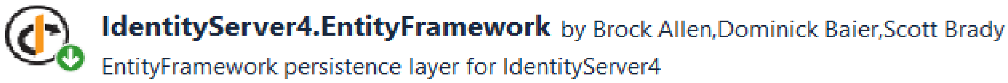
With all these in place, as soon as we request access to the protected API's endpoint, this logic will kick in. If the access token has expired or is about to expire, it will be refreshed. Feel free to wait a couple of minutes and try accessing the Companies action. You will see the data fetched from the API for sure.

9 Migrating IdentityServer4 Configuration to the Database with Entity Framework Core

In all the previous sections of this book, we have been working with the in-memory IDP configuration. But, every time we wanted to change something in that configuration, we had to restart our Identity Server to load the new configuration. In this section, we are going to learn how to migrate the IdentityServer4/Duende configuration to the database using Entity Framework Core (EF Core), so we could persist our configuration across multiple IdentityServer instances.

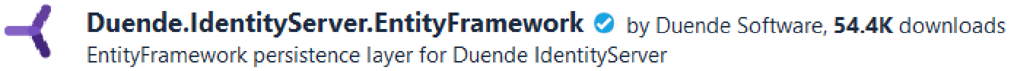
Let's start by adding NuGet packages required for the IdentityServer4 configuration migration process.

The first package we require is **IdentityServer4.EntityFramework**:



IdentityServer4.EntityFramework by Brock Allen,Dominick Baier,Scott Brady
EntityFramework persistence layer for IdentityServer4

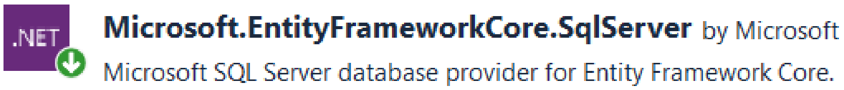
For Duende, we need a package with a different name **Duende.IdentityServer.EntityFramework**:



Duende.IdentityServer.EntityFramework by Duende Software, **54.4K** downloads
EntityFramework persistence layer for Duende IdentityServer

This package implements the required stores and services using two context classes: **ConfigurationDbContext** and **PersistedGrantDbContext**. It uses the first context class for the configuration of clients, resources, and scopes. The second context class is used for temporary operational data like authorization codes and refresh tokens.

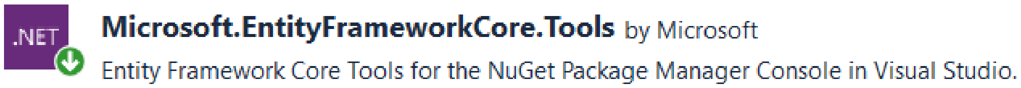
The second library we require is **Microsoft.EntityFrameworkCore.SqlServer**:



Microsoft.EntityFrameworkCore.SqlServer by Microsoft
Microsoft SQL Server database provider for Entity Framework Core.

As the package description states, it is a database provider for the EF Core.

Finally, we require **Microsoft.EntityFrameworkCore.Tools** package to support migrations:



Microsoft.EntityFrameworkCore.Tools by Microsoft
Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.

After these installations, we can move on to the configuration part.

9.1 Preparing Migrations for the IS4 Configuration

First thing's first – let's create the **appsettings.json** file, for Duende it is already created and populated with the Serilog configuration, and modify it:

```
"ConnectionStrings": {
    "sqlConnection": "server=.; database=CompanyEmployeeOAuth; Integrated Security=true;
TrustServerCertificate=true"
}
```

After adding the SQL connection string, we are going to modify the **Startup** class by injecting the **IConfiguration** interface for the IS4 project:

```
public IConfiguration Configuration { get; set; }

public Startup(IWebHostEnvironment environment, IConfiguration configuration)
{
    Environment = environment;
    Configuration = configuration;
}
```

To use the **IConfiguration** interface we have to use **Microsoft.Extensions.Configuration** namespace.

For Duende, we don't have to add the **Configuration** property as we can use it with the **builder** object.

Next, let's modify the **ConfigureServices** method:

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

    var migrationAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

    var builder = services.AddIdentityServer(options =>
    {
            // see https://identityserver4.readthedocs.io/en/latest/topics/resources.html
            options.EmitStaticAudienceClaim = true;
    })
    .AddTestUsers(TestUsers.Users)
    .AddConfigurationStore(opt =>
    {
         opt.ConfigureDbContext = c =>
c.UseSqlServer(Configuration.GetConnectionString("sqlConnection"),
         sql => sql.MigrationsAssembly(migrationAssembly));
    })
    .AddOperationalStore(opt =>
    {
         opt.ConfigureDbContext = o =>
o.UseSqlServer(Configuration.GetConnectionString("sqlConnection"),
         sql => sql.MigrationsAssembly(migrationAssembly));
    });

    builder.AddDeveloperSigningCredential();
}
```

For the Duende project, the code is almost the same:

```
public static WebApplication ConfigureServices(this WebApplicationBuilder builder)
{
    builder.Services.AddRazorPages();

    var migrationAssembly = typeof(Program).GetTypeInfo().Assembly.GetName().Name;

    builder.Services.AddIdentityServer(options =>
    {
        options.EmitStaticAudienceClaim = true;
    })
    .AddTestUsers(TestUsers.Users)
    .AddConfigurationStore(opt =>
    {
            opt.ConfigureDbContext = c =>
c.UseSqlServer(builder.Configuration.GetConnectionString("sqlConnection"),
            sql => sql.MigrationsAssembly(migrationAssembly));
    })
    .AddOperationalStore(opt =>
    {
    opt.ConfigureDbContext = o =>
o.UseSqlServer(builder.Configuration.GetConnectionString("sqlConnection"),
        sql => sql.MigrationsAssembly(migrationAssembly));
    });

    return builder.Build();
}
```

So, we start by extracting the assembly name for our migrations. We need that because we have to inform EF Core that our project will contain the migration code. Additionally, EF Core needs this information because our project is in a different assembly than the one containing the DbContext classes.

After that, we replace the **AddInMemoryClients**, **AddInMemoryIdentityResources, AddInMemoryApiScopes** and **AddInMemoryApiResources** methods with the **AddConfigurationStore** and **AddOperationalStore** methods. Both methods require information about the connection string and migration assembly.

9.2 Creating Migrations

As previously mentioned, we are working with two DbContext classes, and for each of them we have to create a separate migration.

But before we create those migrations, we need to mention that if you use the Duende project, you will get the **HostAbortedException** after each migration is created. It is expected exception since Entity Framework CLI internally starts up IdentityServer for a short time in order to read our database configuration. After it has read the configuration, it shuts IdentityServer down by throwing this exception.

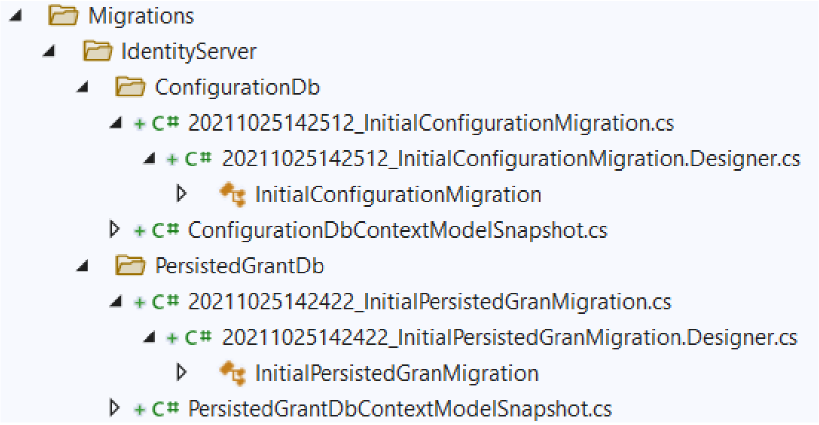
If you don't want to see it, you can simply wrap the catch block in the **Program** class:

```
catch (Exception ex)
{
    if (ex.GetType().Name != "HostAbortedException")
    {
        Log.Fatal(ex, $"Unhandled exception {ex.GetType().Name}");
    }
}
```

Now, we can create our migrations without an exception logged:

```
PM> Add-Migration InitialPersistedGrantMigration -c PersistedGrantDbContext -o
Migrations/IdentityServer/PersistedGrantDb

PM> Add-Migration InitialConfigurationMigration -c ConfigurationDbContext -o
Migrations/IdentityServer/ConfigurationDb
```

As we can see, we are using two flags for our migrations: – c and – o. The – c flag stands for Context and the – o flag stands for OutputDir. So basically, we have created migrations for each context class in a separate folder:



```
◢ Migrations
    ◢ IdentityServer
        ◢ ConfigurationDb
            ◢ + C# 20211025142512_InitialConfigurationMigration.cs
                ◢ + C# 20211025142512_InitialConfigurationMigration.Designer.cs
                    ▷ InitialConfigurationMigration
            ▷ + C# ConfigurationDbContextModelSnapshot.cs
        ◢ PersistedGrantDb
            ◢ + C# 20211025142422_InitialPersistedGranMigration.cs
                ◢ + C# 20211025142422_InitialPersistedGranMigration.Designer.cs
                    ▷ InitialPersistedGranMigration
            ▷ + C# PersistedGrantDbContextModelSnapshot.cs
```

Once we have our migration files, we are going to create a new **InitialSeed** folder with a new class to seed our data:

```
public static class MigrationManager
{
    public static IHost MigrateDatabase(this IHost host)
    {
        using (var scope = host.Services.CreateScope())
        {
            scope.ServiceProvider
                .GetRequiredService<PersistedGrantDbContext>()
                .Database
                .Migrate();

            using (var context = scope.ServiceProvider
                .GetRequiredService<ConfigurationDbContext>())
            {
                try
                {
                    context.Database.Migrate();

                    if (!context.Clients.Any())
                    {
                        foreach (var client in Config.Clients)
                        {
                            context.Clients.Add(client.ToEntity());
                        }
                        context.SaveChanges();
                    }

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

                    if (!context.ApiScopes.Any())
                    {
                        foreach (var apiScope in Config.ApiScopes)
                        {
                            context.ApiScopes.Add(apiScope.ToEntity());
                        }
                            context.SaveChanges();
                    }

                    if (!context.ApiResources.Any())
                    {
                        foreach (var resource in Config.Apis)
                        {
                            context.ApiResources.Add(resource.ToEntity());
                        }
                        context.SaveChanges();
                    }
                }
                catch (Exception ex)
                {
                    //Log errors or do anything you think it's needed
                    throw;
                }
            }
        }

        return host;
    }
}
```

These are the required namespaces:

```
using IdentityServer4.EntityFramework.DbContexts;
using IdentityServer4.EntityFramework.Mappers;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using System;
using System.Linq;
```

Just for the Duende implementation, you will have to replace the two IdentityServer4 namespaces with Duende namespaces:

```
using Duende.IdentityServer.EntityFramework.DbContexts;
using Duende.IdentityServer.EntityFramework.Mappers;
```

Also, for Duende we are not extending the **IHost** interface but the **WebApplication** class:

```
public static WebApplication MigrateDatabase(this WebApplication app)
```

So, rename all the **host** to **app** in your code.

Here, we create a scope and use it to migrate all the tables from the **PersistedGrantDbContext** class. A few lines after that, we create a context for the **ConfigurationDbContext** class and use the **Migrate** method to apply the migration. Then, we go through all the clients, identity resources, and API scopes and resources, add each of them to the context and call the **SaveChanges** method.

Now, all we have to do is to modify **Program.cs** class:

```
try
{
    Log.Information("Starting host...");
    CreateHostBuilder(args).Build().MigrateDatabase().Run();
    return 0;
}
```

For Duende, the code is slightly different:

```
var app = builder
    .ConfigureServices()
    .ConfigurePipeline();

app.MigrateDatabase().Run();
```

After we do as suggested, once the IDP starts, the database will be created with all the tables inside it. You will now find additional tables like `ApiScopes` and `ApiResourceScopes` to support the changes in the newest IS4 version.

Additionally, we can start other projects and confirm that everything is still working as it was before applying these changes.

10 ASP.NET Core Identity Integration with Existing IdentityServer4 Project

Up until now, we have been working with in-memory users placed inside the TestUsers class. But, as we did with the IdentityServer4/Duende configuration, we want to transfer these users to the database as well. Additionally, with IS4/Duende, we can work with Authentication and Authorization, but we can't work with user management, and for that, we are going to integrate the ASP.NET Core Identity library. We are going to create a new database for this purpose and transfer our users to it. Later on, you will see how to use ASP.NET Core Identity features like registration, password reset, etc.

But, before we continue, it is important to mention that there is a template command which creates a project where both IS4 and Identity are integrated. So, if you are starting a new project and you want both integrated, you can use:

```
dotnet new is4aspid -n IdentityServerAspNetIdentity
```

Or for Duende:

```
dotnet new isaspid -n IdentityServerAspNetIdentity
```

As we said, this would create a new project with the basic configuration for the IdentityServer4/Duende and ASP.NET Core Identity.

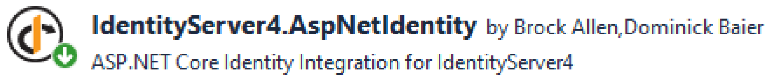
But, while learning, we always prefer the step-by-step approach, as we did with all the content in this book. Furthermore, we have already created an IDP project, so we have to implement the ASP.NET Core Identity on our own. Once we completely understand the process – which is the goal of manual implementation, we can use the template command for our next projects.

10.1 Integrating the ASP.NET Core Identity

First, let’s add a new connection string to the **appsettings.json** file:

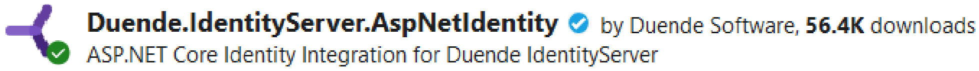
```
"ConnectionStrings": {
    "sqlConnection": "server=.; database=CompanyEmployeeOAuth; Integrated Security=true;
TrustServerCertificate=true",
    "identitySqlConnection": "server=.; database=CompanyEmployeeOAuthIdentity; Integrated
Security=true; TrustServerCertificate=true"
  }
```

Now, we need to install a Nuget package to support the use of Identity in the IDP project:



IdentityServer4.AspNetIdentity by Brock Allen,Dominick Baier
ASP.NET Core Identity Integration for IdentityServer4

For Duende, we need a different package:



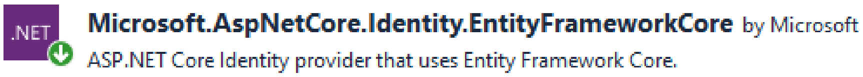
Duende.IdentityServer.AspNetIdentity by Duende Software, 56.4K downloads
ASP.NET Core Identity Integration for Duende IdentityServer

After the installation is completed, we are going to create an **Entities** folder and inside it a **User** class:

```
public class User : IdentityUser
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public string Country { get; set; }
}
```

Our **User** class must inherit from the **IdentityUser** class to accept all the default fields that Identity provides. Additionally, we extend the IdentityUser class with our properties.

Now, let’s install **Microsoft.AspNetCore.Identity.EntityFrameworkCore** library required for our context class:



Microsoft.AspNetCore.Identity.EntityFrameworkCore by Microsoft
ASP.NET Core Identity provider that uses Entity Framework Core.

Then, we can create a context class in the **Entities** folder:

```
public class UserContext : IdentityDbContext<User>
{
    public UserContext(DbContextOptions<UserContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);
    }
}
```

The last step for the integration process is the **Startup** class modification (For Duende it is the **HostingExtensions** class where we have to use **builder.Services** and **builder.Configuration**):

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

    var migrationAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

    services.AddDbContext<UserContext>(options => options
        .UseSqlServer(Configuration.GetConnectionString("identitySqlConnection")));

    services.AddIdentity<User, IdentityRole>()
        .AddEntityFrameworkStores<UserContext>()
        .AddDefaultTokenProviders();

    var builder = services.AddIdentityServer(options =>
    {
        options.EmitStaticAudienceClaim = true;
    })
    .AddTestUsers(TestUsers.Users)
    .AddConfigurationStore(opt =>
    {
        opt.ConfigureDbContext = c =>
c.UseSqlServer(Configuration.GetConnectionString("sqlConnection"),
        sql => sql.MigrationsAssembly(migrationAssembly));
    })
    .AddOperationalStore(opt =>
    {
        opt.ConfigureDbContext = o =>
o.UseSqlServer(Configuration.GetConnectionString("sqlConnection"),
        sql => sql.MigrationsAssembly(migrationAssembly));
    })
    .AddAspNetIdentity<User>();

    builder.AddDeveloperSigningCredential();
}
```

So, first, we register the **UserContext** class into the service collection. Then, by calling the **AddIdentity** method, we register ASP.NET Core Identity with a specific user and role classes. Additionally, we register the Entity Framework Core for Identity and provide a default token provider. The last thing we changed here is the call to the **AddAspNetIdentity** method. This way, we add an integration layer to allow IdentityServer to access user data from the ASP.NET Core Identity user database.

10.2 Initial Migrations

To create a required database and its tables, we have to add a new migration:

```
PM> Add-Migration CreateIdentityTables –Context UserContext
```

As you can see, we have to specify the context class as well, since we already have two context classes related to IS4.

After the file creation, let’s execute this migration with the **Update-Database** command:

```
PM> Update-Database -Context UserContext
```

This should create a database with all the required tables, and, if you inspect the **AspNetUsers** table, you are going to see additional columns (FirstName, LastName, Address and Country).

Since we have all the required tables, we can create default roles required for our users. To do that, we are going to create a **Configuration** folder inside the **Entities** folder. Next, let’s create a new class in the **Configuration** folder:

```
public class RoleConfiguration : IEntityTypeConfiguration<IdentityRole>
{
    public void Configure(EntityTypeBuilder<IdentityRole> builder)
    {
        builder.HasData(new IdentityRole
        {
            Name = "Administrator",
            NormalizedName = "ADMINISTRATOR",
            Id = "c3a0cb55-ddaf-4f2f-8419-f3f937698aa1"
        },
        new IdentityRole
        {
            Name = "Visitor",
            NormalizedName = "VISITOR",
            Id = "6d506b42-9fa0-4ef7-a92a-0b5b0a123665"
        });
    }
}
```

As you can see, we are preparing two roles related to our test users.

For this class, we require a couple of namespaces to be included:

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
```

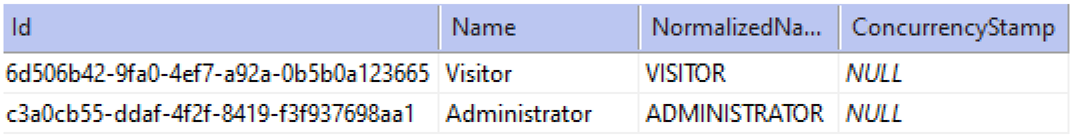
Now, let’s modify the **OnModelCreating** method in the **UserContext** class:

```
protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);

    builder.ApplyConfiguration(new RoleConfiguration());
}
```

With this in place, all we have to do is to create and execute the migration:

```
PM> Add-Migration AddRolesToDb -Context UserContext

PM> Update-Database -Context UserContext
```

At this point, we can find the roles in the **AspNetRoles** table:



Id	Name	NormalizedNa...	ConcurrencyStamp
6d506b42-9fa0-4ef7-a92a-0b5b0a123665	Visitor	VISITOR	*NULL*
c3a0cb55-ddaf-4f2f-8419-f3f937698aa1	Administrator	ADMINISTRATOR	*NULL*

10.3 Seeding the Users

It’s time to transfer our test users to the database. For this action, we can’t create a seed configuration class as we did with the roles because we have additional actions that are related to hashing passwords and adding roles and claims to our users. To be able to do that, we have to use the UserManager class and register Identity as we did in the Startup class.

So, let’s create a new **SeedUserData** class and register Identity in the local service collection:

```
public class SeedUserData
{
    public static void EnsureSeedData(string connectionString)
    {
        var services = new ServiceCollection();
        services.AddLogging();
        services.AddDbContext<UserContext>(options =>
            options.UseSqlServer(connectionString));

        services.AddIdentity<User, IdentityRole>(o => {
            o.Password.RequireDigit = false;
            o.Password.RequireNonAlphanumeric = false;
        }).AddEntityFrameworkStores<UserContext>()
          .AddDefaultTokenProviders();

        using (var serviceProvider = services.BuildServiceProvider())
        {
            using (var scope = serviceProvider
                .GetRequiredService<IServiceScopeFactory>().CreateScope())
            {
                CreateUser(scope, "John", "Doe", "John Doe's Boulevard 323", "USA",
                    "97a3aa4a-7a89-47f3-9814-74497fb92ccb", "JohnPassword",
                    "Administrator", "john@mail.com");

                CreateUser(scope, "Jane", "Doe", "Jane Doe's Avenue 214", "USA",
                    "64aca900-7bc7-4645-b291-38f1b7b5963c", "JanePassword",
                    "Visitor", "jane@mail.com");
            }
        }
    }
}
```

Here, we create a new local service collection and register the logging service and our **UserContext** and Identity as services as well. Identity registration is almost the same as in the Startup class. The only difference is that we modify the Password identity options (we need this to support current passwords from our test users).

Next, we create a local service provider and with its help, we create a service scope that we need to retrieve the **UserManager** service from the service collection. As soon as we have our scope, we call the **CreateUser** method twice, with data for each of our test users.

Now, let’s take a look at that method:

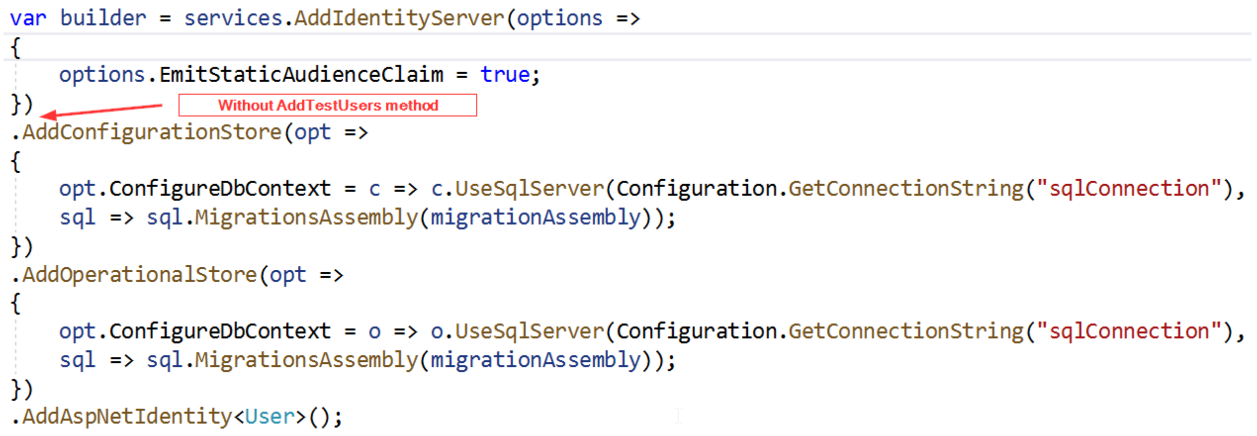
```
private static void CreateUser(IServiceScope scope, string name, string lastName,
  string address, string country, string id, string password, string role, string email)
{
    var userMgr = scope.ServiceProvider.GetRequiredService<UserManager<User>>();
    var user = userMgr.FindByNameAsync(email).Result;
    if (user == null)
    {
        user = new User
        {
            UserName = email, Email = email, FirstName = name,
            LastName = lastName, Address = address, Country = country,
            Id = id
        };
        var result = userMgr.CreateAsync(user, password).Result;
        CheckResult(result);

        result = userMgr.AddToRoleAsync(user, role).Result;
        CheckResult(result);

        result = userMgr.AddClaimsAsync(user, new Claim[]
        {
            new Claim(JwtClaimTypes.GivenName, user.FirstName),
            new Claim(JwtClaimTypes.FamilyName, user.LastName),
            new Claim(JwtClaimTypes.Role, role),
            new Claim(JwtClaimTypes.Address, user.Address),
            new Claim("country", user.Country)
        }).Result;
        CheckResult(result);
    }
}
```

In this method, we use the scope object with the **ServiceProvider** and the **GetRequiredService** method to get the **UserManager** service. Once we have it, we use it to fetch a user by their username. If it doesn’t exist, we create a new user, add a role to that user and add claims. As you can see, for each create operation, we check the result with the **CheckResult** method:

```
private static void CheckResult(IdentityResult result)
{
    if (!result.Succeeded)
    {
        throw new Exception(result.Errors.First().Description);
    }
}
```

As we don’t want to repeat this logic, we extract it in a separate private method.

Now, we have to remove the **AddTestUsers(TestUsers.Users)** method from the **ConfigureServices method**, since we are not going to use these users anymore:



```
var builder = services.AddIdentityServer(options =>
{
    options.EmitStaticAudienceClaim = true;
})
.AddConfigurationStore(opt =>
{
    opt.ConfigureDbContext = c => c.UseSqlServer(Configuration.GetConnectionString("sqlConnection"),
    sql => sql.MigrationsAssembly(migrationAssembly));
})
.AddOperationalStore(opt =>
{
    opt.ConfigureDbContext = o => o.UseSqlServer(Configuration.GetConnectionString("sqlConnection"),
    sql => sql.MigrationsAssembly(migrationAssembly));
})
.AddAspNetIdentity<User>();
```

Other than that, we have to modify the **Program.cs** class a bit:

```
try
{
    Log.Information("Starting host...");
    var builder = CreateHostBuilder(args).Build();

    var config = builder.Services.GetRequiredService<IConfiguration>();
    var connectionString = config.GetConnectionString("identitySqlConnection");
    SeedUserData.EnsureSeedData(connectionString);

    builder.MigrateDatabase().Run();
    return 0;
}
```

So, we create a **builder** object to be able to get the **IConfiguration** service. With this service, we fetch the connection string from the **appsettings.json** file. After we have the connection string, we call the **EnsureSeedData** method to start the migration.

For Duende, the code is slightly different:

```
var app = builder
    .ConfigureServices()
    .ConfigurePipeline();

var config = app.Services.GetRequiredService<IConfiguration>();
var connectionString = config.GetConnectionString("identitySqlConnection");
SeedUserData.EnsureSeedData(connectionString);
```

And that's it. As soon as we start our IDP project, we can check the tables in the **CompanyEmployeeOAuthIdentity** database. All the required tables (AspNetUsers, AspNetUserRoles, AspNetUserClaims) are most definitely populated with valid data.

Of course, after we are done, we can comment out all the seed calls because we don't want to call them every time we start our app.

10.4 Modifying Login and Logout Logic

The last thing we have to modify is the **AccountController** file in the **QuickStart/Account** folder. Since we are now using ASP.NET Core Identity for user management actions, we have to inject required services:

```
private readonly IIdentityServerInteractionService _interaction;
private readonly IClientStore _clientStore;
private readonly IAuthenticationSchemeProvider _schemeProvider;
private readonly IEventService _events;
private readonly UserManager<User> _userManager;
private readonly SignInManager<User> _signInManager;

public AccountController(
    IIdentityServerInteractionService interaction, IClientStore clientStore,
    IAuthenticationSchemeProvider schemeProvider, IEventService events,
    UserManager<User> userManager, SignInManager<User> signInManager)
    {
        _interaction = interaction;
        _clientStore = clientStore;
        _schemeProvider = schemeProvider;
        _events = events;
        _userManager = userManager;
        _signInManager = signInManager;
    }
```

We remove the `TestUserStore` service and inject the `UserManager` and `SignInManager` services.

For Duende, the login page is named Index and we can find it in the `Pages\Account\Login` folder. There, we have to modify the `Index.cshtml.cs` file:

```
private readonly UserManager<User> _userManager;
private readonly SignInManager<User> _signInManager;

...

public Index(
    IIdentityServerInteractionService interaction,
    IAuthenticationSchemeProvider schemeProvider,
    IIdentityProviderStore identityProviderStore, IEventService events,
    UserManager<User> userManager, SignInManager<User> signInManager)
{
    ...
    _userManager = userManager;
    _signInManager = signInManager;
}
```

After that, we have to modify the HttpPost `Login` method by replacing the code inside the model validation check. The same thing we have to do for Duende, just in the `OnPost` method where we have to use `Input` instead of `model`, `LoginOptions` instead of `AccountOptions`, and pass just a second argument (Input.ReturnUrl) to the `LoadingPage` method:

```
 if (ModelState.IsValid)
{
     var result = await _signInManager.PasswordSignInAsync(model.Username,
model.Password, model.RememberLogin, lockoutOnFailure: true);
     if (result.Succeeded)
     {
         var user = await _userManager.FindByNameAsync(model.Username);
         await _events.RaiseAsync(new UserLoginSuccessEvent(user.UserName, user.Id,
user.UserName, clientId: context?.Client.ClientId));

         if (context != null)
         {
             if (context.IsNativeClient())
             {
                 return this.LoadingPage("Redirect", model.ReturnUrl);
             }

             return Redirect(model.ReturnUrl);
         }

         if (Url.IsLocalUrl(model.ReturnUrl))
         {
             return Redirect(model.ReturnUrl);
         }
         else if (string.IsNullOrEmpty(model.ReturnUrl))
         {
             return Redirect("~/");
         }
         else
         {
             throw new Exception("invalid return URL");
         }
     }

     await _events.RaiseAsync(new UserLoginFailureEvent(model.Username, "invalid
credentials", clientId:context?.Client.ClientId));
     ModelState.AddModelError(string.Empty,
AccountOptions.InvalidCredentialsErrorMessage);
}
```

The main difference with this approach is the use of the **_userManager** and **_signInManager** objects for authentication actions.

Finally, we have to modify the HttpPost **Logout** method by replacing:

```
// delete local authentication cookie
await HttpContext.SignOutAsync();
```

with this one:

```
await _signInManager.SignOutAsync();
```

For Duende, we have to inject the **SignInManager<User>** service, and apply the same change to the **OnPost** method in the **Pages\Account\Logout\Index.cshtml.cs** file.

And that’s all we have to do.

Now, we can start all three applications and log in with john@mail.com or jane@mail.com username. We are going to be navigated to the consent screen and we can access the Companies page. Since John is an Administrator, he can see the Privacy page and additional actions on the Companies page.

So, everything is working as it was before, but this time we are using ASP.NET Core Identity for the user management actions. Furthermore, we are now able to work with other actions like user registration, email confirmation, forgot password, etc.

11 User Registration with ASP.NET Core Identity

As we already know, we have our users in a database and we can use their credentials to log in to our application. But, these users were added to the database with the migration process and we don’t have any other way in our application to create new users. Well, in this section, we are going to create the user registration functionality by using ASP.NET Core Identity, thus providing a way for a user to register into our application.

11.1 Actions, View and ViewModel

Let’s start with the **UserRegistrationModel** class that we are going to use to transfer the user data between the view and the action. So, in the **IDP/Entities** folder, we are going to create a new **ViewModels** folder and inside it the mentioned class:

```
public class UserRegistrationModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [Required(ErrorMessage = "Address is required")]
    public string Address { get; set; }

    [Required(ErrorMessage = "Country is required")]
    public string Country { get; set; }

    [Required(ErrorMessage = "Email is required")]
    [EmailAddress]
    public string Email { get; set; }

    [Required(ErrorMessage = "Password is required")]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not 
match.")]
    public string ConfirmPassword { get; set; }
}
```

The Address, Country, Email, and Password properties are required and the ConfirmPassword property must match the Password property. We require the Address and Country because we have a policy-based authorization implemented that relies on these two claims.

Now, let's continue with the required actions. For this, we are going to use the existing **Account** controller in the IDP project:

```
[HttpGet]
public IActionResult Register(string returnUrl)
{
    ViewData["ReturnUrl"] = returnUrl;
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Register(UserRegistrationModel userModel, string returnUrl)
{
    return View();
}
```

Of course, for Duende, inside the **Pages\Account** folder we can remove the Create folder because we are not going to use that one, we will add our own logic. So, let's create a new **Register** folder, and inside that folder a new **Register** razor page:

```
[AllowAnonymous]
public class RegisterModel : PageModel
{
    private readonly UserManager<User> _userManager;

    [BindProperty]
    public UserRegistrationModel Input { get; set; }
    [BindProperty]
    public string ReturnUrl { get; set; }

    public RegisterModel(UserManager<User> userManager)
    {
        _userManager = userManager;
    }

    public IActionResult OnGet(string returnUrl)
    {
        ReturnUrl = returnUrl;
        return Page();
    }

    public async Task<IActionResult> OnPost(string returnUrl)
    {
        return Page();
    }
}
```

So, we create two Register actions (GET and POST). We are going to use the first one to show the view and the second one for the user registration logic.

That said, let’s create a IS4 view for the GET Register action:

```
@model CompanyEmployees.IDP.Entities.ViewModels.UserRegistrationModel

<h2>Register</h2>

<h4>UserRegistrationModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Register" asp-route-returnUrl="@ViewData["ReturnUrl"]" >
            <partial name="_ValidationSummary" />
            <div class="form-group">
                <label asp-for="FirstName" class="control-label"></label>
                <input asp-for="FirstName" class="form-control" />
                <span asp-validation-for="FirstName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="LastName" class="control-label"></label>
                <input asp-for="LastName" class="form-control" />
                <span asp-validation-for="LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Address" class="control-label"></label>
                <input asp-for="Address" class="form-control" />
                <span asp-validation-for="Address" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Country" class="control-label"></label>
                <input asp-for="Country" class="form-control" />
                <span asp-validation-for="Country" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Email" class="control-label"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Password" class="control-label"></label>
                <input asp-for="Password" class="form-control" />
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ConfirmPassword" class="control-label"></label>
                <input asp-for="ConfirmPassword" class="form-control" />
                <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Register" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

Basically, we have all the input fields from our model in this view. Of course, when we click the Create button, it will direct us to the POST Register method with the **UserRegistrationModel** populated.

For Duende, since we use razor pages, this is the difference from the previous code:

```
<form asp-page="/Account/Register/Register">
    <partial name="_ValidationSummary" />
    <input asp-for="ReturnUrl" type="hidden"/>
    <div class="form-group">
        <label asp-for="Input.FirstName" class="control-label"></label>
        <input asp-for="Input.FirstName" class="form-control" />
        <span asp-validation-for="Input.FirstName" class="text-danger"></span>

    </div> …
```

We are using the **asp-page** attribute, a hidden input for binding, and we have to use the **Input** property to access individual properties of our model.

Unfortunately, having this page created isn't enough. We also need a way to navigate to this registration page. So, for that, let's modify the **_Lauout** view in the client application:

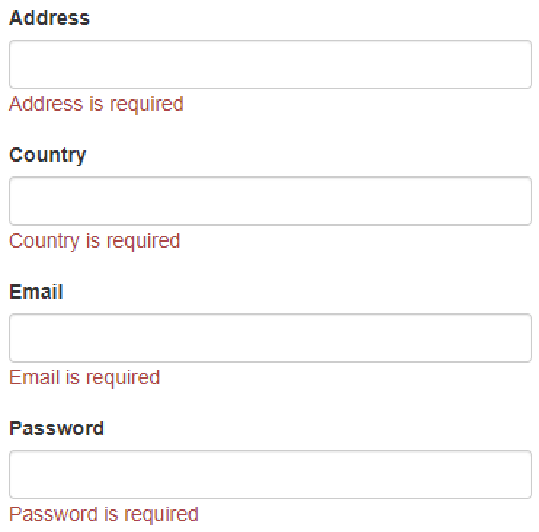
```
@if (!User.Identity.IsAuthenticated)
{
    <section>
        <a class="nav-link text-dark" asp-area="" asp-controller="Auth"
            asp-action="Login">Login</a>
    </section>
    <section>
        <a class="nav-link text-dark" asp-action="Register"
               asp-controller="Account" asp-protocol="https"
               asp-host="localhost:5005"
```

```
                asp-route-returnUrl="https://localhost:5010">Register</a>
    </section>
}
```

For Duende project, as it uses razor pages, we have to create a bit different link:

```
<section>
    <a class="nav-link text-dark" asp-protocol="https"
    asp-host="localhost:5005/Account/Register/Register"
    asp-route-returnUrl="https://localhost:5010">Register</a>
</section>
```

We just create a new menu link that navigates to the Register page at the IDP level. As you can see, we are using additional attributes to navigate to the Register page at the IDP level and provide a query string to return. If we start the IDP and Client applications, click the Register link, and then just click the Register button without entering data, we are going to see the following validation messages:



Address is required

Country is required

Email is required

Password is required

The validation works, so let’s continue towards the action implementation.

11.2 Implementing Registration Action

Before we start with the implementation, let's install the **AutoMapper** library in the IDP project so that we can map the **UserRegistrationModel** to the **User** class:

```
PM> Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection
```

Then, let's register it in the **ConfigureServices** method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAutoMapper(typeof(Startup));

    services.AddControllersWithViews();
```

For Duende it is:

```
builder.Services.AddAutoMapper(typeof(Program));
```

Additionally, for mapping action to work, we have to create a mapping profile. So, let's create a new **MappingProfile** class and modify it:

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        CreateMap<UserRegistrationModel, User>()
            .ForMember(u => u.UserName, opt => opt.MapFrom(x => x.Email));
    }
}
```

Since we are using the **Email** property to populate the **UserName** column, we have to add a rule to this mapping profile.

The final action related to the AutoMapper is to inject it into the **Account** controller or the **Register** page for Duende:

```
private readonly IMapper _mapper;

public AccountController(
    ...
    UserManager<User> userManager, SignInManager<User> signInManager,
    IMapper mapper)
{
    ...
    _userManager = userManager;
    _signInManager = signInManager;
    _mapper = mapper;
}
```

And that's all regarding the AutoMapper.

Now, we can modify the **Register** action:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(UserRegistrationModel userModel string
returnUrl)
{
    if (!ModelState.IsValid)
    {
        return View(userModel);
    }

    var user = _mapper.Map<User>(userModel);

    var result = await _userManager.CreateAsync(user, userModel.Password);
    if (!result.Succeeded)
    {
        foreach (var error in result.Errors)
        {
            ModelState.TryAddModelError(error.Code, error.Description);
        }

        return View(userModel);
    }

    await _userManager.AddToRoleAsync(user, "Visitor");

    await _userManager.AddClaimsAsync(user, new List<Claim>
    {
        new Claim(JwtClaimTypes.GivenName, user.FirstName),
        new Claim(JwtClaimTypes.FamilyName, user.LastName),
        new Claim(JwtClaimTypes.Role, "Visitor"),
        new Claim(JwtClaimTypes.Address, user.Address),
        new Claim("country", user.Country)
     });

     return Redirect(returnUrl);
}
```

For Duende, as we already saw, we have to return the **Page** method and use **Input** as our model:

```
public async Task<IActionResult> OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var user = _mapper.Map<User>(Input);

    var result = await _userManager.CreateAsync(user, Input.Password);
    ...

    return Redirect(ReturnUrl);
}
```

You have probably noticed that the action is asynchronous now. That is because the UserManager's helper methods are asynchronous as well. Inside this action, we check the model validity. If it is invalid, we just return the same view with the invalid model. If it is valid, we map the registration model to the user.

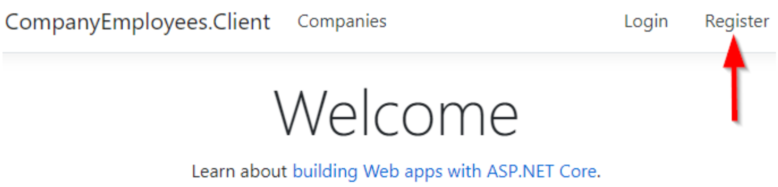
As you can see, we use the **CreateAsync** method to register the user. But, this method does more than that. It hashes a password, performs additional user checks, and returns a result. If the registration was successful, we attach the default role and claims to the user, again, with the UserManager's help and redirect the user to the Home page.

But, if the registration fails, we loop through all the errors and add them to the **ModelState**.

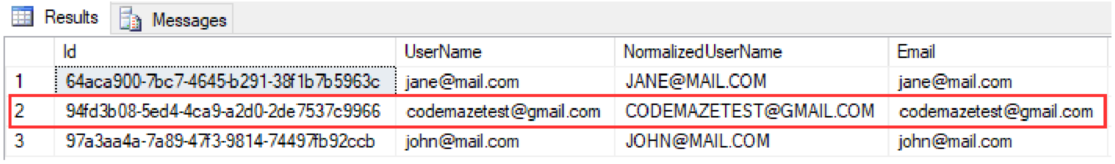
Now, we are ready to test this.

11.3 Testing User Registration

Let's start all three applications and click the Register link:



Then, let's populate all fields and click the Register button. After successful registration, we should be directed to the Home page. If we inspect the database, we can find a new user inside the **AspNetUsers** table:



	Id	UserName	NormalizedUserName	Email
1	64aca900-7bc7-4645-b291-38f1b7b5963c	jane@mail.com	JANE@MAIL.COM	jane@mail.com
2	94fd3b08-5ed4-4ca9-a2d0-2de7537c9966	codemazetest@gmail.com	CODEMAZETEST@GMAIL.COM	codemazetest@gmail.com
3	97a3aa4a-7a89-47f3-9814-74497fb92ccb	john@mail.com	JOHN@MAIL.COM	john@mail.com

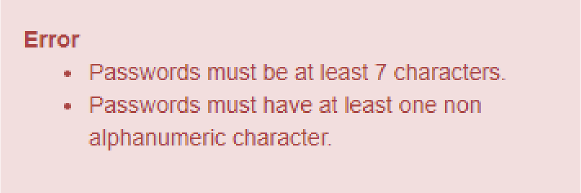
Additionally, you can inspect the **AspNetUserRoles** and **AspNetUserClaims** tables to see the role and the claims attached to this new user. Furthermore, we can use this user to log in and navigate to the Companies page. We are going to see the data but no actions, and that's exactly what we expected.

11.4 Working with Identity Options

By default, Identity implements different configuration rules for the password, email, or any other validation. For example, by default, the password must be at least six characters long, with an uppercase character, non-alphanumeric character and a digit. If we want to override this, we can do it simply by modifying the **AddIdentity** method:

```
services.AddIdentity<User, IdentityRole>(opt =>
{
    opt.Password.RequireDigit = false;
    opt.Password.RequiredLength = 7;
    opt.Password.RequireUppercase = false;
})
```

With this configuration, once we try to register with the "pass" password, an error message will show several violations but none of them will be about an uppercase character or a digit for sure:



Error

- Passwords must be at least 7 characters.
- Passwords must have at least one non alphanumeric character.

12 Reset Password with ASP.NET Core Identity

A common practice in user account management is to provide users with the possibility to change their passwords if they forget them. The password reset process shouldn't involve application administrators because the users themselves should be able to go through the entire process on their own. Usually, the user is provided with the Forgot Password link on the login page and that is exactly what we are going to do.

So, let's explain how the Password Reset process should work in a nutshell.

A user clicks on the Forgot password link and gets redirected to a view with an email input field. After a user populates that field, the application sends a valid link to that email address. The user clicks on the link in the email and gets redirected to the reset password view with a generated token. After the user populates all the fields in the form, the application resets the password and the user gets redirected to the Login (or Home) page.

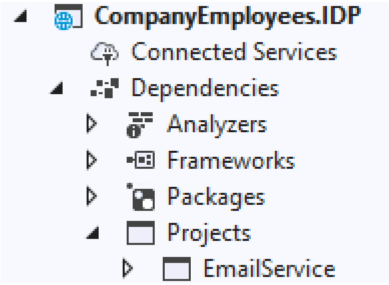
As we can, this entire process requires an email to be sent from our application, so, this is going to be the first thing we are going to deal with.

12.1 Using Email Service in the IdentityServer4 Project

We have already prepared the EmailService project and you can find it in the source code folder called 12-Reset Password. So, the first thing we are going to do is to add this EmailService inside the IDP project as an existing project:

1. Right-click on the solution in the Solution Explorer window
2. Add => Existing Project...
3. Find the EmailService csproj file
4. Select it and click the Open button

After that, we have to add the reference to the EmailService project inside the IDP project:



```
CompanyEmployees.IDP
    Connected Services
    Dependencies
        Analyzers
        Frameworks
        Packages
        Projects
            EmailService
```

Next, we are going to add a configuration for the email service in the **appsettings.json** file:

```
{
  "ConnectionStrings": {
    "sqlConnection": "server=.; database=CompanyEmployeeOAuth; Integrated Security=true;
TrustServerCertificate=true",
    "identitySqlConnection": "server=.; database=CompanyEmployeeOAuthIdentity; Integrated
Security=true; TrustServerCertificate=true"
  },
  "EmailConfiguration": {
    "From": "codemazetest@gmail.com",
    "SmtpServer": "smtp.gmail.com",
    "Port": 465,
    "Username": "codemazetest@gmail.com",
    "Password": "*******"
  }
}
```

Of course, you are going to use your email provider with your credentials.

After that, we are going to extract this data in a singleton service and register our EmailService as a scoped service:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAutoMapper(typeof(Startup));

    var emailConfig = Configuration
        .GetSection("EmailConfiguration")
        .Get<EmailConfiguration>();
    services.AddSingleton(emailConfig);
    services.AddScoped<IEmailSender, EmailSender>();

    services.AddControllersWithViews();
```

For Duende, the code is slightly different:

```
builder.Services.AddAutoMapper(typeof(Program));

var emailConfig = builder.Configuration
.GetSection("EmailConfiguration")
.Get<EmailConfiguration>();
builder.Services.AddSingleton(emailConfig);
builder.Services.AddScoped<IEmailSender, EmailSender>();
```

By registering the email configuration as a service, we are allowing our EmailService application to access this data.

Finally, we have to inject the **IEmailSender** interface in the IS4 **Account** controller:

```
private readonly IMapper _mapper;
private readonly IEmailSender _emailSender;

public AccountController(
    ...
    IMapper mapper, IEmailSender emailSender)
{
    ...
    _mapper = mapper;
    _emailSender = emailSender;
}
```

With this in place, we have prepared everything we need to use the EmailService in the IDP project. Now, we can move on to the Forgot Password functionality implementation.

We will do the same for Duende once we create required pages.

12.2 Implementing Forgot Password Functionality

Let’s start with the **ForgotPasswordModel** class inside the **ViewModels** folder:

```
public class ForgotPasswordModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }
}
```

The **Email** property is the only one we require for the **ForgotPassword** view. Now, let's continue by creating additional actions in the **Account** controller:

```
[HttpGet]
public IActionResult ForgotPassword(string returnUrl)
{
    ViewData["ReturnUrl"] = returnUrl;
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> ForgotPassword(ForgotPasswordModel forgotPasswordModel,
string returnUrl)
{
    return View(forgotPasswordModel);
}

public IActionResult ForgotPasswordConfirmation()
{
    return View();
}
```

For Duende, we are going to create a new `ForgotPassword` folder inside the `Pages\Account` folder, and create two new pages inside: `ForgotPassword` and `ForgotPasswordConfirmation`.

Now, let's modify the class file of the `ForgotPassword` page:

```
[AllowAnonymous]
public class ForgotPasswordModel : PageModel
{
    private readonly UserManager<User> _userManager;
    private readonly IEmailSender _emailSender;

    [BindProperty]
    public Entities.ViewModels.ForgotPasswordModel Input { get; set; }
    [BindProperty]
    public string ReturnUrl { get; set; }

    public ForgotPasswordModel(UserManager<User> userManager,
        IEmailSender sender)
    {
        _userManager = userManager;
        _emailSender = sender;
    }

    public IActionResult OnGet(string returnUrl)
    {
        ReturnUrl = returnUrl;

        return Page();
    }

    public async Task<IActionResult> OnPost()
    {
        return Page();
    }
}
```

For the **ForgotPasswordConfirmation** page, we just need **OnGet** to return the **Page** method:

```
[AllowAnonymous]
public class ForgotPasswordConfirmationModel : PageModel
{
    public IActionResult OnGet()
    {
        return Page();
    }
}
```

This is a familiar setup. The first action/method is just for the view creation, the second one is for the main logic and the last one just returns the confirmation view. You can also notice the returnUrl parameter/property. We need it because, in that parameter, IS4/Duende has all the information required to navigate to the Login screen (redirect_uri, response type, scope, etc.).

Of course, we have to create these views. Let's start with the **ForgotPassword** view:

```
@model CompanyEmployees.IDP.Entities.ViewModels.ForgotPasswordModel

<h2>ForgotPassword</h2>

<h4>ForgotPasswordModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="ForgotPassword" asp-route-returnUrl="@ViewData["ReturnUrl"]">
            <div class="form-group">
                <label asp-for="Email" class="control-label"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Submit" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

And for Duende:

```
@page
@model CompanyEmployees.IDP.Pages.Account.ForgotPassword.ForgotPasswordModel

<h2>ForgotPassword</h2>

<h4>ForgotPasswordModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-page="/Account/ForgotPassword/ForgotPassword">
            <input asp-for="ReturnUrl" type="hidden"/>
            <div class="form-group">
                <label asp-for="Input.Email" class="control-label"></label>
                <input asp-for="Input.Email" class="form-control" />
                <span asp-validation-for="Input.Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Submit" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

After that, let's move on to the **ForgotPasswordConfirmation** view:

```
<h1>ForgotPasswordConfirmation</h1>

<p>
    The link has been sent, please check your email to reset your password.
</p>
```

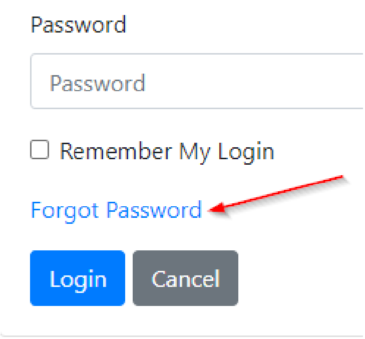
If we want to navigate to the **ForgotPassword** view, we have to click on the Forgot Password link in the **Login** view. So, let's add it there:

```
<div class="form-group">
    <a asp-action="ForgotPassword"
        asp-route-returnUrl="@Model.ReturnUrl">Forgot Password</a>
</div>
<button class="btn btn-primary" name="button" value="login">Login</button>
<button class="btn btn-default" name="button" value="cancel">Cancel</button>
```

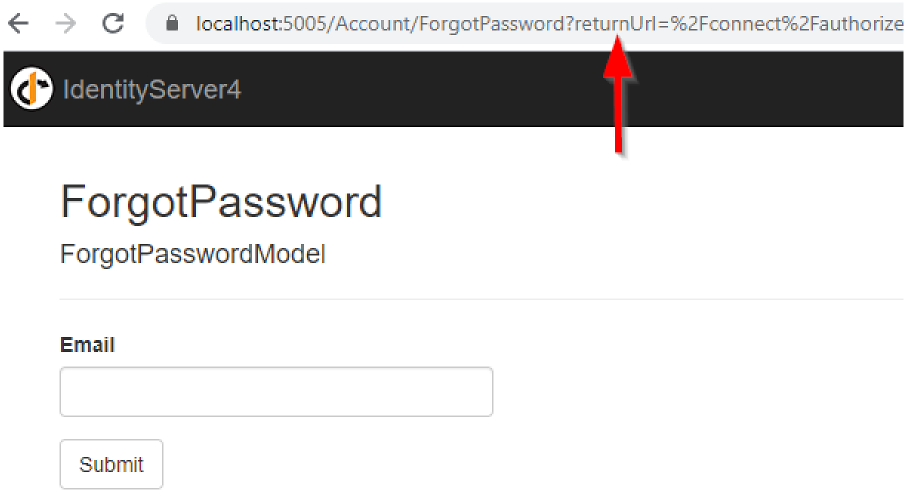
For Duende it is **Account\Login\Index.cshtml** file, with a bit different link:

```
<div class="form-group">
    <a asp-page="/Account/ForgotPassword/ForgotPassword"
    asp-route-returnUrl="@Model.Input.ReturnUrl">Forgot Password</a>
</div>
<button class="btn btn-primary" name="Input.Button" value="login">Login</button>

<button class="btn btn-secondary" name="Input.Button" value="cancel">Cancel</button>
```

As soon as we start our applications and navigate to the Login page, we can see the **Forgot Password** link:



After we click the link, we are going to see our page with the **returnUrl** parameter populated:


localhost:5005/Account/ForgotPassword?returnUrl=%2Fconnect%2Fauthorize

Now, we can modify the POST action:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> ForgotPassword(ForgotPasswordModel forgotPasswordModel,
string returnUrl)
{
    if (!ModelState.IsValid)
        return View(forgotPasswordModel);

    var user = await _userManager.FindByEmailAsync(forgotPasswordModel.Email);
    if (user == null)
        return RedirectToAction(nameof(ForgotPasswordConfirmation));

    var token = await _userManager.GeneratePasswordResetTokenAsync(user);
    var callback = Url.Action(nameof(ResetPassword), "Account", new { token, email =
user.Email, returnUrl }, Request.Scheme);

    var message = new Message(new string[] { user.Email }, "Reset password token",
callback, null);
    await _emailSender.SendEmailAsync(message);

    return RedirectToAction(nameof(ForgotPasswordConfirmation));
}
```

If the model is valid, we fetch the user information from the database by using their email. If the user doesn't exist, we don't show a message that the user with the provided email doesn't exist in the database, but rather just redirect that user to the confirmation page.

This is a good practice for security reasons.

Then, if the user does exist, we generate a token with the **GeneratePasswordResetTokenAsync** method and create a callback link to the action we are going to use for the reset logic. Finally, we send an email to the provided email address and redirect the user to the **ForgotPasswordConfirmation** view.

For Duende, the code in the **OnPost** method is a bit different but the logic is still the same:

```
public async Task<IActionResult> OnPost()
{
    if (!ModelState.IsValid)
        return Page();

    var user = await _userManager.FindByEmailAsync(Input.Email);
    if (user == null)
        return RedirectToPage("/Account/ForgotPassword/ForgotPasswordConfirmation");

    var token = await _userManager.GeneratePasswordResetTokenAsync(user);
    var callback = Url.Page("/Account/ResetPassword/ResetPassword", null, new { token,
email = user.Email, ReturnUrl }, Request.Scheme);

    var message = new Message(new string[] { user.Email }, "Reset password token",
callback, null);
    await _emailSender.SendEmailAsync(message);

    return RedirectToPage("/Account/ForgotPassword/ForgotPasswordConfirmation");

}
```

If we inspect the **ConfigureServices** method, we can see the call to the **AddDefaultTokenProviders** method. With this, we have enabled the token creation action. But there's one more thing left to do. We want our password reset token to be valid for a limited time, for example, 2 hours. So, to do that, we have to configure a token lifespan in the **ConfigureServices** method:

```
builder.AddDeveloperSigningCredential();

services.Configure<DataProtectionTokenProviderOptions>(opt =>
    opt.TokenLifespan = TimeSpan.FromHours(2));
```

For Duende:

```
builder.Services.Configure<DataProtectionTokenProviderOptions>(opt =>
    opt.TokenLifespan = TimeSpan.FromHours(2));

return builder.Build();
```

Excellent. We can proceed to the Reset Password functionality.

12.3 Implementing Reset Password Functionality

Before we start with the **ResetPassword** actions, we have to create the **ResetPasswordModel** class:

```
public class ResetPasswordModel
{
    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
    [DataType(DataType.Password)]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not
match.")]
    public string ConfirmPassword { get; set; }

    public string Email { get; set; }
    public string Token { get; set; }
}
```

Now, let's create the required actions in the **Account** controller:

```
[HttpGet]
public IActionResult ResetPassword(string token, string email, string returnUrl)
{
    ViewData["ReturnUrl"] = returnUrl;
    var model = new ResetPasswordModel { Token = token, Email = email };
    return View(model);
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> ResetPassword(ResetPasswordModel resetPasswordModel,
string returnUrl)
{
    return View();
}

[HttpGet]
public IActionResult ResetPasswordConfirmation(string returnUrl)
{
    ViewData["ReturnUrl"] = returnUrl;
    return View();
}
```

This is a similar setup to the one we had with the **ForgotPassword** actions. The HttpGet **ResetPassword** action will accept a request from the email, extract the token, email, and returnUrl values, and create a view. The HttpPost **ResetPassword** action is here for the main logic. And the **ResetPasswordConfirmation** is just a helper action to create a view for the user to get a confirmation regarding the action.

In Duende project, we are going to create a new **ResetPassword** folder inside the **Account** folder. Then, let's create two new pages: **ResetPassword** and **ResetPasswordConfirmation**.

After that, we can modify the class file of the **ResetPassword** page:

```
[AllowAnonymous]
public class ResetPasswordModel : PageModel
{
    private readonly UserManager<User> _userManager;

    [BindProperty]
    public Entities.ViewModels.ResetPasswordModel Input { get; set; }
    [BindProperty]
    public string ReturnUrl { get; set; }

    public ResetPasswordModel(UserManager<User> userManager)
    {
        _userManager = userManager;
    }

    public IActionResult OnGet(string token, string email, string returnUrl)
    {
        Input = new Entities.ViewModels.ResetPasswordModel { Token = token, Email = email
};
     ReturnUrl= returnUrl;

        return Page();
    }

    public async Task<IActionResult> OnPost()
    {
        return Page();
    }
}
```

Also, we can modify the class file for the second page:

```
[AllowAnonymous]
public class ResetPasswordConfirmationModel : PageModel
{
    [BindProperty]
    public string ReturnUrl { get; set; }

    public IActionResult OnGet(string returnUrl)
    {
        ReturnUrl= returnUrl;

        return Page();
    }
}
```

Now, let’s create our views. First, we are going to create the **ResetPassword** view:

```
@model CompanyEmployees.IDP.Entities.ViewModels.ResetPasswordModel
```

```
<h2>ResetPassword</h2>

<div class="row">
    <div class="col-md-4">
        <form asp-action="ResetPassword" asp-route-returnUrl="@ViewData["ReturnUrl"]">
            <partial name="_ValidationSummary" />
            <div class="form-group">
                <label asp-for="Password" class="control-label"></label>
                <input asp-for="Password" class="form-control" />
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ConfirmPassword" class="control-label"></label>
                <input asp-for="ConfirmPassword" class="form-control" />
                <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
            </div>
            <input type="hidden" asp-for="Email" class="form-control" />
            <input type="hidden" asp-for="Token" class="form-control" />
            <div class="form-group">
                <input type="submit" value="Reset" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

Pay attention that **Email** and **Token** are hidden inputs and we already have these values.

For Duende the code is quite similar:

```
@page
@model CompanyEmployees.IDP.Pages.Account.ResetPassword.ResetPasswordModel

<h2>ResetPassword</h2>

<div class="row">
    <div class="col-md-4">
        <form asp-page="/Account/ResetPassword/ResetPassword">
            <partial name="_ValidationSummary" />
            <input asp-for="ReturnUrl" type="hidden" />
            <div class="form-group">
                <label asp-for="Input.Password" class="control-label"></label>
                <input asp-for="Input.Password" class="form-control" />
                <span asp-validation-for="Input.Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.ConfirmPassword" class="control-label"></label>
                <input asp-for="Input.ConfirmPassword" class="form-control" />
                <span asp-validation-for="Input.ConfirmPassword" class="text-
danger"></span>
            </div>
            <input type="hidden" asp-for="Input.Email" class="form-control" />
            <input type="hidden" asp-for="Input.Token" class="form-control" />
            <div class="form-group">
                <input type="submit" value="Reset" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

After this, let’s create a view for the confirmation:

```
<h2>ResetPasswordConfirmation</h2>

<p>
    Your password has been reset. Please
        <a asp-action="Login" asp-route-returnUrl="@ViewData["returnUrl"]">click here to
log in</a>.
</p>
```

And the code for Duende:

```
@page
@model CompanyEmployees.IDP.Pages.Account.ResetPassword.ResetPasswordConfirmationModel

<h2>ResetPasswordConfirmation</h2>

<p>
    Your password has been reset. Please
    <a asp-page="/Account/Login/Index" asp-route-returnUrl="@Model.ReturnUrl">click here
to log in</a>.
</p>
```

Excellent.

Now, we can modify the POST action:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> ResetPassword(ResetPasswordModel resetPasswordModel,
string returnUrl)
{
    if (!ModelState.IsValid)
        return View(resetPasswordModel);

    var user = await _userManager.FindByEmailAsync(resetPasswordModel.Email);
    if (user == null)
        RedirectToAction(nameof(ResetPasswordConfirmation), new { returnUrl });

    var resetPassResult = await _userManager.ResetPasswordAsync(user,
resetPasswordModel.Token, resetPasswordModel.Password);
    if (!resetPassResult.Succeeded)
    {
       foreach (var error in resetPassResult.Errors)
       {
           ModelState.TryAddModelError(error.Code, error.Description);
       }

       return View();
    }

    return RedirectToAction(nameof(ResetPasswordConfirmation), new { returnUrl });
}
```

The first two actions are the same as in the **ForgotPassword** action. We check the model validity and whether the user exists in the database. After that, we execute the password reset action using the **ResetPasswordAsync** method. If the action fails, we add errors to the model state and return a view. Otherwise, we just redirect the user to the confirmation page.

Of course, for Duende, we are going to modify the **OnPost** method:

```
public async Task<IActionResult> OnPost()
{
    if (!ModelState.IsValid)
        return Page();

    var user = await _userManager.FindByEmailAsync(Input.Email);
    if (user == null)
        RedirectToPage("/Account/ResetPassword/ResetPasswordConfirmation", new {
ReturnUrl });

    var resetPassResult = await _userManager.ResetPasswordAsync(user, Input.Token,
Input.Password);
    if (!resetPassResult.Succeeded)
    {
        foreach (var error in resetPassResult.Errors)
        {
            ModelState.TryAddModelError(error.Code, error.Description);
        }

        return Page();
    }

    return RedirectToPage("/Account/ResetPassword/ResetPasswordConfirmation", new {
ReturnUrl }); ;
}
```

12.4 Testing the Solution

Before we begin, it is important to mention that From May 30, 2022, Google no longer supports the use of third-party apps or devices which ask you to sign in to your Google Account using only your username and password. So, we have to use a different solution for our application. To do this, we need to enable 2-step verification for our Gmail account first and then we can use the App Password feature to overcome this issue.

So, to enable 2-step verification, we have to:

- Navigate to our Google Account – the account you use to send the emails (https://myaccount.google.com/)
- In the menu on the left, we should select Security
- Then under the "Signing in to Google" section, we can see that 2-Step Verification is off – so we have to click on it
- Click Get Started, provide your password, and confirm the code by providing a mobile number
- If everything goes well, you should see the Turn On option, so just click on it

At this point, we have enabled our 2-Step verification and we can return to the Security page. There, under the same "Signing in to Google" section, we can find the App passwords option set to None.

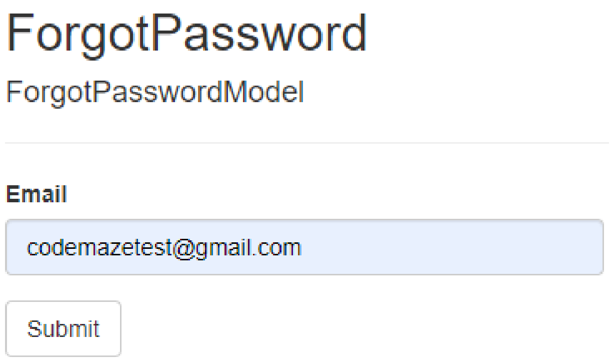
So, we have to:

- Click on it and provide a password
- Click the Select app menu and choose the Other (Custom Name) option
- Now, all we have to do is to provide any name we like for our app and click the Generate button

This will generate a new password for us, which we should use in our `appsettings.json` file instead of our personal password.

With this in place, we are enabled again to send emails with our third-party apps.

Now, let's start our applications, navigate to the **Login** page and click the **Forgot Password** link. In the Email field, we are going to enter our user's email address:



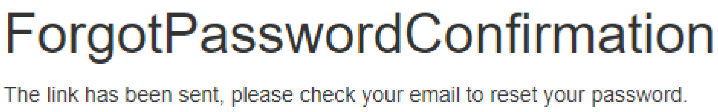
ForgotPassword

ForgotPasswordModel

Email

codemazetest@gmail.com

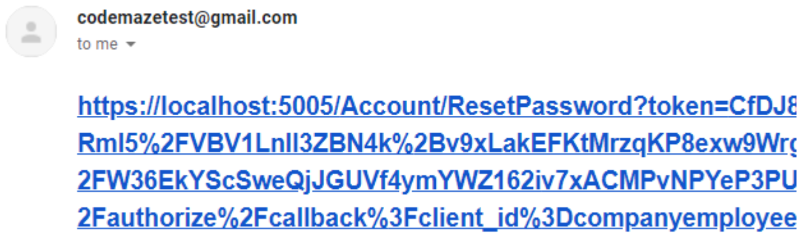
After we click the **Submit** button, the following message is displayed:



ForgotPasswordConfirmation

The link has been sent, please check your email to reset your password.

Now, let's open the email we have received and click the provided link:

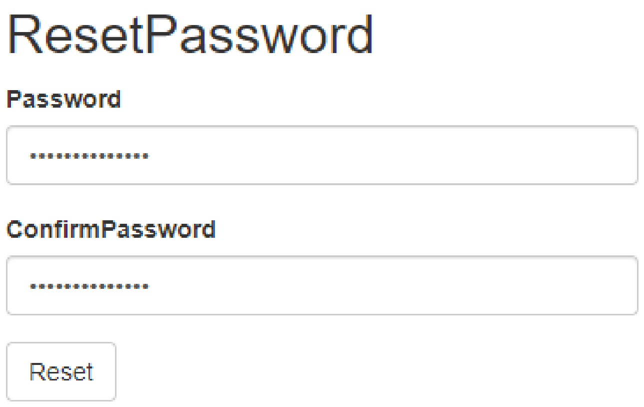


codemazetest@gmail.com

to me

https://localhost:5005/Account/ResetPassword?token=CfDJ8
Rml5%2FVBV1LnlI3ZBN4k%2Bv9xLakEFKtMrzqKP8exw9Wr
2FW36EkYScSweQjJGUVf4ymYWZ162iv7xACMPvNPYeP3PU
2Fauthorize%2Fcallback%3Fclient_id%3Dcompanyemployee

This should navigate us to the page where we need to enter a new password and confirm it:



ResetPassword

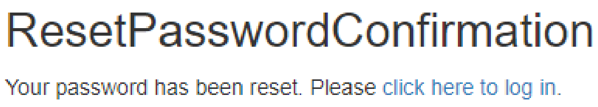
Password

••••••••••••••

ConfirmPassword

••••••••••••••

After we click the **Reset** button, we are going to be redirected to the confirmation page:



ResetPasswordConfirmation

Your password has been reset. Please click here to log in.

Finally, we can click the provided link that will navigate us to the **Login** page and enter our new credentials. After allowing access to the consent page, we are going to be redirected to the **Home** page.

And that’s it. Nicely done.

We have our reset password functionality up and running.

13 EMAIL CONFIRMATION

Email Confirmation is quite an important part of the user registration process. It allows us to verify the registered user is indeed the owner of the provided email. But why is this so important?

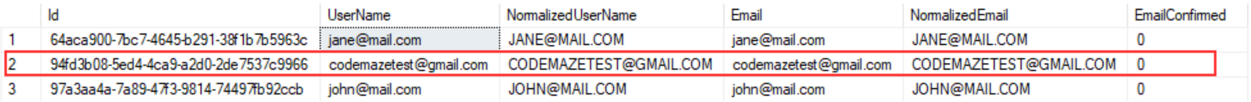
Well, let's imagine the following scenario – we have two users with similar email addresses who want to register in our application. Michael registers first with michel@mail.com instead of michael@mail.com which is his real address. Without an email confirmation, this registration will execute successfully. Now, Michel comes to the registration page and tries to register with his email michel@mail.com. Our application will return an error that the user with that email is already registered. So, thinking that he already has an account, he just resets the password and successfully logs in to the application.

We can see where this could lead and what problems it could cause.

So, let's see how to implement an email confirmation feature in our application to avoid situations such as the one we just described.

13.1 Preparing Email Confirmation Implementation

If we inspect our **codemazetest** user in the database, we can see the email is not confirmed:



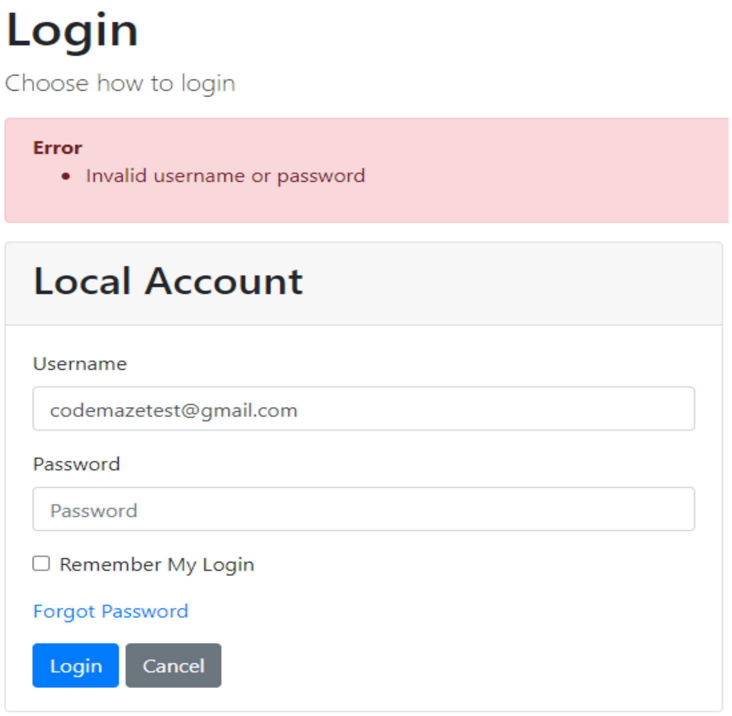
	Id	UserName	NormalizedUserName	Email	NormalizedEmail	EmailConfirmed
1	64aca900-7bc7-4645-b291-38f1b7b5963c	jane@mail.com	JANE@MAIL.COM	jane@mail.com	JANE@MAIL.COM	0
2	94fd3b08-5ed4-4ca9-a2d0-2de7537c9966	codemazetest@gmail.com	CODEMAZETEST@GMAIL.COM	codemazetest@gmail.com	CODEMAZETEST@GMAIL.COM	0
3	97a3aa4a-7a89-47f3-9814-74497fb92ccb	john@mail.com	JOHN@MAIL.COM	john@mail.com	JOHN@MAIL.COM	0

So, even though we didn't confirm our email, we were able to register. Now, we are going to change that by modifying the Identity configuration in the `ConfigureServices` method:

```
services.AddIdentity<User, IdentityRole>(opt =>
{
    opt.Password.RequireDigit = false;
    opt.Password.RequiredLength = 7;
    opt.Password.RequireUppercase = false;
    opt.User.RequireUniqueEmail = true;
    opt.SignIn.RequireConfirmedEmail = true;
})
```

With the **SignIn.RequireConfirmedEmail** we disable successful login if the email isn’t confirmed. The other change is not related to this functionality, we just want to disable using the same email addresses for multiple users.

If we try to log in now, we are going to get this error:



Login

Choose how to login

Error
- Invalid username or password

Local Account

Username: codemazetest@gmail.com

Password: Password

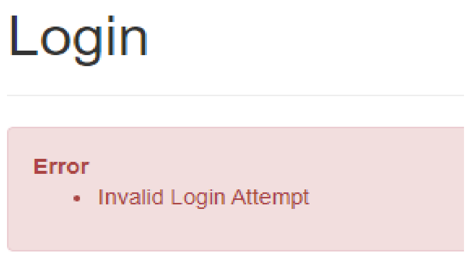
But, we know this is not the case. So, let’s make some modifications to the **AccountOptions** class in the **Account** folder:

```
public static string InvalidLoginAttempt = "Invalid Login Attempt";
```

For Duende it is the **LoginOptions** class inside **Account\Login**

We just modified the name and the value of the **InvalidCredentialsErrorMessage** field. If you renamed this field by pressing F2, as we did, Visual Studio will rename it all over the project which is exactly what we want, since it was used just in the **Login** method in the **Account** controller. If you renamed it manually, make sure to rename the same field in the Login action as well.

Now, we can try to log in again with valid credentials:



Login

Error

- Invalid Login Attempt

As expected, we can see our new error message.

13.2 Implementing Email Confirmation

To Implement Email Confirmation in our project, we have to modify the POST **Register** action and add a few new actions.

So, let’s start with the modification first:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(UserRegistrationModel userModel, string
returnUrl)
{
    ...

    await SendEmailConfirmationLink(user, returnUrl);

    return Redirect(nameof(SuccessRegistration));
}
```

Right after the Claims definition, we call the private **SendEmailConfirmationLink** method. After that, we redirect the user to the **SuccessRegistration** action.

For Duende, we have to do the same for the **OnPost** method in the **Register** page:

```
public async Task<IActionResult> OnPost()
{
    ...

    await SendEmailConfirmationLink(user, ReturnUrl);

    return RedirectToPage("/Account/Register/SuccessRegistration");
}
```

Now, let’s inspect the **SendEmailConfirmationLink** method:

```
private async Task SendEmailConfirmationLink(User user, string returnUrl)
{
    var token = await _userManager.GenerateEmailConfirmationTokenAsync(user);
    var confirmationLink = Url.Action(nameof(ConfirmEmail), "Account",
        new { token, email = user.Email, returnUrl }, Request.Scheme);

    var message = new Message(new string[] { user.Email },
        "Confirmation email link", confirmationLink, null);

    await _emailSender.SendEmailAsync(message);
}
```

As in the **ForgotPassword** action, we create a token but with a different helper method: **GenerateEmailConfirmationTokenAsync(TUser)**. Then, we create a confirmation link and send a message to the user.

For Duende, we have to inject the email sender service first in the **Register** page class:

```
...
private readonly IEmailSender _emailSender;

...

public RegisterModel(UserManager<User> userManager, IMapper mapper, IEmailSender
emailSender)
{
    _userManager = userManager;
    _mapper = mapper;
    _emailSender = emailSender;
}
```

And then, we can add the same private method with a few changes:

```
private async Task SendEmailConfirmationLink(User user)
{
    var token = await _userManager.GenerateEmailConfirmationTokenAsync(user);
    var confirmationLink = Url.Page("/Account/Register/ConfirmEmail", null,
        new { token, email = user.Email, ReturnUrl }, Request.Scheme);

    var message = new Message(new string[] { user.Email },
        "Confirmation email link", confirmationLink, null);

    await _emailSender.SendEmailAsync(message);
}
```

Next, for IS4, we have to create the missing actions: **ConfirmEmail** and **SuccessRegistration**:

```
[HttpGet]
public async Task<IActionResult> ConfirmEmail(string token, string email, string
returnUrl)
{
    ViewData["ReturnUrl"] = returnUrl;

    var user = await _userManager.FindByEmailAsync(email);
    if (user == null)
        return RedirectToAction(nameof(Error), new { returnUrl });

    var result = await _userManager.ConfirmEmailAsync(user, token);
    if (result.Succeeded)
        return View(nameof(ConfirmEmail));
    else
        return RedirectToAction(nameof(Error), new { returnUrl });
}

[HttpGet]
public IActionResult SuccessRegistration()
{
    return View();
}
```

In the **ConfirmEmail** action, we check if the user exists in the database by searching the database for their email address. If the user doesn't exist, we return the **Error** action. Otherwise, we use the **ConfirmEmailAsync** method to check if the email was confirmed. Depending on the result, we are going to return either the **ConfirmEmail** view or redirect to the **Error** action. The **SuccessRegistration** action was just used to show the view.

Of course for Duende, we have to create required pages in the `Register` folder. First the `ConfirmEmail` page:

```
[AllowAnonymous]
public class ConfirmEmailModel : PageModel
{
    private readonly UserManager<User> _userManager;

    public string ReturnUrl { get; set; }

    public ConfirmEmailModel(UserManager<User> userManager)
    {
        _userManager = userManager;
    }

    public async Task<IActionResult> OnGet(string token, string email, string returnUrl)
    {
        ReturnUrl = returnUrl;

        var user = await _userManager.FindByEmailAsync(email);
        if (user == null)
            return RedirectToPage("/Account/Error", new { returnUrl });

        var result = await _userManager.ConfirmEmailAsync(user, token);
        if (result.Succeeded)
            return Page();
        else
            return RedirectToPage("/Account/Error", new { returnUrl });
    }
}
```

And then the `SuccessRegistration` page:

```
[AllowAnonymous]
public class SuccessRegistrationModel : PageModel
{
    public IActionResult OnGet()
    {
        return Page();
    }
}
```

Next, we have to create the `ConfirmEmail` view:

```
<h2>ConfirmEmail</h2>

<p>
    Thank you for confirming your email. Please
    <a href="@ViewData["ReturnUrl"]">click here to navigate to the Home page </a>.
</p>
```

This is a bit different for Duende:

```
@page
@model CompanyEmployees.IDP.Pages.Account.Register.ConfirmEmailModel

<h2>ConfirmEmail</h2>

<p>
    Thank you for confirming your email. Please
    <a href="@Model.ReturnUrl">click here to navigate to the Home page </a>.
</p>
```

After that, let’s create the `SuccessRegistration` view:

```
<h2>SuccessRegistration</h2>

<p>
    Please check your email for the verification action.
</p>
```

Finally, we just need to create the `Error` action:

```
[HttpGet]
public IActionResult Error(string returnUrl)
{
    ViewData["ReturnUrl"] = returnUrl;

    return View();
}
```

And its view:

```
<h2 class="text-danger">Error.</h2>
<h3 class="text-danger">An error occurred while processing your request.</h3>
<a href="@ViewData["ReturnUrl"]">click here navigate to the Home page</a>.
```

For Duende, we are going to create a new `Error` page in the root of the `Account` folder:

```
[AllowAnonymous]
public class ErrorModel : PageModel
{
    public string ReturnUrl { get; set; }

    public IActionResult OnGet(string returnUrl)
    {
        ReturnUrl = returnUrl;

        return Page();
    }
}
```

And the page view as well:

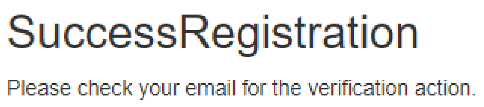
```
@page
@model CompanyEmployees.IDP.Pages.Account.ErrorModel

<h2 class="text-danger">Error.</h2>
<h3 class="text-danger">An error occurred while processing your request.</h3>

<a href="@Model.ReturnUrl">click here navigate to the Home page</a>.
```

That’s it. Now, let’s test this.

13.3 Testing Email Confirmation

You can register a new user with a valid email address if you want, but we are going to register the same one, and for that, we have to remove it from the AspNetUsers table, their related role from the AspNetUserRoles table, and their claims from the AspNetUserClaims table. It is enough to remove the user from the AspNetUsers table and SQL Management Studio will resolve the rest for you.

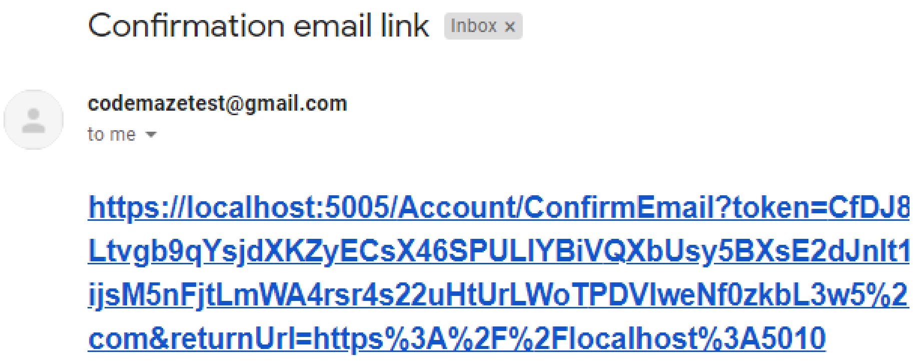
Now, let’s start our applications and navigate to the register view. After we populate all the fields and click the Register button, we are going to be redirected to the **SuccessRegistration** page:



SuccessRegistration

Please check your email for the verification action.

Let’s check our mailbox and open the email we received:

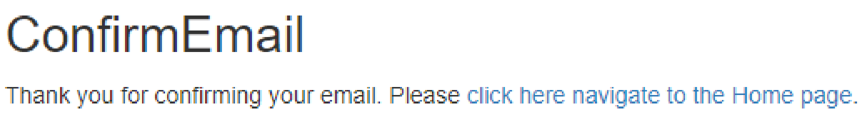


Confirmation email link

codemazetest@gmail.com

https://localhost:5005/Account/ConfirmEmail?token=CfDJ8
Ltvgb9qYsjdXKZyECsX46SPULIYBiVQXbUsy5BXsE2dJnlt1
ijsM5nFjtLmWA4rsr4s22uHtUrLWoTPDVlweNf0zkbL3w5%2
com&returnUrl=https%3A%2F%2Flocalhost%3A5010

Next, let’s click the provided link:

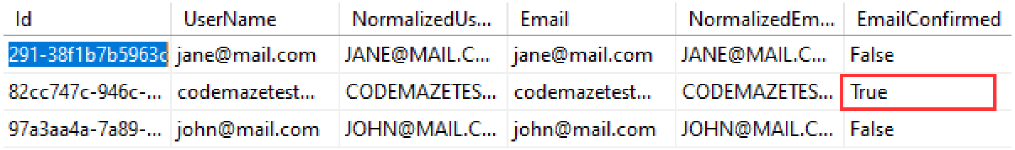


ConfirmEmail

Thank you for confirming your email. Please click here navigate to the Home page.

As you can see, we are navigated to the **ConfirmEmail** view, which means that we have successfully confirmed our email. You can click the link on the page that will navigate you to the Home page and then log in with a newly created user.

If we inspect the database, we are going to see our email is confirmed:



Id	UserName	NormalizedUs...	Email	NormalizedEm...	EmailConfirmed
291-38f1b7b5963d	jane@mail.com	JANE@MAIL.C...	jane@mail.com	JANE@MAIL.C...	False
82cc747c-946c-...	codemazetest...	CODEMAZETES...	codemazetest...	CODEMAZETES...	True
97a3aa4a-7a89-...	john@mail.com	JOHN@MAIL.C...	john@mail.com	JOHN@MAIL.C...	False

Excellent.

13.4 Modifying Lifespan of the Email Token

Now, if we look at the **ConfigureServices** method, we are going to see that this token lasts for two hours, the same as the reset password token. That’s because the reset and confirmation functionalities both use the same data protection token provider with the same instance of the **DataProtectionTokenProviderOptions** class.

But, we don't want our email token to last two hours – usually, it should last longer. For the reset password functionality, a short period is quite ok, but for the email confirmation, it isn't. A user could, for example, easily get distracted and come back to confirm their email address after a day. Thus, we have to increase the lifespan of this type of token.

To do that, we have to create a custom token provider.

Let's create the **CustomTokenProviders** folder with the **EmailConfirmationTokenProvider** class in it:

```
namespace CompanyEmployees.IDP.CustomTokenProviders
{
    public class EmailConfirmationTokenProvider<TUser> :
DataProtectorTokenProvider<TUser> where TUser : class
    {
        public EmailConfirmationTokenProvider(IDataProtectionProvider
dataProtectionProvider,
            IOptions<EmailConfirmationTokenProviderOptions> options,
            ILogger<DataProtectorTokenProvider<TUser>> logger)
            : base(dataProtectionProvider, options, logger)
        {
        }
    }

    public class EmailConfirmationTokenProviderOptions :
DataProtectionTokenProviderOptions
    {
    }
}
```

And that's all we have to do – create a class that implements the **DataProtectionTokenProvider** and override its constructor with a custom token provider options class.

Though, we do need additional namespaces to be included:

```
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
```

Now, we can modify the **ConfigureServices** method:

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddIdentity<User, IdentityRole>(opt =>
    {
        ...
        opt.SignIn.RequireConfirmedEmail = true;
        opt.Tokens.EmailConfirmationTokenProvider = "emailconfirmation";
    })
    .AddEntityFrameworkStores<UserContext>()
    .AddDefaultTokenProviders()
    .AddTokenProvider<EmailConfirmationTokenProvider<User>>("emailconfirmation");
    ...

    services.Configure<DataProtectionTokenProviderOptions>(opt =>
        opt.TokenLifespan = TimeSpan.FromHours(2));

    services.Configure<EmailConfirmationTokenProviderOptions>(opt =>
        opt.TokenLifespan = TimeSpan.FromDays(3));
}
```

In the **`AddIdentity`** method, we state that we want our **`EmailConfirmationTokenProvider`** to use the provider with the name – „emailconfirmation‟. Then, we register our custom token provider with the **`AddTokenProvider<T>`** method and, finally, set its lifespan to three days.

Of course, for Duende, we just have to use the **`builder`** object to access the **`Services`** property.

And that's all it takes. Our user has more time to confirm the provided email address.

14 User Lockout

The user lockout feature is the way to improve the application's security by locking out a user who enters the password incorrectly several times. This technique can help us protect the application against brute force attacks, where an attacker repeatedly tries to guess the password.

That said, let's see how we can implement this functionality.

14.1 Configuration

The default configuration for the lockout functionality is already in place, but, if we want, we can apply our configuration. To do that, we have to modify the **AddIdentity** method in the **ConfigureService** method:

```
services.AddIdentity<User, IdentityRole>(opt =>
{
    ...
    opt.Tokens.EmailConfirmationTokenProvider = "emailconfirmation";
    opt.Lockout.AllowedForNewUsers = true;
    opt.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(2);
    opt.Lockout.MaxFailedAccessAttempts = 3;
})
```

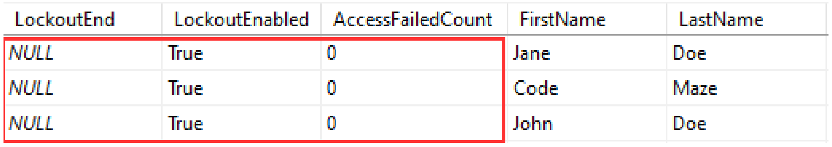
The user lockout feature is enabled by default, but, just as an example, let's explicitly set the **AllowedForNewUsers** property to true. Additionally, let's set the lockout period to two minutes (default is five) and maximum failed login attempts to three (default is five). Of course, the period is set to two minutes just for the sake of this example, that value should be a bit higher in a production environment.

14.2 Lockout Implementation

Before we start, we are going to inspect the **PasswordSignInAsync** method:

```
var result = await _signInManager.PasswordSignInAsync(model.Username, model.Password,
model.RememberLogin, lockoutOnFailure: true);
```

We use the last parameter from this method to enable or disable the lockout feature. By setting the **lockoutOnFailure** parameter to true, we enable the lockout functionality, thus enabling modification of the **AccessFailedCount** and **LockoutEnd** columns in the **AspNetUsers** table:



LockoutEnd	LockoutEnabled	AccessFailedCount	FirstName	LastName
NULL	True	0	Jane	Doe
NULL	True	0	Code	Maze
NULL	True	0	John	Doe

The **AccessFailedCount** column will increase for every failed login attempt and reset once the account is locked out. The **LockoutEnd** column represents the period until this account is locked out.

That said, let's modify the HttpPost **Login** action:

```
if (ModelState.IsValid)
{
    var result = await _signInManager.PasswordSignInAsync ...
    if (result.Succeeded)
    {
        ...
    }
    if (result.IsLockedOut)
    {
        await HandleLockout(model.Username, model.ReturnUrl);
    }
    else
    {
        await _events.RaiseAsync(...);
        ModelState.AddModelError(...);
    }
}
```

The same we have to do in Duende project in `Login\Index.cshtml.cs` file. Just instead of the `model` variable, we have to use the `Input` property.

So, if the sign-in was not successful, we check if the account is locked out. If it is locked out, we call the **HandleLockout** method and just return a Login view/page.

That said, let’s inspect the **HandleLockout** method:

```
private async Task HandleLockout(string email, string returnUrl)
{
    var user = await _userManager.FindByEmailAsync(email);
    var forgotPassLink = Url.Action(nameof(ForgotPassword), "Account",
        new { returnUrl }, Request.Scheme);
    var content = string.Format(@"Your account is locked out,
        to reset your password, please click this link: {0}", forgotPassLink);

    var message = new Message(new string[] { user.Email },
        "Locked out account information", content, null);
    await _emailSender.SendEmailAsync(message);

    ModelState.AddModelError("", "The account is locked out");
}
```

Here, we fetch the user from the database, create a link and content for the email message and send that email to the user. It is a good practice to send an email to the user. By doing that, we encourage them to act proactively. They can reset the password or report that something is wrong because they didn’t try to log in, which could mean that someone is trying to hack their account.

For Duende, we first have to inject the email service in the **Login** page class:

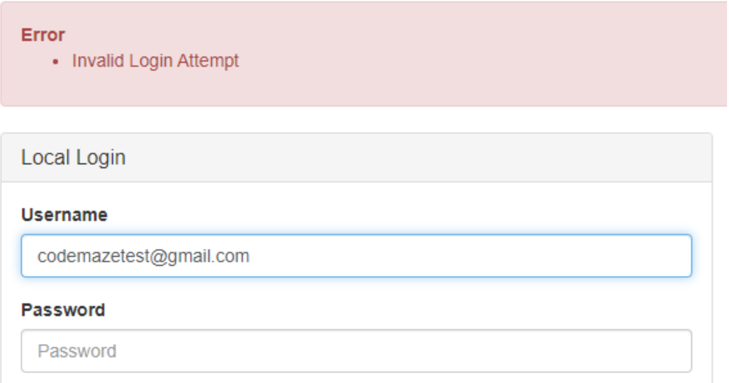
```
private readonly IEmailSender _emailSender;

...

public Index(
    ...
    UserManager<User> userManager, SignInManager<User> signInManager,
    IEmailSender emailSender)
{
    ...
    _emailSender = emailSender;
}
```

And then, we can add the required private method:

```
private async Task HandleLockout(string email, string returnUrl)
{
    var user = await _userManager.FindByEmailAsync(email);
    var forgotPassLink = Url.Page("/Account/ForgotPassword/ForgotPassword",
        null, new { returnUrl }, Request.Scheme);
    var content = string.Format(@"Your account is locked out,
    to reset your password, please click this link: {0}", forgotPassLink);

    var message = new Message(new string[] { user.Email },
        "Locked out account information", content, null);
    await _emailSender.SendEmailAsync(message);

    ModelState.AddModelError("", "The account is locked out");
}
```

14.3 Testing Lockout

If we login with the wrong credentials, we are going to get the familiar error message:



Error

- Invalid Login Attempt

Local Login

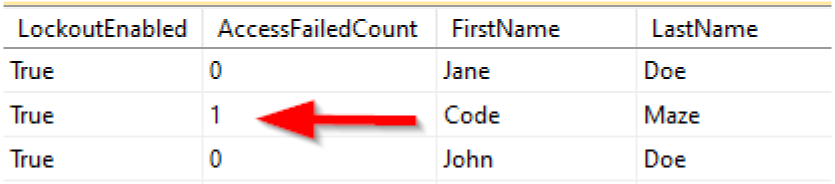
Username

codemazetest@gmail.com

Password

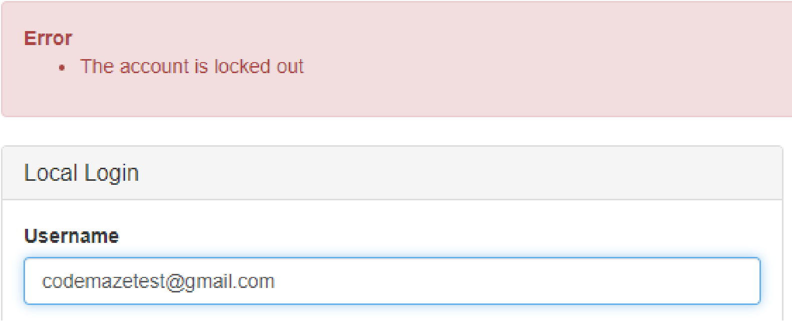
Password

If we inspect the database, we are going to see increased value for the **AccessFailedCount** column:



LockoutEnabled	AccessFailedCount	FirstName	LastName
True	0	Jane	Doe
True	1	Code	Maze
True	0	John	Doe

Now, if we try to log in with the wrong credentials two more times, we can see our account got locked out:



Error

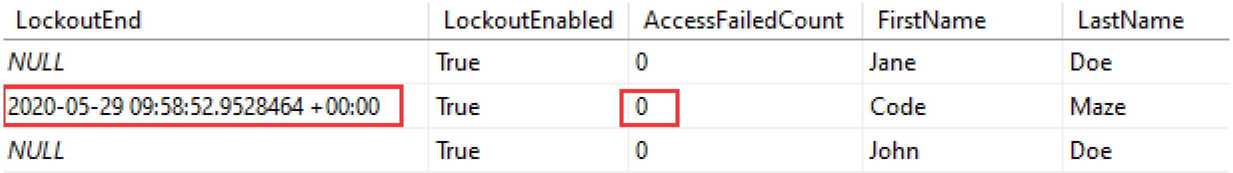
- The account is locked out

Local Login

Username

codemazetest@gmail.com

We can confirm that in the database as well:



LockoutEnd	LockoutEnabled	AccessFailedCount	FirstName	LastName
NULL	True	0	Jane	Doe
2020-05-29 09:58:52.9528464 +00:00	True	0	Code	Maze
NULL	True	0	John	Doe

If we check our email, we will find a link to the **ForgotPassword** action. The rest of the process is the same as explained in the Reset Password section of this book.

Now, there is one important thing to mention. We have two situations here.

Our project is a good example of the first situation. After the user resets the password, they will have to wait for the lockout period to expire to try to log in again. If you want this kind of behavior, you can leave the code as-is.

In the second situation, the user account gets unlocked as soon as the password is reset. If you want a behavior like this one, you should modify the HttpPost **ResetPassword** action. With the **await _userManager.IsLockedOutAsync(user);** expression you can check if the account is locked out, and with the **await _userManager.SetLockoutEndDateAsync(user, new DateTimeOffset(dateInThePast));** expression, you can set the date in the past, which will unlock the account.

If your project requires the second approach, the code should look something like this:

```
 [HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> ResetPassword(ResetPasswordModel resetPasswordModel,
string returnUrl)
{
    ...
    if (!resetPassResult.Succeeded)
    {
        ...
    }

    if(await _userManager.IsLockedOutAsync(user))
    {
        await _userManager.SetLockoutEndDateAsync(user, new DateTimeOffset(new
DateTime(1000, 1, 1, 1, 1, 1)));
    }

    return RedirectToAction(nameof(ResetPasswordConfirmation), new { returnUrl });
}
```

The same code can be used for Duende as well in the `ResetPassword` page class in the `OnPost` method.

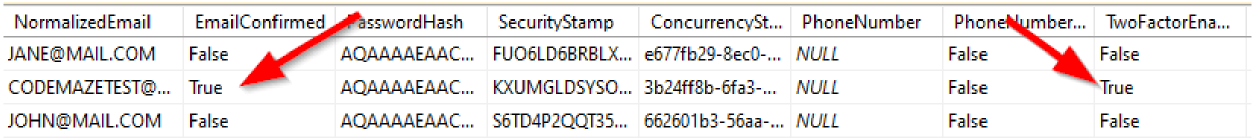
As we said, any date in the past will unlock the account immediately.

15 Two-Step Verification

The two-step verification is a process where a user enters their credentials and, after successful password validation, receives an OTP (one-time-password) via email or SMS. That OTP has to be entered in the Two-Step Verification form on our site to log in successfully.

15.1 Enabling Verification Process

The first thing we have to do is to edit our user in the AspNetUsers table by setting the TwoFactorEnabled column to true:



NormalizedEmail	EmailConfirmed	PasswordHash	SecurityStamp	ConcurrencySt...	PhoneNumber	PhoneNumber...	TwoFactorEna...
JANE@MAIL.COM	False	AQAAAAEAAC...	FUO6LD6BRBLX...	e677fb29-8ec0-...	*NULL*	False	False
CODEMAZETEST@...	True	AQAAAAEAAC...	KXUMGLDSYSO...	3b24ff8b-6fa3-...	*NULL*	False	True
JOHN@MAIL.COM	False	AQAAAAEAAC...	S6TD4P2QQT35...	662601b3-56aa-...	*NULL*	False	False

The confirmed email is also a requirement, but we have already done that.

You can always set the value for the **TwoFactorEnabled** column from the code during the registration process:

```
await _userManager.SetTwoFactorEnabledAsync(user, true);
```

Now, we can modify the **Login** action:

```
if (result.IsLockedOut)
{
    await HandleLockout(model.UserName, model.ReturnUrl);
}
if(result.RequiresTwoFactor)
{
    return RedirectToAction(nameof(LoginTwoStep),
        new { Email = model.Username, model.RememberLogin, model.ReturnUrl });
}
else
{
    await _events.RaiseAsync(...)
    ModelState.AddModelError(string.Empty, AccountOptions.InvalidLoginAttempt);
}
```

For Duende, the code is slightly different:

```
if (result.IsLockedOut)
{
    await HandleLockout(Input.Username, Input.ReturnUrl);
}
if (result.RequiresTwoFactor)
{
    return RedirectToPage("/Account/Login/LoginTwoStep",
        new { Email = Input.Username, Input.RememberLogin, Input.ReturnUrl });
}
```

One of the properties the **result** variable contains is the **RequiresTwoFactor** property. The **PasswordSignInAsync** method will set that property to true if the **TwoFactorEnabled** column for the current user is set to true. Also, the **Succeeded** property will be set to false. Therefore, we check if the **RequiresTwoFactor** property is true and if it is, we redirect the user to a different action/page with the email (we use the email for the username), rememberLogin, and ReturnUrl parameters.

Now, let’s create the **TwoStepModel** class in the **Entities/ViewModels** folder:

```
public class TwoStepModel
{
    [Required]
    [DataType(DataType.Text)]
    public string TwoFactorCode { get; set; }
    public bool RememberLogin { get; set; }
}
```

Next, let’s add new required actions for the IS4 project:

```
[HttpGet]
public async Task<IActionResult> LoginTwoStep(string email, bool rememberLogin, string
returnUrl)
{
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> LoginTwoStep(TwoStepModel twoStepModel, string
returnUrl, string email)
{
    return View();
}
```

Or for Duende, we have to create a new page inside the **Account\Login** folder:

```
[AllowAnonymous]
[BindProperties]
public class LoginTwoStepModel : PageModel
{
    private readonly UserManager<User> _userManager;
    private readonly IEmailSender _emailSender;
    private readonly SignInManager<User> _signInManager;

    public TwoStepModel Input { get; set; }
    public string ReturnUrl { get; set; }
    public string Email { get; set; }

    public LoginTwoStepModel(UserManager<User> userManager, IEmailSender emailSender,
        SignInManager<User> signInManager)
    {
        _userManager = userManager;
        _emailSender = emailSender;
        _signInManager = signInManager;
    }


    public async Task<IActionResult> OnGet(string email, bool rememberLogin, string
returnUrl)
    {
        ReturnUrl = returnUrl;
        Email = email;
        Input = new TwoStepModel { RememberLogin = rememberLogin };

        return Page();
    }

    public async Task<IActionResult> OnPost()
    {
        return Page();
    }
}
```

Excellent.

We have prepared everything for the two-step verification process. So, let’s implement it.

15.2 Two-Step Implementation

With everything in place, we are ready to modify the HttpGet **LoginTwoStep** action:

```
[HttpGet]
public async Task<IActionResult> LoginTwoStep(string email, bool rememberLogin, string
returnUrl)
{
    var user = await _userManager.FindByEmailAsync(email);
    if (user == null)
    {
        return RedirectToAction(nameof(Error), new { returnUrl });
    }

    var providers = await _userManager.GetValidTwoFactorProvidersAsync(user);
    if (!providers.Contains("Email"))
    {
        return RedirectToAction(nameof(Error), new { returnUrl });
    }

    var token = await _userManager.GenerateTwoFactorTokenAsync(user, "Email");

    var message = new Message(new string[] { email }, "Authentication token", token,
null);
    await _emailSender.SendEmailAsync(message);

    ViewData["RouteData"] = new Dictionary<string, string>
    {
        { "returnUrl", returnUrl },
        { "email", email }
    };
    return View();
}
```

We check if the current user exists in the database. If that’s not the case, we display the error page. But if we do find the user, we have to check if there is a provider for Email because we want to send our two-step code by using an email message. After that check, we just create a token with the **GenerateTwoFactorTokenAsync** method and send the email message.

For Duende, we have to modify the OnGet method:

```
public async Task<IActionResult> OnGet(string email, bool rememberLogin, string
returnUrl)
{
    ReturnUrl = returnUrl;
    Email = email;
    Input = new TwoStepModel { RememberLogin = rememberLogin };

    var user = await _userManager.FindByEmailAsync(email);
    if (user == null)
    {
        return RedirectToPage("/Account/Error", new { ReturnUrl });
    }
    var providers = await _userManager.GetValidTwoFactorProvidersAsync(user);
    if (!providers.Contains("Email"))
    {
        return RedirectToPage("/Account/Error", new { ReturnUrl });
    }

    var token = await _userManager.GenerateTwoFactorTokenAsync(user, "Email");

    var message = new Message(new string[] { email }, "Authentication token", token, 
null);
    await _emailSender.SendEmailAsync(message);

    return Page();
}
```

Now, let’s create a view for this action:

```
@model CompanyEmployees.IDP.Entities.ViewModels.TwoStepModel

<h2>LoginTwoStep</h2>

<div class="row">
    <div class="col-md-4">
        <form asp-action="LoginTwoStep"
             asp-all-route-data="(Dictionary<string, string>)@ViewData["RouteData"]">
            <div class="form-group">
                <partial name="_ValidationSummary" />
                <label asp-for="TwoFactorCode" class="control-label"></label>
                <input asp-for="TwoFactorCode" class="form-control" />
                <span asp-validation-for="TwoFactorCode" class="text-danger"></span>
            </div>
            <input type="hidden" asp-for="RememberLogin" />
            <div class="form-group">
                <input type="submit" value="Login" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

For Duende, we have to modify the .cshtml file:

```
@page
@model CompanyEmployees.IDP.Pages.Account.Login.LoginTwoStepModel

<h2>LoginTwoStep</h2>

<div class="row">
    <div class="col-md-4">
        <form asp-page="/Account/Login/LoginTwoStep">
            <div class="form-group">
                <partial name="_ValidationSummary" />
                <label asp-for="Input.TwoFactorCode" class="control-label"></label>
                <input asp-for="Input.TwoFactorCode" class="form-control" />
                <span asp-validation-for="Input.TwoFactorCode" class="text-
danger"></span>
            </div>
            <input type="hidden" asp-for="Input.RememberLogin" />
            <input type="hidden" asp-for="@Model.ReturnUrl" />
            <input type="hidden" asp-for="@Model.Email" />
            <div class="form-group">
                <input type="submit" value="Login" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

After the vew creation, we can modify the HttpPost **LoginTwoStep** action:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> LoginTwoStep(TwoStepModel twoStepModel, string
returnUrl, string email)
{
    if (!ModelState.IsValid)
    {
        return View(twoStepModel);
    }

    var user = await _signInManager.GetTwoFactorAuthenticationUserAsync();
    if (user == null)
    {
        return RedirectToAction(nameof(Error), new { returnUrl });
    }

    var result = await _signInManager.TwoFactorSignInAsync("Email",
twoStepModel.TwoFactorCode, twoStepModel.RememberLogin, rememberClient: false);
    if (result.Succeeded)
    {
        return this.LoadingPage("Redirect", returnUrl);
    }
    else if (result.IsLockedOut)
    {
        await HandleLockout(email, returnUrl);
        return View(twoStepModel);
    }
    else
    {
        return RedirectToAction(nameof(Error), new { returnUrl });
    }
}
```

First, we check the model validity. If the model is valid, we use the **GetTwoFactorAuthenticationUserAsync** method to get the current user. We do that with the help of our **Identity.TwoFactorUserId** cookie (it was automatically created by Identity in the HttpGet **LoginTwoStep** action). This will prove that the user indeed went through all the verification steps to get to this point. If we find that user, we use the **TwoFactorSignInAsync** method to verify the **TwoFactorToken** value and sign in the user.

If the sign-in was successful, we use the returnUrl parameter to redirect the user. Otherwise, we carry out additional checks on the result variable and force appropriate actions.

As usual, we have a bit different code for Duende:

```
public async Task<IActionResult> OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var user = await _signInManager.GetTwoFactorAuthenticationUserAsync();
    if (user == null)
    {
        return RedirectToPage("/Account/Error", new { ReturnUrl });
    }

    var result = await _signInManager.TwoFactorSignInAsync("Email", Input.TwoFactorCode, 
Input.RememberLogin, rememberClient: false);
    if (result.Succeeded)
    {
        return this.LoadingPage(ReturnUrl);
    }
    else if (result.IsLockedOut)
    {
        await HandleLockout(Email, ReturnUrl);

        return Page();
    }
    else
    {
        return RedirectToPage("/Account/Error", new { ReturnUrl });
    }
}
```

But now, we have a small issue. The **HandleLockout** method exists inside the **Login** page class as a private method, and we don’t want to recreate it in this class again.

So, what we can do, as one of the solutions, is to create a new **BasePage** class in the root of the **Account** folder:

```
public class BasePage : PageModel
{
    private readonly UserManager<User> _userManager;
    private readonly IEmailSender _emailSender;

    public BasePage(UserManager<User> userManager, IEmailSender emailSender)
    {
        _userManager = userManager;
        _emailSender = emailSender;
    }

    protected async Task HandleLockout(string email, string returnUrl)
    {
        var user = await _userManager.FindByEmailAsync(email);
        var forgotPassLink = Url.Page("/Account/ForgotPassword/ForgotPassword",
            null, new { returnUrl }, Request.Scheme);

        var content = string.Format(@"Your account is locked out,
    to reset your password, please click this link: {0}", forgotPassLink);

        var message = new Message(new string[] { user.Email },
            "Locked out account information", content, null);
        await _emailSender.SendEmailAsync(message);

        ModelState.AddModelError("", "The account is locked out");
    }
}
```

The most important part is that this class implements the **PageModel** class. The rest of the code is the same.

Now, we can modify the **Login** page class by removing the **HandleLockout** method and using a different base class:

```
[SecurityHeaders]
[AllowAnonymous]
public class Index : BasePage
{
    ...

    public Index(
        ...
        UserManager<User> userManager, SignInManager<User> signInManager,
        IEmailSender emailSender) : base(userManager, emailSender)
```

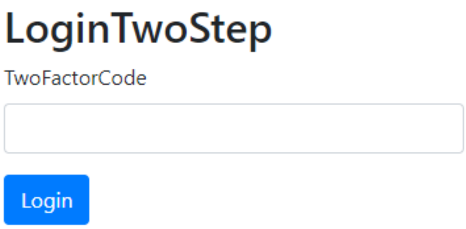
We have to do the same for the **LoginTwoStep** page class:

```
[AllowAnonymous]
[BindProperties]
public class LoginTwoStepModel : BasePage
{
    ...

    public LoginTwoStepModel(UserManager<User> userManager, IEmailSender emailSender,
        SignInManager<User> signInManager) : base (userManager, emailSender)
```

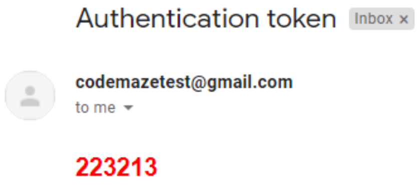
15.3 Testing Two-Step Verification

With everything in place, we can test our functionality.

As soon as we enter valid credentials, we are going to see a new view:



Now, we can check our email and look for the sent code:

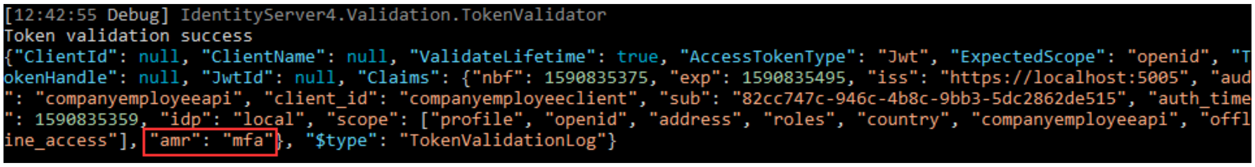


codemazetest@gmail.com

223213

Once we enter a valid token in the LoginTwoStep form, we are going to be redirected to the consent page. If we click the Allow button, we are going to be redirected to the Home page.

So, everything works great. If we inspect the console log window, we can see the value for the **amr** property (Authentication Method Reference) is now **mfa** (Multiple-factor authentication):



```
[12:42:55 Debug] IdentityServer4.Validation.TokenValidator
Token validation success
{"ClientId": null, "ClientName": null, "ValidateLifetime": true, "AccessTokenType": "Jwt", "ExpectedScope": "openid", "T
okenHandle": null, "JwtId": null, "Claims": {"nbf": 1590835375, "exp": 1590835495, "iss": "https://localhost:5005", "aud
": "companyemployeeapi", "client_id": "companyemployeeclient", "sub": "82cc747c-946c-4b8c-9bb3-5dc2862de515", "auth_time
": 1590835359, "idp": "local", "scope": ["profile", "openid", "address", "roles", "country", "companyemployeeapi", "offl
ine_access"], "amr": "mfa"}, "$type": "TokenValidationLog"}
```

16 External Provider with IdentityServer4 and ASP.NET Core Identity

Using an external provider when logging in to the application is quite common. This enables us to log in with our external accounts such as Google, Facebook, etc.

In this part of the book, we are going to learn how to configure an external identity provider in our IdentityServer4 application and how to use a Google account to successfully authenticate. Of course, in a very similar way, you can configure the system to use any other external account.

There is one important thing to keep in mind here. Once an external user logs in to our system, they will always have an identifier that is unique for that user in our system. That means that the user could have different Ids for different sites but for our site, that Id will always be the same.

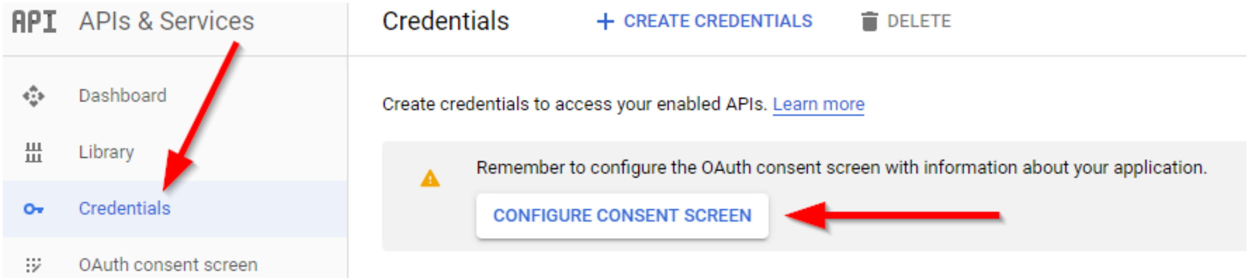
Let's get started.

16.1 Getting ClientId and ClientSecret

The first thing we have to do is to navigate to the Integrate Google Sign-In page. In the Create authorization credentials section, we can see the Credentials page link with additional explanations below it.

After clicking that link, we are going to be redirected to the page for creating our credentials. If we don't have any project created, we have to click the **create project** button at the top-right corner of the screen, add a project name, and click the create button.

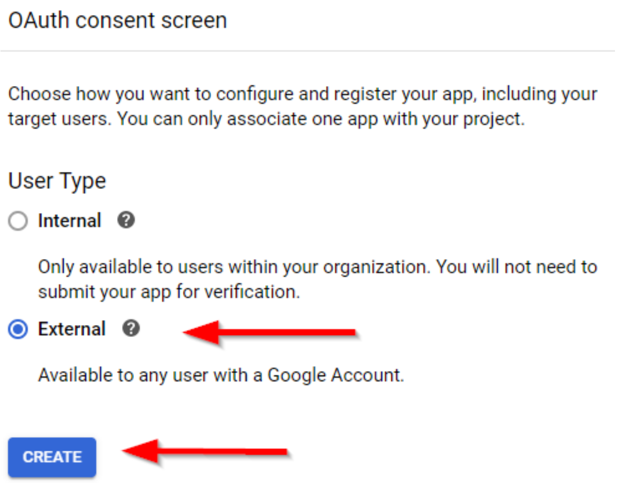
Then in the Credentials menu, we have to configure a consent screen:



Create credentials to access your enabled APIs. Learn more

Remember to configure the OAuth consent screen with information about your application.

Then, we need to choose the External user type:



Choose how you want to configure and register your app, including your target users. You can only associate one app with your project.

User Type

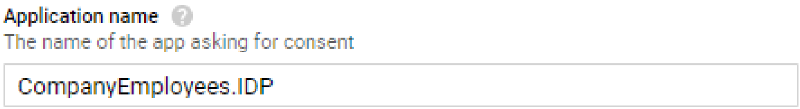
○ Internal

Only available to users within your organization. You will not need to submit your app for verification.

● External

Available to any user with a Google Account.

And finally, we have to add the name of the application, some required emails, and then just leave everything else to default (scopes, test users...):

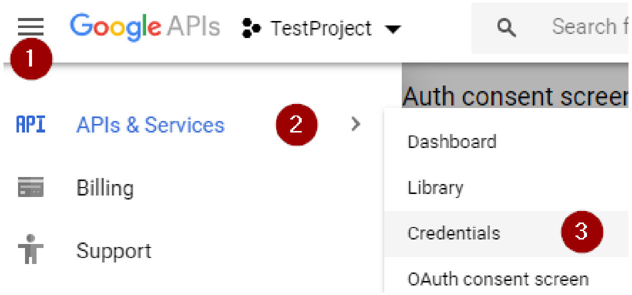


Application name

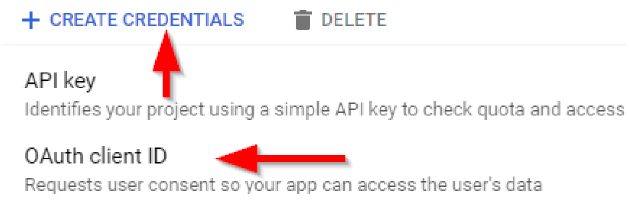
The name of the app asking for consent

CompanyEmployees.IDP

After that, we can navigate back to the Credentials page:



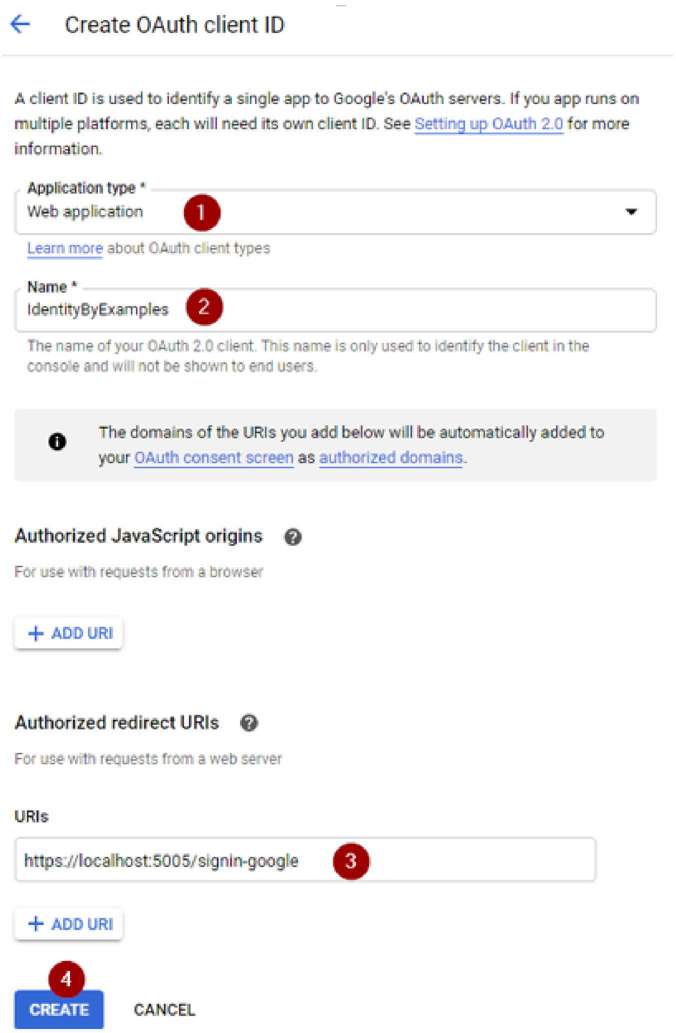
There, we can click the **create credentials** link menu and choose the **OAuth client ID**:



API key
Identifies your project using a simple API key to check quota and access

OAuth client ID
Requests user consent so your app can access the user's data

Now, we have to choose the Application type, Name and add an authorized redirect URI for our application:



Create OAuth client ID

A client ID is used to identify a single app to Google's OAuth servers. If you app runs on multiple platforms, each will need its own client ID. See Setting up OAuth 2.0 for more information.

Application type *
Web application

Learn more about OAuth client types

Name *
IdentityByExamples

The name of your OAuth 2.0 client. This name is only used to identify the client in the console and will not be shown to end users.

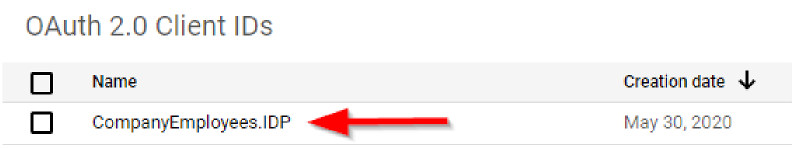
The domains of the URIs you add below will be automatically added to your OAuth consent screen as authorized domains.

Authorized JavaScript origins
For use with requests from a browser

Authorized redirect URIs
For use with requests from a web server

URIs
https://localhost:5005/signin-google

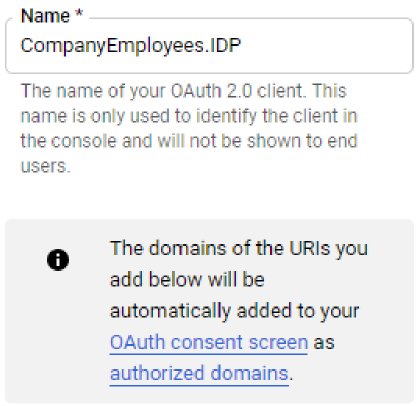
Once we click the Create button, we will get the ClientID and ClientSecret values. You can save them but Google will save them for you anyway. To find these credentials, all we have to do is to click on the created web application:



OAuth 2.0 Client IDs

□	Name	Creation date ↓
□	CompanyEmployees.IDP	May 30, 2020

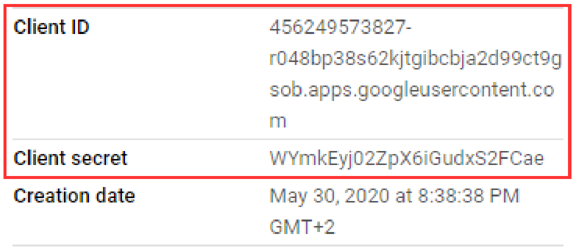
There, we are going to see the credentials for sure:



Name *
CompanyEmployees.IDP

The name of your OAuth 2.0 client. This name is only used to identify the client in the console and will not be shown to end users.

The domains of the URIs you add below will be automatically added to your OAuth consent screen as authorized domains.

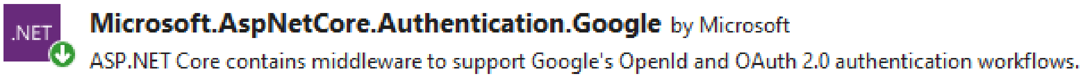


Client ID	456249573827-r048bp38s62kjtgibcbja2d99ct9gsob.apps.googleusercontent.com
Client secret	WYmkEyj02ZpX6iGudxS2FCae
Creation date	May 30, 2020 at 8:38:38 PM GMT+2

And that’s it. We can move on to the project configuration.

16.2 Configuring External Identity Provider

To be able to use Google as our external provider, we have to install the **Microsoft.AspNetCore.Authentication.Google** package:



Microsoft.AspNetCore.Authentication.Google by Microsoft
ASP.NET Core contains middleware to support Google's OpenId and OAuth 2.0 authentication workflows.

After the installation is completed, we are going to modify the **ConfigureServices** method:

```
services.Configure<EmailConfirmationTokenProviderOptions>(opt =>
    opt.TokenLifespan = TimeSpan.FromDays(3));

services.AddAuthentication()
    .AddGoogle(options =>
    {
        options.ClientId = "your id";
        options.ClientSecret = "your secret";
    });
```

The **AddIdentity** method configures default scheme settings. But, the **AddAuthentication** method allows configuring different authentication options, like Google for example. That’s why this method must be placed after the **AddIdentity** method.

This is completely the same for Duende where you just have to use the **builder** object to access the **Services** property.

16.3 External Provider Implementation

Now, if we inspect the **Login.cshtml** file in the **Views/Account** folder, we can see this part of the code:

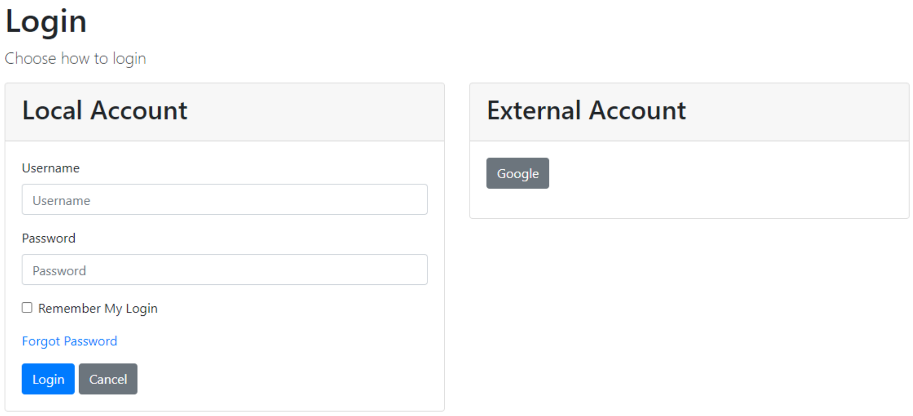
```
@if (Model.VisibleExternalProviders.Any())
{
    <div class="col-md-6 col-sm-6 external-providers">
        <div class="panel panel-default">
            <div class="panel-heading">
                <h3 class="panel-title">External Login</h3>
            </div>
            <div class="panel-body">
               <ul class="list-inline">
                  @foreach (var provider in Model.VisibleExternalProviders)
                  {
                     <li>
                         <a class="btn btn-default"
                            asp-controller="External"
                            asp-action="Challenge"
                            asp-route-provider="@provider.AuthenticationScheme"
                            asp-route-returnUrl="@Model.ReturnUrl">
                            @provider.DisplayName
                         </a>
                     </li>
                  }
               </ul>
            </div>
        </div>
    </div>
}
```

This code is slightly different for Duende in the Account/Login/Index.cshtml file:

```
@if (Model.View.VisibleExternalProviders.Any())
```

```
{
    <div class="col-sm-6">
        <div class="card">
            <div class="card-header">
                <h2>External Account</h2>
            </div>
            <div class="card-body">
                <ul class="list-inline">
                    @foreach (var provider in Model.View.VisibleExternalProviders)
                    {
                        <li class="list-inline-item">
                            <a class="btn btn-secondary"
                                asp-page="/ExternalLogin/Challenge"
                                asp-route-scheme="@provider.AuthenticationScheme"
                                asp-route-returnUrl="@Model.Input.ReturnUrl">
                                @provider.DisplayName
                            </a>
                        </li>
                    }
                </ul>
            </div>
        </div>
    </div>
}
```

This code will iterate through all the registered external providers and add a button for them on the right side of the Login page:



In addition to this file, we have the **ExternalController** file in the **Quickstart/Account** folder for the IS4 project. But, the code in this file is not valid for us, since we are now using ASP.NET Core Identity for the user management actions. Moreover, we don’t have the test users anymore and this file works with test users.

For Duende we have the `Pages\ExternalLogin` folder. There, we can find two pages: `Callback` and `Challenge`. The code from the `Challenge` page is perfect for us and we will leave it as is. But the code in the `Callback` page is not, and this is what we are going to remove.

Since modifying this file to support ASP.NET Core Identity would be quite messy and hard to explain without additional confusion, we are going to remove this file/page from the project.

So, for IS4 project, let's open the **16-External Provider** folder in our source code and in the root of that folder, we can find the **ExternalController** file.

This file is taken from the Identity + IS4 template, modified to suit our needs, and made more readable and easier to maintain.

Let's copy this file and paste it into our current project at the **Quickstart/Account** location.

For Duende, we have to navigate to the **CompanyEmployees.IDP/Pages/ExternalLogin** folder and copy the `Callback.cshtml` and `Callback.cshtml.cs` files and paste them to the same folder in your project.

We could have created our custom logic for the external provider, but, since we already have it implemented (in the template project that supports Identity), we have decided to use that file and modify it to suit our needs. You can find the command for creating this template in section 10 of this book.

If you open the file, you will see a lot of code broken into smaller methods for better readability. So, let's explain it.

As soon as we click the Google button, we are going to hit the **Challenge** action (For Duende it is `OnGet` in the `Challenge` page class). In this action, we create **AuthenticationProperties** object required for the challenge window and call the **Challenge** action/page which returns a **ChallengeResult**.

The **Callback** action/page is going to be called as soon as we choose the external account we want to log in within the Challenge window. This action contains the main logic. With the **AuthenticateExternalScheme** method, we authenticate the external scheme and return the result. Then, with the **FindUserFromExternalProviderAsync** method, we return the user with the already associated external login account. Additionally, we return the provider, providerUserId, and claims. If this method doesn't find the user, it will return null.

If null is returned, we call the **AutoProvisionUserAsync** method. In that method, we extract the name and email claims and try to fetch the user by the email claim. If we find the user, we simply attach the external login provider to the user's account. We do that by inserting a new row in the **AspNetUserLogins** table. But, if we don't find the user, we create a new one and attach the claims and the external provider.

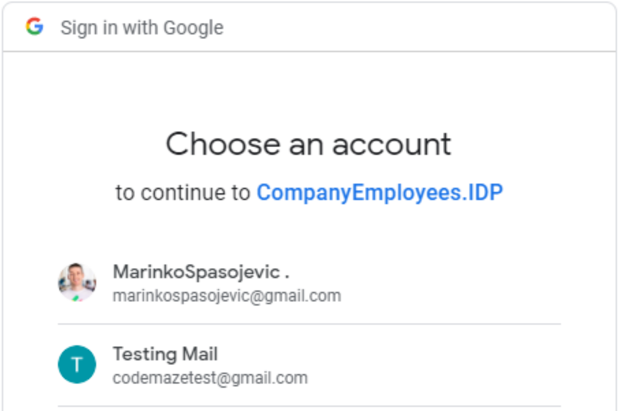
Then, back in the **Callback** method, we extract the additional claims, local sign-in properties, and the name claim and use them to sign in the user locally. We do that by calling the **LocalSignIn** method.

Finally, we redirect the user to the Home page.

If you want to inspect the code yourself, you can always place a breakpoint in the **Callback** method and click the Google button on the Login screen. Then, you can inspect the code line by line.

16.4 Testing

We are going to test this logic with an existing user first and then with a new one. So, let’s start the client and IDP applications, navigate to the Login screen and click the Google button:



Sign in with Google

Choose an account

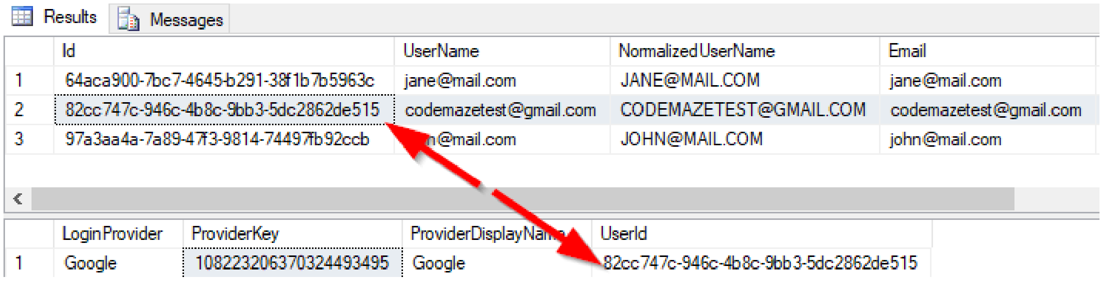
to continue to CompanyEmployees.IDP

MarinkoSpasojevic .
marinkospasojevic@gmail.com

Testing Mail
codemazetest@gmail.com

We are going to choose the Testing Mail account first. Once we click it, are going to be redirected to the consent screen. In some cases Google might ask for the phone verification process. If we click the Allow button on the consent screen, we are going to see the Home screen, which means that we are logged in.

Now, let’s inspect the database:

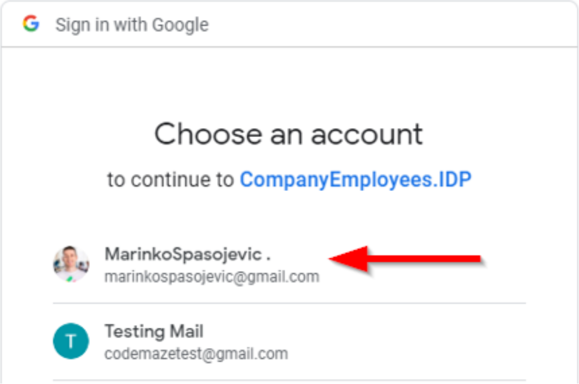


	Id	UserName	NormalizedUserName	Email
1	64aca900-7bc7-4645-b291-38f1b7b5963c	jane@mail.com	JANE@MAIL.COM	jane@mail.com
2	82cc747c-946c-4b8c-9bb3-5dc2862de515	codemazetest@gmail.com	CODEMAZETEST@GMAIL.COM	codemazetest@gmail.com
3	97a3aa4a-7a89-47f3-9814-74497fb92ccb	n@mail.com	JOHN@MAIL.COM	john@mail.com

	LoginProvider	ProviderKey	ProviderDisplayNa	UserId
1	Google	108223206370324493495	Google	82cc747c-946c-4b8c-9bb3-5dc2862de515

We can see that we have only attached a new provider to the existing account.

Now, let’s log out and then log in with the external provider, but this time we are going to choose the first option in the Challenge window:



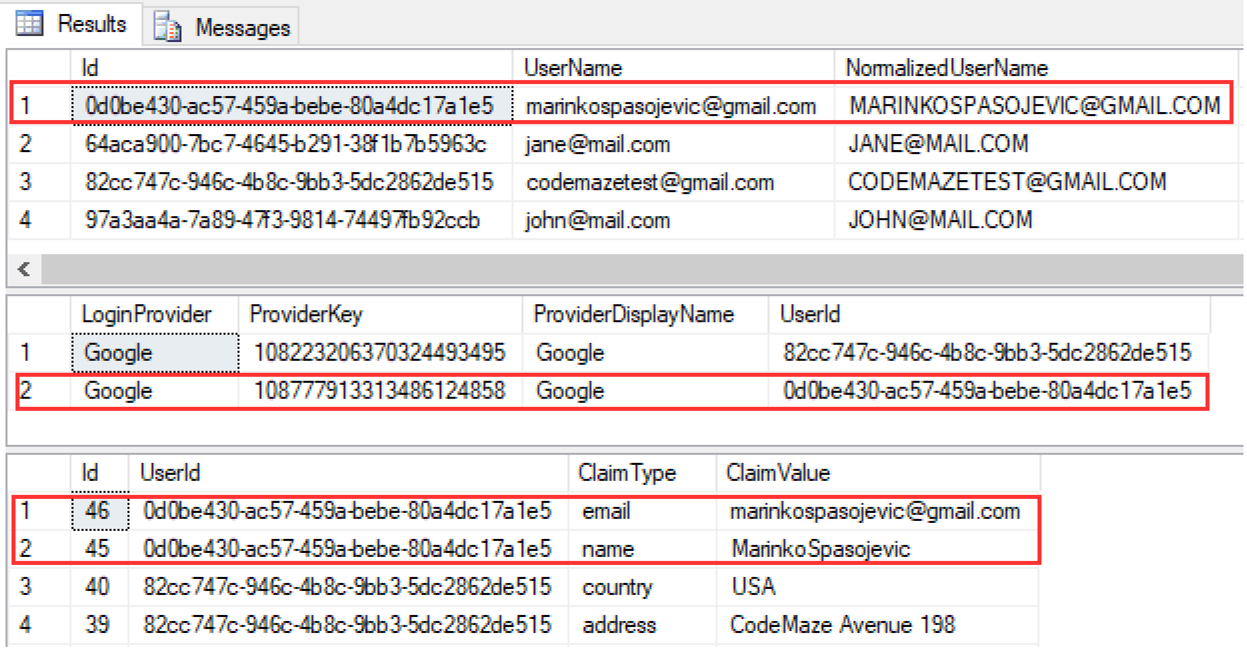
Choose an account

to continue to CompanyEmployees.IDP

MarinkoSpasojevic .
marinkospasojevic@gmail.com

Testing Mail
codemazetest@gmail.com

After choosing that account, we are going to see the consent screen again and after clicking the Allow button, we are going to see the Home screen. So, we have successfully logged in again, but, if we inspect the database, we can see a new account has been created with the external provider and claims:



	Id	UserName	NormalizedUserName
1	0d0be430-ac57-459a-bebe-80a4dc17a1e5	marinkospasojevic@gmail.com	MARINKOSPASOJEVIC@GMAIL.COM
2	64aca900-7bc7-4645-b291-38f1b7b5963c	jane@mail.com	JANE@MAIL.COM
3	82cc747c-946c-4b8c-9bb3-5dc2862de515	codemazetest@gmail.com	CODEMAZETEST@GMAIL.COM
4	97a3aa4a-7a89-47f3-9814-74497fb92ccb	john@mail.com	JOHN@MAIL.COM

	LoginProvider	ProviderKey	ProviderDisplayName	UserId
1	Google	108223206370324493495	Google	82cc747c-946c-4b8c-9bb3-5dc2862de515
2	Google	108777913313486124858	Google	0d0be430-ac57-459a-bebe-80a4dc17a1e5

	Id	UserId	ClaimType	ClaimValue
1	46	0d0be430-ac57-459a-bebe-80a4dc17a1e5	email	marinkospasojevic@gmail.com
2	45	0d0be430-ac57-459a-bebe-80a4dc17a1e5	name	MarinkoSpasojevic
3	40	82cc747c-946c-4b8c-9bb3-5dc2862de515	country	USA
4	39	82cc747c-946c-4b8c-9bb3-5dc2862de515	address	CodeMaze Avenue 198

Awesome job.

With this out of the way, we have finished our ASP.NET Core security journey.

You have a vast knowledge of both IdentityServer4 and ASP.NET Core Identity implementations for securing your applications.

Best regards and all the best from the Code Maze team.